

Your Brain on Design Patterns

# Head First Design Patterns

Avoid those  
embarrassing  
coupling mistakes



Discover the secrets  
of the Patterns Guru



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



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



Load the patterns  
that matter straight  
into your brain



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

O'REILLY®

Eric Freeman & Elisabeth Freeman  
with Kathy Sierra & Bert Bates

# Head First Design Patterns

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

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

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

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

"This book is close to perfect, because of the way it combines expertise and readability. It speaks with authority and it reads beautifully."

—David Gelernter, Professor of Computer Science, Yale University

"One of the funniest and smartest books on software design I've ever read."

—Aaron LaBerge,  
VP Technology, ESPN.com

Software Development/Java

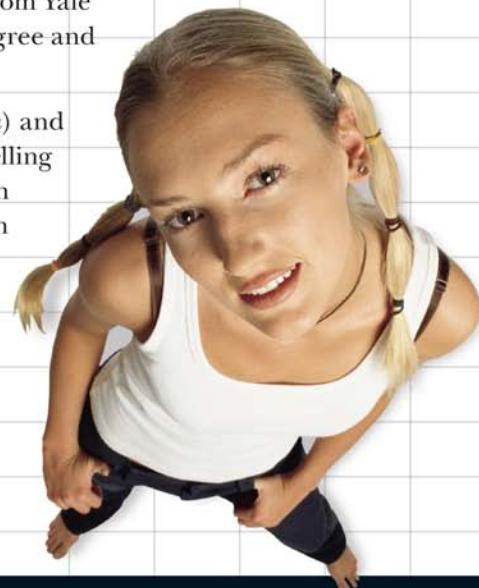
You know you don't want to reinvent the wheel (or worse, a flat tire), so you look to design patterns—the lessons learned by those who've faced the same software design problems. With design patterns, you get to take advantage of the best practices and experience of others, so that you can spend your time on...something else. Something more challenging. Something more complex. Something more *fun*. You want to learn:

- The patterns that *matter*
- *When* to use them, and *why*
- How to *apply* them to your own designs, *right now*
- When *not* to use them (how to avoid pattern fever)
- OO design principles on which patterns are based

Most importantly, you want to learn design patterns in a way that won't put you to sleep. If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. Using the latest research in neurobiology, cognitive science, and learning theory, *Head First Design Patterns* will load patterns into your brain in a way that sticks. In a way that makes you better at solving software design problems, and better at speaking the language of patterns with others on your team.

**Eric Freeman** and **Elisabeth Freeman** are authors, educators, and technology innovators. After four years leading digital media and Internet efforts at the Walt Disney Company, they're applying some of that pixie dust to their own media, including this book. Eric and Elisabeth both hold computer science degrees from Yale University: Elisabeth holds an M.S. degree and Eric a Ph.D.

**Kathy Sierra** (founder of *javaranch.com*) and **Bert Bates** are the creators of the best-selling Head First series and developers of Sun Microsystems Java developer certification exams.



[www.oreilly.com](http://www.oreilly.com)

US \$44.95 CAN \$65.95

ISBN-10: 0-596-00712-4

ISBN-13: 978-0-596-00712-6



5 4 4 9 5

9 780596 007126

O'REILLY®

## Praise for Head First Design Patterns

received the book yesterday and started to read it on the way home... and couldn't stop. took it to the gym and e pect people saw me smiling a lot while was e ergizing and reading. his is tres cool'. t is fun but they cover a lot of ground and they are right to the point. 'm really impressed.

**r ch amma st shed eer  
a d co a thor o es atter s**

Head First Design Patterns' manages to mi 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 bjectville.

**chard elm coa thor o es atter s w th rest o the  
a o o r r ch amma alph oh so a d oh l ss des**

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

**ard ham e tor o the  
a d o der o the lls de ro p**

his book is close to perfect, because of the way it combines e pertise and readability. t speaks with authority and it reads beautifully. t's one of the very few software books 've ever read that strikes me as indispensable. ( 'd put maybe books in this category, at the outside.)

**a d eler ter ro essor o omp ter c e ce  
ale ers t a d a thor o rrор orlds a d ach e ea t**

ose ive into the realm of patterns, a land where comple things become simple, but where simple things can also become comple . can think of no better tour guides than the Freemans.

**o ats m ra d str al st he ddleware ompa  
ormer h e a a a el st cros stems**

laughed, cried, it moved me.

**a el te ber d tor h e a a et**

My first reaction was to roll on the oor laughing. fter picked myself up, reali ed that not only is the book technically accurate, it is the easiest to understand introduction to design patterns that have seen.

**r. moth . dd ssoc ate ro essor o omp ter c e ce at  
re o tate ers t a d a thor o more tha ado e boo s  
cl d or a a ro rammers**

erry ice runs patterns better than any receiver in the FL, but the Freemans have out run him. eriously...this is one of the funniest and smartest books on software design 've ever read.

**aro a er e ech olo .com**

## **ore 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. His 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.

**or octorow co ed tor o o o  
a d a thor o ow a d t the a c dom  
a d omeo e omes to ow omeo e ea es ow**

here'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 esign 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?

storyteller? An architect or a builder? 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 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 esign is Life, then perhaps it is time we spent some uality cycles thinking about what it is.

Ric and Lisabeth Freeman have intrepidly volunteered to look behind the code curtain for us in Head First esign Patterns. I'm not sure either of them cares all that much about the Play tation or Bo , 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. Ophists and circus barkers need not apply. The t generation literati please come equipped with a pencil.

**e oldste ec t e ce res de t a a rector  
s e l e**

ust 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.

**ra s ala c o der o co r a d ed woosh  
ember o the**

This book combines good humors, great examples, and in depth knowledge of esign 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 esign Patterns not only helps us create reusable and maintainable uality 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.

**ewto ee o der a d d tor h e ssoc at o or omp t  
ach er s omp ters tera me t acmc e.or**

## **ore Praise for Head First Design Patterns**

f there's one subject that needs to be taught better, needs to be more fun to learn, it's design patterns.  
hank goodness for Head First Design Patterns.

From the awesome Head First Java folks, this book uses every conceivable trick to help you understand and remember. Not just loads of pictures pictures of humans, which tend to interest other humans. Surprises everywhere. Stories, because humans love narrative. ( Stories about things like pi a and chocolate. Need we say more?) Plus, it's darned funny.

It also covers an enormous swath of concepts and techniques, including nearly all the patterns you'll use most (observer, decorator, factory, singleton, command, adapter, facade, template method, iterator, composite, state, proxy). Read it, and those won't be just words they'll be memories that tickle you, and tools you own.

### **Il amarda**

fter using Head First Java to teach our freshman how to start programming, was eagerly waiting to see the next book in the series. Head First Design Patterns is that book and I am delighted. I am sure it will quickly become the standard first design patterns book to read, and is already the book I am recommending to students.

**e ederso ssoc ate ro essor o omp ter c e ce rector o the  
ma omp ter teract o ab ers t o ar la d**

usually when reading through a book or article on design patterns I'd have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.

While other books on design patterns are saying, Buehler... Buehler... Buehler... this book is on the boat belting out 'Take it up, baby'

### **r c ehler**

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

### **at sh mar**

## **Praise for the Head First approach**

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

**cott c eal cros stems ha rma res de t a d**

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

**e r old ormer e or eer at cros stems  
o a thor w th ames osl creator o a a  
he a a ro ramm a a e**

### **ther related oo s fro ' eilly**

Learnin Ja a  
Ja a in a N t ell  
Ja a Enterpri e in a N t ell  
Ja a E ample in a N t ell  
Ja a Coo oo  
J2EE De i n Pattern

### **ther oo s in eilly s Head First series**

Head Fir t Ja a  
Head Fir t EJB  
Head Fir t Ser let & JSP  
Head Fir t O ect-Oriented Analy i & De i n  
Head Fir t HTML wit CSS & XHTML  
Head R A a  
Head Fir t PMP  
Head Fir t SQL (2007)  
Head Fir t C# (2007)  
Head Fir t So ware De elopment (2007)  
Head Fir t Ja aScript (2007)

### **e watching for ore oo s in the Head First series**



Wouldn't it be dreamy if  
there was a Design Patterns  
book that was more fun than going  
to the dentist, and more revealing  
than an IRS form? It's probably  
just a fantasy...

O'REILLY®

e i i g      a      ridge      l      aris      e asto ol      ai ei      o yo

# **Head First Design Patterns**

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

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

Printed in the United States of America.

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

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

**Editor:** Mike Loukides

**Cover Designer:** Ellie Volckhausen

**Pattern Wranglers:** Eric Freeman, Elisabeth Freeman

**Facade Decoration:** Elisabeth Freeman

**Strategy:** Kathy Sierra and Bert Bates

**Observer:** Oliver



**Printing History:**

October 2004: First Edition.

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

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

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

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

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

No ducks were harmed in the making of this book.

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

ISBN-10: 0-596-00712-4

ISBN-13: 978-0-596-00712-6

[M]

[7/07]

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

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

# Authors/Developers of Head First Design Patterns

Elisabeth Freeman



Eric Freeman



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

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

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

You can send her email at [beth@wickedlysmart.com](mailto:beth@wickedlysmart.com)

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

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

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

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

Write to him at [eric@wickedlysmart.com](mailto:eric@wickedlysmart.com) or visit his blog at <http://www.ericfreeman.com>

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

Kathy Sierra



Bert Bates



**Kathy** has been interested in learning theory since her days as a game designer (she wrote games for Virgin, MGM, and IBM). She developed much of the Head First format while teaching New Media Authoring for

CITI tension'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 thousands of developers. Kathy is the founder of javaranch.com, which won a Gold and Software Development magazine's Cola Productivity Award. You might catch her teaching Java on the Java Jam Geek Cruise ([geekcruises.com](http://geekcruises.com)).

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

Likes running, skiing, skateboarding, playing with her Icelandic horse, and weird science. dislikes entropy.

You can find her on javaranch, or occasionally blogging on [java.net](http://java.net). Write to her at [kathy@wickedlysmart.com](mailto: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 Nederland, the Weather Channel, and the Arts Entertainment Network (AETN). 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 Gogo server, or you can write to him at [terrapin@wickedlysmart.com](mailto:terrapin@wickedlysmart.com).

# a leo Contents su ar

ntro

v

Welcome to Design Patterns	i tr ti
ceping your objects in the know	t e ser er tter
ecorating objects	t e e r t r tter
Baking with goodness	t e t ry tter
ne of a ind objects	t e i get tter
ncapsulating nvocation	t e mm tter
Being daptive	t e ter e tter s
ncapsulating lgorithms	t e em te et tter
Well managed Collections	t e ter t r m site tter s
he ate of hings	t e t te tter
Controlling object ccess	t e r y tter
Patterns of Patterns	m tter s
Patterns in the real World	etter i i g it tter s
ppendi	e t er tter s

# a leo Contents t e real t ing

## ntro

Here you are tr ing to *learn* something, while here our *brain* is doing ou a favor b ma ing sure the learning doesn't stick. Your brain's thin ing, etter leave room for more important things, li e which wild animals to avoid and whether na ed snowboarding is a bad idea. So how do ou tric our brain into thin ing that our life depends on nowing Design atterns

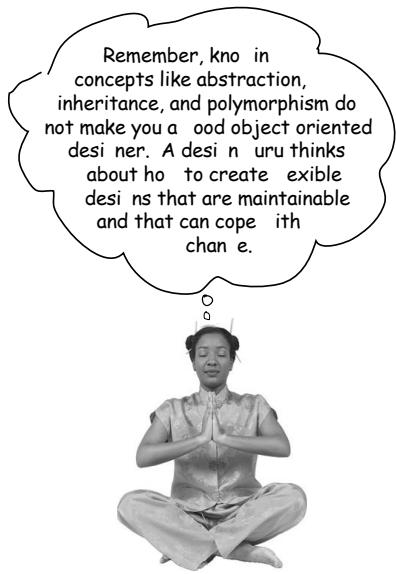
Who is this book for?	vi
We know what your brain is thinking	vii
Metacognition	i
Bend your brain into submission	i
technical reviewers	iv
cnowledgements	v

## intro to Design Patterns

# Introduction to Design Patterns

In this chapter,

you'll learn why (and how) you can exploit wisdom and learn learned other design patterns to make down the same problem road and ride the trip. Before we're done, we'll look at the common design pattern, look at some key OO design principles, and walk through an example of how one pattern works. The key way to use a pattern is to *load your brain* with them and then recognize them in your design and even in application where you can apply them. Instead of reading it with a pattern you get eerie here.



he implements

one thinks about inheritance...

How about an interface?

he one constant in software development

separating what changes from what stays the same

esigning the stuck Behaviors

estimating the stuck code

etting behavior dynamically

the Big Picture on encapsulated behaviors

How can be better than

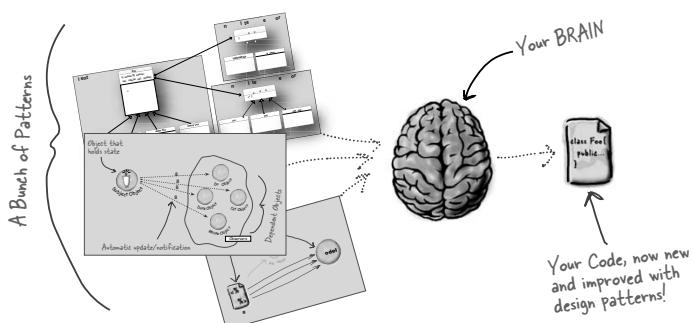
the Strategy Pattern

the power of a shared pattern vocabulary

How do I use design Patterns?

ools for your design toolbox

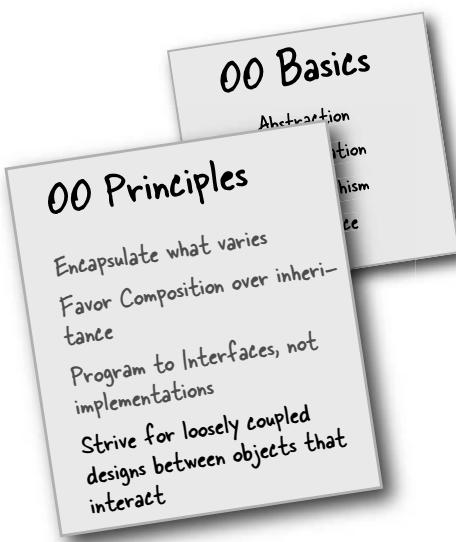
ercise solutions



the observer pattern

## keeping your effects in the now

We've got a pattern that keeps your effect in the now when it comes to objects in memory. It's called the Observer Pattern. Object can even decide at runtime what they want to be kept informed. The Observer Pattern is one of the most easily 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 (yes, that's right, we said coupling). With Observer, you'll get to the Point of Pattern Party.



The Weather Monitoring application

Meet the Observer Pattern

Publishers subscribers Observer Pattern

Five minute drama a subject for observation

the Observer Pattern defined

the power of Loose Coupling

designing the Weather station

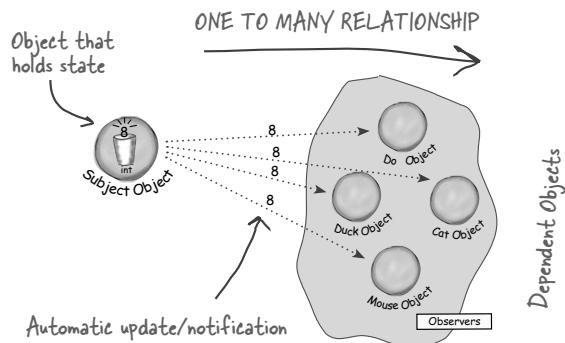
implementing the Weather station

using Java's built-in Observer Pattern

the dark side of java.util.Observable

tools for your design toolbox

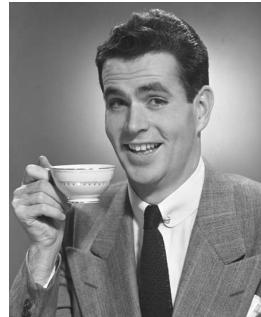
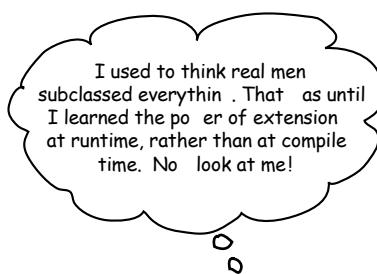
exercise solutions



the Decorator pattern

## Decorating effects

We'll re-examine the typical order of inheritance and you'll learn how to decorate your class at runtime in a more object composition. Why? Once you know the techniques of decoration, you'll be able to impress your colleagues by creating new responsibilities *without changing any code or gestures to the underlying classes*.



Welcome to Tarbus Coffee

he Open/Closed Principle

Meet the Decorator Pattern

Constructing a drink order with decorators

the Decorator Pattern defined

decorating our Beverages

Writing the Tarbus code

Real World Decorators available

Writing your own Java Decorator

Tools for your Design Toolbox

Exercise Solutions

the a tor attorn

## a ing with oodness

There is more to maintain object than in the **ne** operator. We'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to *臭* (smelly) code. And you don't want that, do you? Find out how Factory Pattern can help avoid problems from dependencies.

When you see new, think concrete

objectville Pizza

encapsulating object creation

Building a simple pizza factory

Implementation defined

Framework for the pizza store

allowing the subclasses to decide

Let's make a Pizza store

declaring a factory method

Meet the Factory Method Pattern

Parallel class hierarchies

Factory Method Pattern defined

very dependent Pizza store

Looking at object dependencies

the dependency inversion Principle

Meanwhile, back at the Pizza store...

Families of ingredients...

Building our ingredient factories

Looking at the Abstract Factory

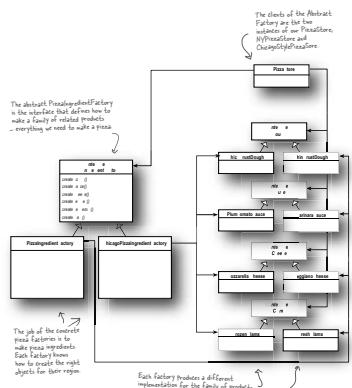
Behind the scenes

Abstract Factory Pattern defined

Factory Method and Abstract Factory compared

Tools for your design toolbox

Exercise solutions



the in eton attern

## ne of a ind ects

o

mi t e appy to now t at o all pattern , t e Sin leton i t e imple t in term  
o it cla dia ram; in act t e dia ram old ta in lecla ! B t don t et  
too com orta le; de pite it implicity rom a cla de i n per pecti e, we ll  
enco nter ite a ew mp and pot ole in it implementation. So c le  
p t i one not a imple a it eem ...

ne and only one object

he Little ingleton

issecting the classic ingleton Pattern

Confessions of a ingleton

he Chocolate Factory

ingleton Pattern defined

Hershey, PA  
Houston, we have a problem...

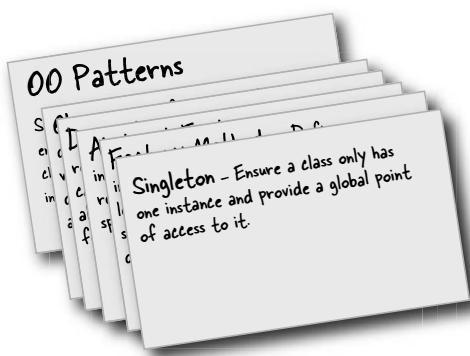
B the M

ealing with multithreading

ingleton

ools for your esign oolbo

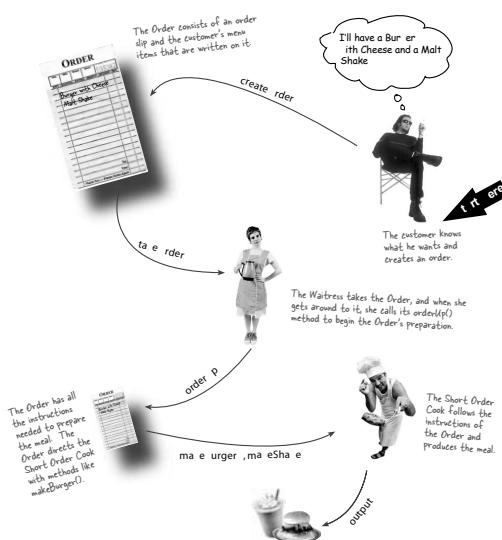
ercise olutions



# the command pattern

## encapsulating invocation

That right, by encapsulating invocation we can bury tallies piece of compilation out of the code in the compilation doesn't need to worry about how to do it; it's the invocation itself that does the work. We can also move wiggly martini with the encapsulated method invocation, like a separate them away or leave them to implement independently.



Home automation or Bust

the remote Control

taking a look at the vendor classes

Meanwhile, back at the inner...

Let's study the inner interaction

the objectville inner roles and responsibilities

From the inner to the Command Pattern

our first command object

the Command Pattern defined

the Command Pattern and the remote Control

implementing the remote Control

Putting the remote Control through its paces

time to write that documentation

sing state to implement and

very remote needs a Party Mode

sing a Macro Command

More uses of the Command Pattern queuing requests

More uses of the Command Pattern Logging requests

ools for your design toolbox

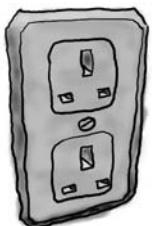
excise solutions

# the Adapter and the eing Adaptive

So and important?

Not when we are Design Pattern. Remember the Decorator Pattern? We're about to implement new responsibilities. Now we're going to wrap some object with a different purpose: to make their interface look like what they're not. Why would we do that? So we can adapt a design pattern one interface to a class that implements a different interface. That's not all, while we're at it we're going to look at another pattern that wraps objects to implement their interface.

European Wall Outlet



Standard AC Plug



apters all around us

object oriented adapters

the Adapter Pattern explained

Adapter Pattern defined

object and Class adapters

tonight's talk the object adapter and Class adapter

real World adapters

adapting an enumeration to an iterator

tonight's talk the Decorator Pattern and the Adapter Pattern

Home sweet Home heater

Lights, Camera, Facade

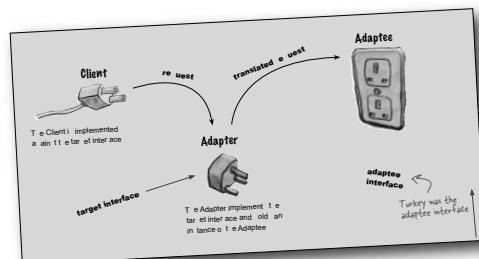
Constructing your Home heater Facade

Facade Pattern defined

the Principle of Least Knowledge

ools for your design toolbox

exercise solutions



the em ate etho attern

## ncapsulating Igorith s

We're oin to et down to encap latin ie es o algorit s o t at cla e can oo t em el e ri t into a comp tation anytime t ey want. We're en oin to learn a o ta de i n principle in pired y Hollywood.



Whipping up some coffee and tea classes

bstracting Coffee and ea

aking the design further

bstracting prepare ecipe()

What have we done?

Meet the emplate Method

Let's make some tea

What did the emplate Method get us?

emplate Method Pattern defined

Code up close

Hooked on emplate Method...

sing the hook

Coffee? ea? ah, let's run the est rive

he Hollywood Principle

he Hollywood Principle and the emplate Method

emplate Methods in the Wild

orting with emplate Method

We've got some ducks to sort

Comparing ducks and ducks

he making of the sorting duck machine

wingin' with Frames

pplets

onight's talk emplate Method and trategy

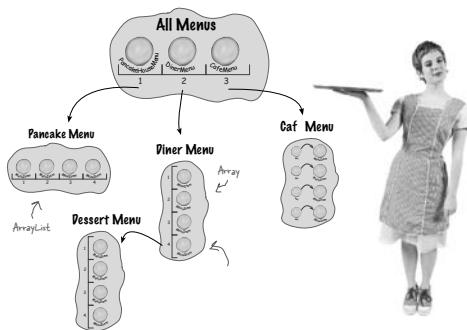
ools for your esign oolbo

ercise olutions

the iterator and composite patterns

## Iteration and managed collections

Put them in an Array, a Stack, a List, a Map, take your pick. Each has its own advantages and tradeoffs. Between your client wanting to iterate over your project, are you going to know how to implement? We certainly hope not! That's what *old fashioned*. Don't worry in this chapter you'll see how you can let your client *iterate* through your project without ever even knowing you're storing your project. It's really going to learn how to create some *smart collections* that can leap over some impressive data structures in a single bound. It's really going to learn about two additional concepts: *iteration* and *parallelism*.



bjectville dinner and Pancake House merge

Comparing Menu implementations

Can we encapsulate the iteration?

Meet the iterator Pattern

Adding an iterator to dinnerMenu

Looking at the design

Cleaning things up with java.util.iterator

What does this get us?

iterator Pattern defined

single responsibility

iterators and Collections

iterators and Collections in Java

just when we thought it was safe...

The Composite Pattern defined

designing Menus with Composite

implementing the Composite Menu

Flashback to iterator

full iterator

the magic of iterator and Composite together...

tools for your design toolbox

exercise solutions

# the state pattern

## the state of things

A year ago, the Strategy Pattern went on to create a wildly successful line around the world. State, however, took a different path. It helped learn to control their behavior by changing their internal state. He often overheard telling his client, "I repeat after me, I'm good enough, I'm smart enough, and do nothing..."

How do we implement state?

state Machines

first attempt at a state machine

ou knew it was coming... a change request

he messy state of things...

efining the state interfaces and classes

implementing our state Classes

eworking the Gumball Machine

he state Pattern defined

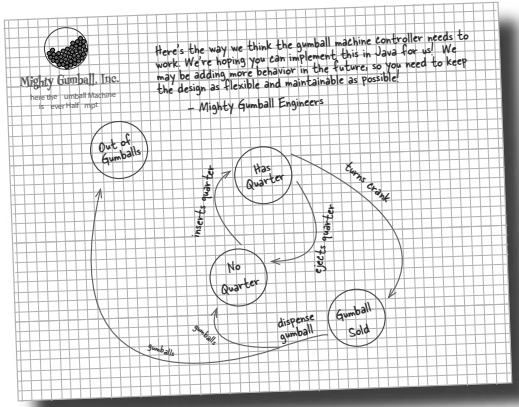
state versus strategy

state sanity check

We almost forgot

ools for your design toolbox

exercise solutions



# the ~~ro~~ pattern

## controlling access

o re t e ood cop and yo pro ide  
all yo r er ice in a nice and riendly manner, tyo dont want e eryo e  
a in yo or er ice , oyo a et e ad cop o trol a ess to yo . T at  
w at pro ie do: control and mana e acce . A yo re oin to ee t ere are  
*lots* o way in w ic pro ie tand in or t e o ect t ey pro y. Pro ie a e  
een nown to a l entire met od call o er t e Internet or t eir pro ied o ect ;  
t ey e al o een nown to patiently tand in t e place or ome pretty la y  
o ect .



Monitoring the gumball machines

he role of the remote pro y'

M detour

GumballMachine remote pro y

emote pro y behind the scenes

he Pro y Pattern defined

Get eady for virtual pro y

esigning the C cover virtual pro y

irtual pro y behind the scenes

sing the ava P 's pro y

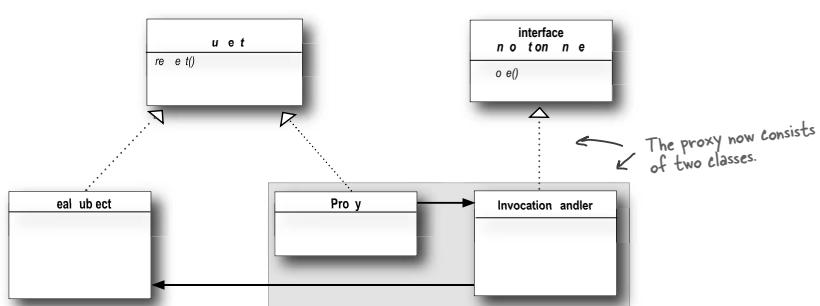
Five minute drama protecting subjects

Creating a dynamic pro y

he Pro y oo

ools for your esign oolbo

ercise olutions



## on our patterns

# Patterns of Patterns

o e already witne ed t e acrimonio Fire ide C at (and e t an l yo didnt a e to ee t e Pattern Deat Matc pa e t att e p li er orced to remo e rom t e oo o we co ld a oid a in to e a Parent Ad i ory warnin la el), o w o wo ld a e t o t pattern can act ally et alon well to et er? Belie e it or not, ome o t e mo t power 100 de i n e e eral pattern to et er. Get ready to ta e yo r pattern ill to t e ne tle el; it time or Compo nd Pattern .J t e care l yo r co-wor er mi t illyo i yo re tr c wit Pattern Fe er.

### Compound Patterns

uck reunion

dding an adapter

dding a decorator

dding a factory

dding a composite, and iterator

dding an observer

### Patterns summary

duck's eye view the class diagram

Model view Controller, the song

esign Patterns are your key to the M C

Looking at M C through patterns colored glasses

sing M C to control the beat...

he Model

he View

he Controller

ploring strategy

dapting the model

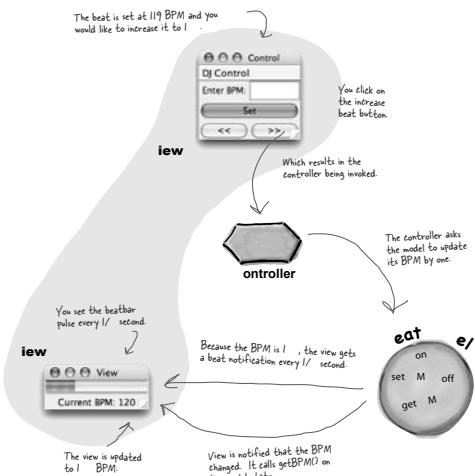
ow we're ready for a HeartController

M C and the Web

esign Patterns and Model

ools for your esignoolbo

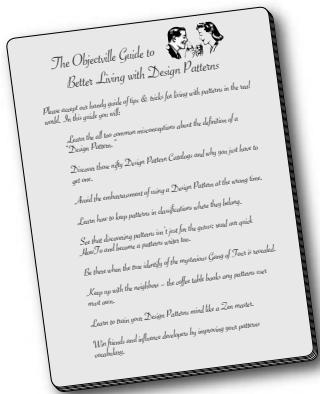
ercise olutions



etter i in ith atterns

## Patterns in the real world

B t, e ore yo o openin all t o e new door o opport nity we need to co er a ew detail t at yo ll enco nter o t in t e real world t in eta a little more comple o tt eret ant ey are ere in O ect ille. Come alon , we e ot a nice ide to elp yo t ro t e tran ition...



our objectville guide

esign Pattern defined

Looking more closely at the esign Pattern definition

May the force be with you

Pattern catalogs

How to create patterns

o you wanna be a esign Patterns writer?

rgani ing esign Patterns

hinking in patterns

our mind on patterns

on't forget the power of the shared vocabulary

op five ways to share your vocabulary

Cruisin' objectville with the Gang of Four

our journey has just begun...

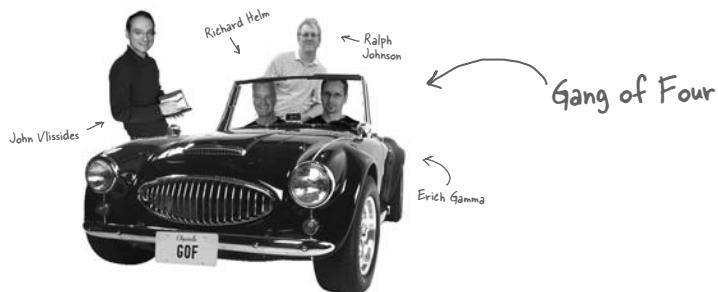
ther esign Pattern resources

he Patterns oo

nihilating evil with nti Patterns

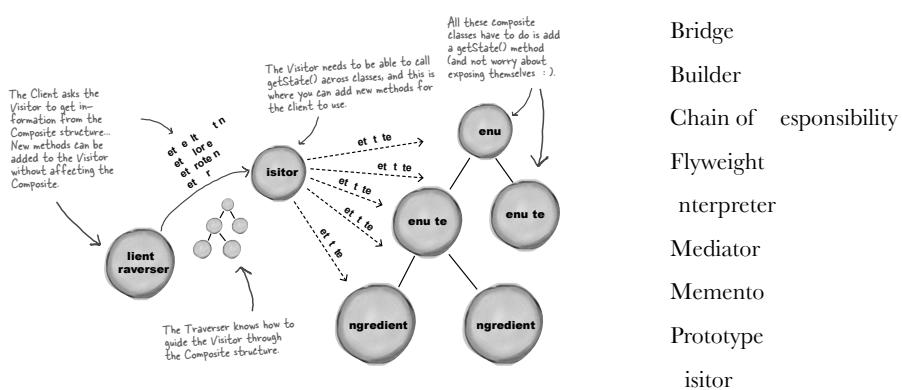
ools for your esign oolbo

Leaving objectville...



## ppendi effover Patterns

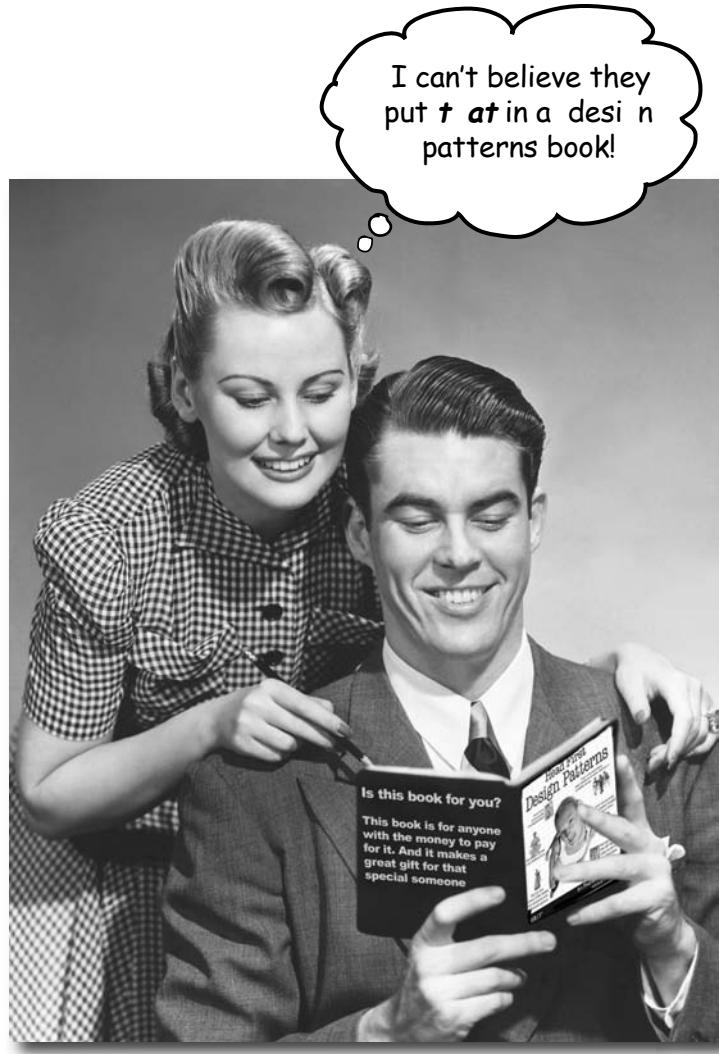
A lot a c an ed in  
t e la t 10 year . Since esig atter s le e ts o e sa le e t rie ted  
ot are rt came o t, de eloper a e applied t e e pattern t o and o time .  
T e pattern we mmari e int i appendi are ll- ed ed, card-carryin , o cial  
GoF pattern , t aren t alway ed a often a t e pattern we ee plored o  
ar. B tt e e pattern are awe ome in t eir own ri t, and i yo r it ation call or  
t em, yo o ld apply t em wit yo r read eld i . O r oal in t i appendi i  
to i e yo a i le el idea o w att e e pattern are all a o t.



i nde

ho to use this boo

## ntro



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

- o o no      o on t nee to e r

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

- Do yo want to le rn n er t n re e er n  
I de i n pattern , incl din t e OO de i n  
principle pon w ic de i n pattern are a ed?

- (3)** o o re er t l tn nner rt on er ton  
to r ll e le t re

this book is for you.

# Who should probably back away from this book?

If you can answer yes to any one of these

- re o o letel ne to

(o dont need to ead anced, and e en i yo  
do t now Ja a, t yo now C#, yo ll pro a ly  
nder tand at lea t 80% o t e code e ample . o  
al o ig t e o ay wit t a C++ ac ro nd.)

- Are yo a ic - tt OO de i ner/de eloper loo in  
or **re eren e oo**

- (3)** Are yo an arc itect loo in or **enter ri e** de i n  
pattern ?

- Are yo r to tr o et n erent?

Wo ld yo rat er a e a root canal t an mi  
tripe wit plaid? Do yo elie e t at a tec nical  
oo can t e erio i Ja a component are  
ant ropomorp i ed?



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.

our brain craves novelty. It's always searching, scanning, looking 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?

Motions crank up. Emotions surge.

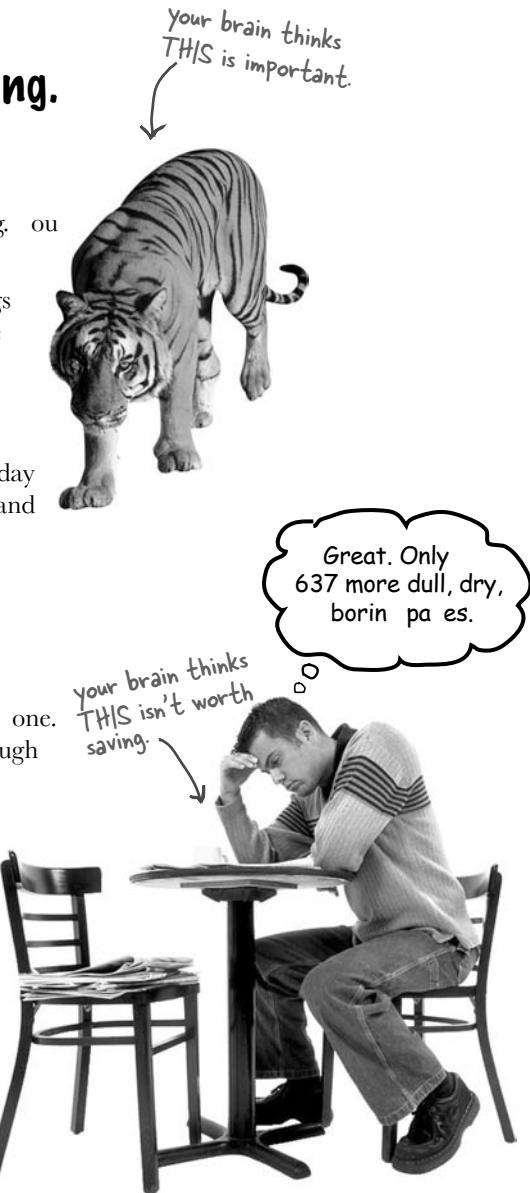
And that's how your brain knows...

### This ~~is~~ 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 ~~is~~ *sigh* 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.

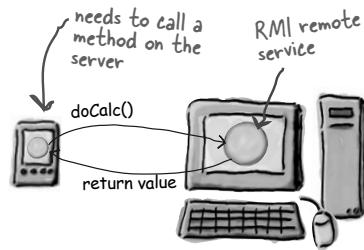
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really want you to keep this stuff around."



# et in o a ea first reader as a learner

## one of the Head First learning principles

**use it visual** Images are far more memorable than words alone, and make learning much more effective up to improvement in recall and transfer studies. It also makes things more understandable. **Put the words within or near the graphics** the 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.



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



abstract void roam();

No method body!  
End it with a semicolon.

**get the learner to think more deeply** In other words, unless you actively fire our neurons, nothing much happens in our head. The 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.

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



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

**punch their emotions** We now know that our ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you feel something. So, 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 rule! That comes when you solve a puzzle, learn something nobody else thinks is hard, or realize you know something that I'm more technical than thou job 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 eager to learn, but rarely tried 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 start it. To get the most from this book, or any book or learning experience, take responsibility for your brain. Your brain on its 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.

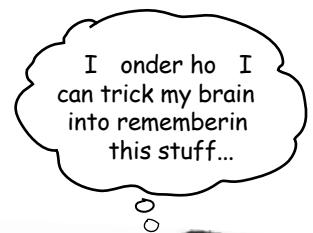
## **So how do you get your brain to think Design Patterns are as important as a tiger?**

Here's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you're 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 seem 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 *next to* 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 make 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.

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 *realize* 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 a thousand 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 *in* 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 three different ways and with different media types, and maybe these, 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 texture, 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 **happiness**, or, **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. Our brain does this even when you're reading it.

We included more than **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 doable, 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.

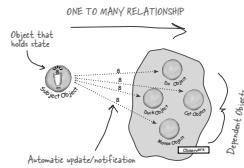
We include content for **both sides of your brain**, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning 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 think about something. Think about it – you can't get your body in shape just by lifting 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's why **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 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.

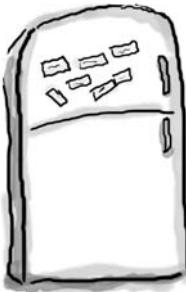


**e Patterns upu**



**Puzzles**





*cut this out and stick it  
on your refrigerator.*

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

o, we did our part. he rest is up to you. hese tips are a starting point; listen to your brain and figure out what works for you and what doesn't. try new things.

### 1 low down the more you understand, the less you have to memorize

on't just re . top and think. When the book asks you a uestions, don't just skip to the answer. imagine that someone really *is* asking the uestions. he 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. nd don't just at the e ercises. **se a pe c 1.** here's plenty of evidence that physical activity *while* learning can increase the learning.

### 3 Read the here are No Du uestions

hat means all of them. hey're not optional side bars **they're part of the core content!** on't skip them.

### 4 Revisit 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. our brain needs time on its own, to do more processing. f you put in something new during that processing time, some of what you just learned will be lost.

### 5 Drink water lots of it

our brain works best in a nice bath of uid. e hydration (which can happen before you ever feel thirsty) decreases cognitive function.

### 6 Talk out loud

peaking activates a different part of the brain. f you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to e plain it out loud to someone else. ou'll learn more uickly, 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. f you find yourself starting to skim the surface or forget what you just read, it's time for a break. nce 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

our brain needs to know that this m tters. 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

pply this to something new you're designing, or refactor an older project. ust do s met i g to get some e perience beyond the e ercises and activities in this book. ll you need is a pencil and a problem to solve... a problem that might benefit from one or more design patterns.

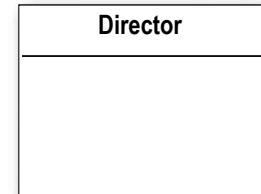
# Read Me

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

We use a simpler, modified faux-UML ↴

## e use si ple li e diagra s

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. To in other words, you won't have to worry about Design Patterns and UML at the same time. Our diagrams are UML *i.e.* while we try to be true to UML there are times we bend the rules a bit, usually for our own selfish artistic reasons.



## e 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), UML's own patterns, P 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.

## he activities are N 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. **on'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.

## e use the word co position in the general sense, which is more flexible than the strict use of co position

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.

## **he redundancy is intentional and important**

ne distinct difference in a Head First book is that we want you to *re* *g* 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 *re* *i* *g*, so you'll see some of the same concepts come up more than once.

## **he code examples are as lean as possible**

ur readers tell us that it's frustrating to wade through 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.

n 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 Java API, we mention it. We've also placed all the source code on the web so you can download it. You'll find it at <http://www.headfirstlabs.com/books/hfdp/>

lso, 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 *re* in packages.

## **he Brain Power exercises don't have answers**

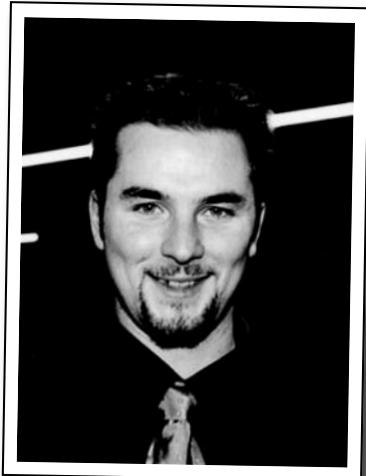
For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises you will find hints to point you in the right direction.

# Tech Reviewers

Jef Cumps



Valentin Crettaz



Barney Marispini



Ike Van Atta ↘



Fearless leader of  
the Hadoop Extreme  
Review Team.



Mark Spritzler ↗



Johannes de Jong ↗



Dirk Schreckmann

Jason Menard





## *In memory of Philippe Maquet*

1960-2004

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

*e i e er rget y*

Philippe Maquet

## Acknowledgments

### *At O'Reilly:*

ur biggest thanks to **e o des** at 'eilly 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, **m e ll**. Thanks to the clever Head First series mom **le art**, to rock and roll star **ll e ol ha se** for her inspired cover design and also to **ollee orma** for her hardcore copyedit. Finally, thanks to **e e dr c so** for championing this esign Patterns book, and building the team.

### *Our intrepid reviewers:*

We are extremely grateful for our technical review director **oha es de o**. You are our hero, ohannes. And we deeply appreciate the contributions of the co manager of the **a ara ch** review team, the late **h l ppe a et**. You have single handedly brightened the lives of thousands of developers, and the impact you've had on their (and our) lives is forever.

**e mps** is scarily good at finding problems in our draft chapters, and once again made a huge difference for the book. Thanks ef **ale t reta** (P 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 alentin (but lose the tie).

Two newcomers to the HF review team, Barney Marispini and ke an tta did a kick butt job on the book you guys gave us some *re* y crucial feedback. Thanks for joining the team.

We also got some excellent technical help from avaranch moderators gurus **ar pr t ler**, **aso e ard**, **r chrec ma**, **homas a l**, and **ar ar ta sae a**. And as always, thanks especially to the javaranch.com rail Boss, **a l heato**.

Thanks to the finalists of the avaranch Pick the Head First esign Patterns Cover contest. The winner, i Brewster, submitted the winning essay that persuaded us to pick the woman you see on our cover. Other finalists include Andrew SSE, Gian Franco Casula, Helen Crosbie, Phoebe, Helen Thomas, Ateesh Ommineni, and Jeff Fisher.

# Even more people

## **ro ric and lis a eth**

Writing a Head First book is a wild ride with two amazing tour guides **ath erra** and **ert ates**. With athy 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 year olds we know. So... what's next?

A big thank you to **e o des** and **e e dr c so**. 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.

Very special thanks to **r ch amma**, who went far beyond the call of duty in reviewing this book (he even took a draft with him on vacation). Rich, your interest in this book inspired us and your thorough technical review improved it immeasurably. Thanks as well to the entire **a o o r** for their support, interest, and for making a special appearance in Objectville. We are also indebted to **ard ham** and the patterns community who created the Portland Pattern Repository, an indispensable resource for us in writing this book.

It takes a village to write a technical book. **ll h** and **e r old** gave us expert advice on ingleton. **osh a ar acc** provided rockin' wing tips and advice. **oh rewer s** Why a duck? paper inspired him to duck (and we're glad he likes ducks too). **a r edma** inspired the Little ingleton example. **a el te ber** acted as our technical liaison and our emotional support network. And thanks to people's **ames empse** for allowing us to use his MC song.

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

## **ro athy and ert**

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

\*T e lar en m ero ac nowled ment i eca e were te tin t et eory  
t ate eryone mentioned in a oo ac nowled ment will y at lea t one copy,  
pro a ly more, w at wit relati e and e eryt in . I yo d lie to e int e  
ac nowled ment o or e t oo , and yo a e a lar e amily, write to .

## ntro to Design Patterns

# elco e to *Design Patterns*

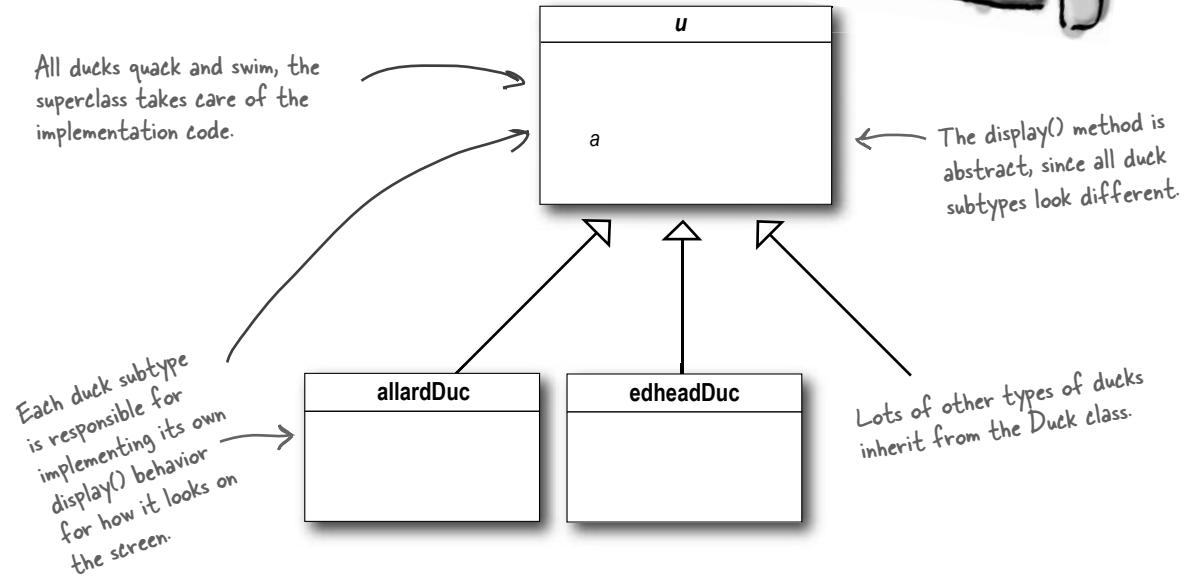
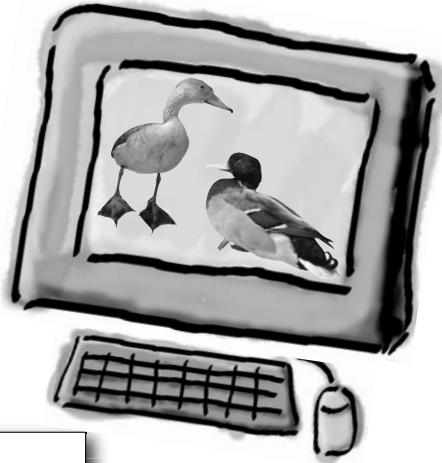


No that e're livin  
in Objectville, e've just ot  
to et into Desi n Patterns...  
everyone is doin them. Soon  
e'll be the hit of Jim and  
Betty's Wednesday ni ht  
patterns roup!

In t i c apter, yo ll learn  
w y (and ow) yo can e ploit t e wi dom and le on learned y ot er de eloper w o e  
een down t e ame de i n pro lem road and r i ed t e trip. Be ore we re done, we ll  
loo att e e and ene t o de i n pattern , loo at ome ey OO de i n principle , and  
wal t ro an e ample o ow one pattern wor . T e e tway to e pattern i to load  
yo r rai wit t em and t en re og i e la es in yo rde i n and e i tin application  
w ere yo can a lyt e . In tead o oder e, wit pattern yo et e erie e re e.

## 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 techniques and created one Duck superclass from which all other duck types inherit.

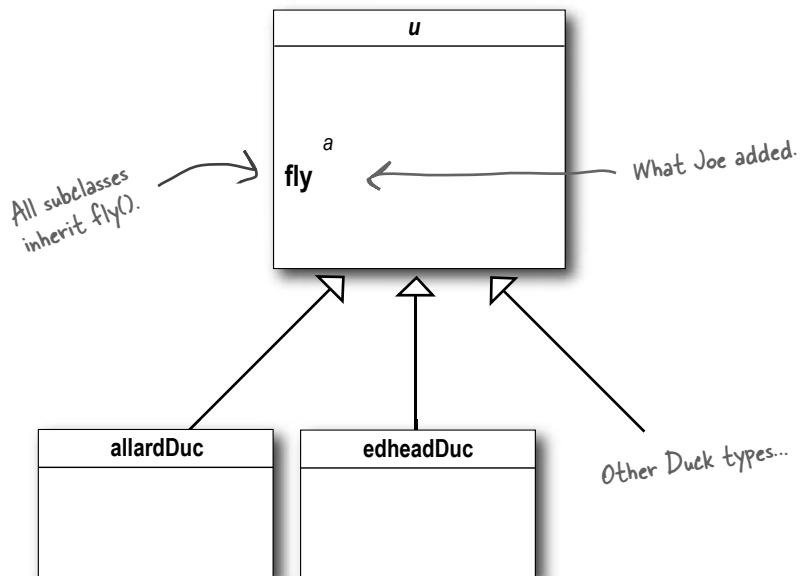
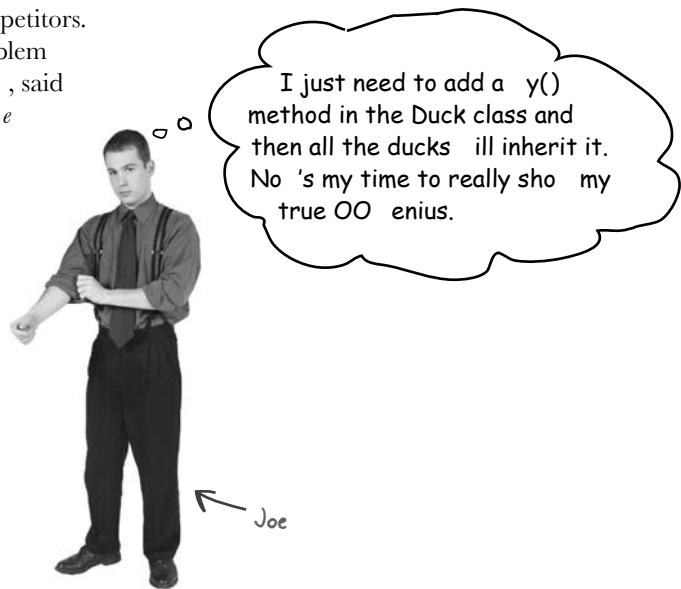


In the last year, the company has been under increasing pressure from competitors. After a week long off site brainstorming session over golf, the company executives think it's time for a big innovation. They need something really impressive to show at the upcoming shareholders meeting in Maui *next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

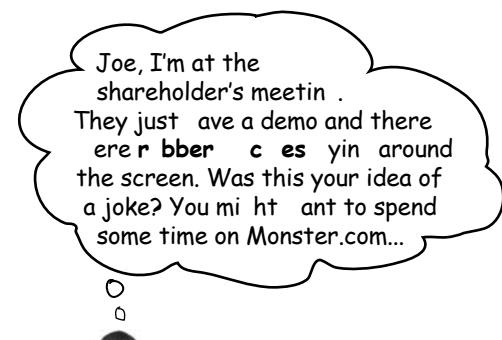
## But now we need the ducks to F Y

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 experienced programmer...



## But something went horribly wrong...

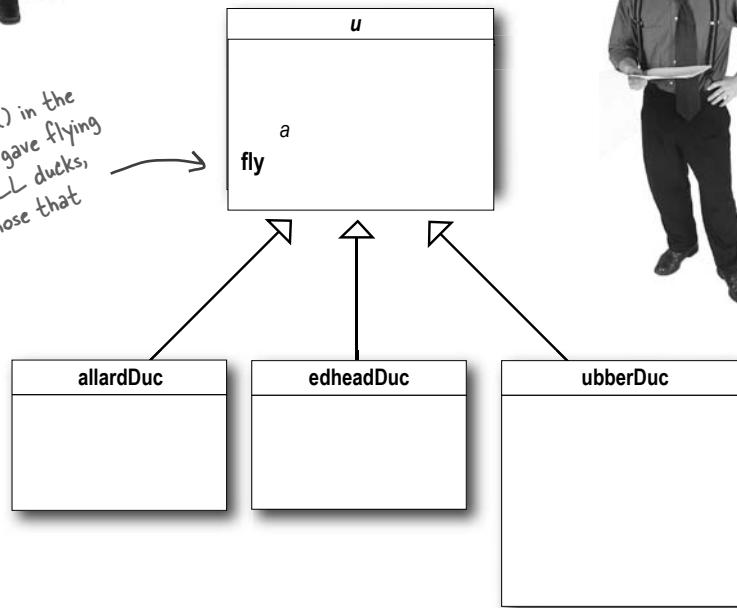


### **hat happened?**

oe failed to notice that not subclasses of Duck should fly. When oe added new behavior to the Duck superclass, he was also adding behavior that was t appropriate for some Duck subclasses. He now has ying inanimate objects in the im Duck program.

i e t e t e e se  
si e e t yi gr er s

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



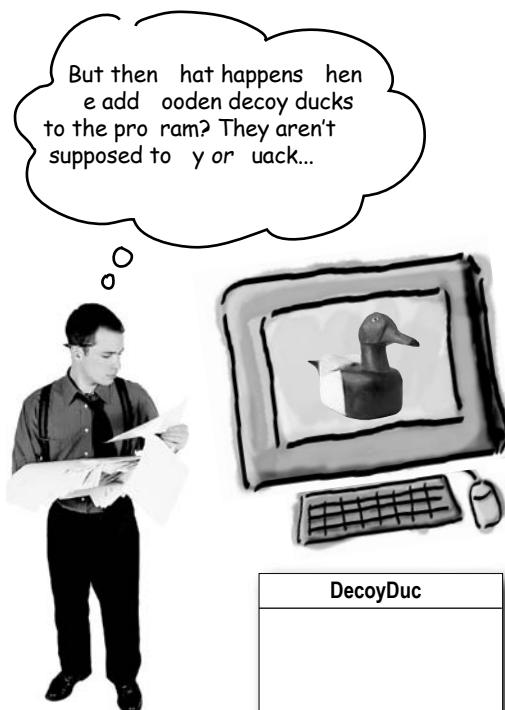
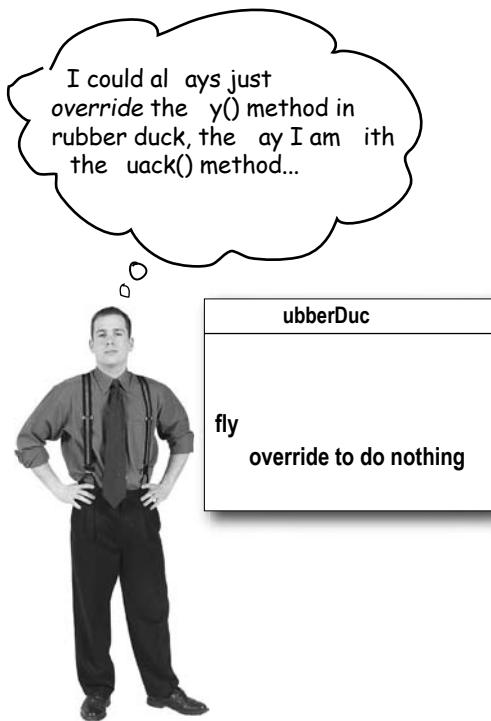
ha ter



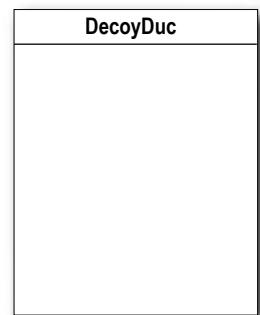
**What he thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance.**

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

## Joe thinks about inheritance...



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



### Sharpen your pencil

---

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

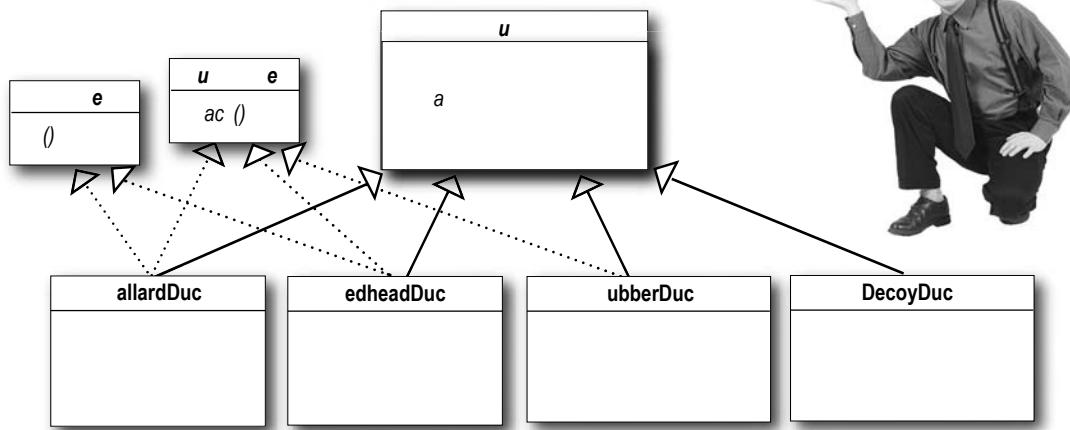
- |   |   |
|---|---|
| <input type="checkbox"/> A. Code is duplicated across subclasses.   | <input type="checkbox"/> D. Hard to gain knowledge of all duck behaviors.   |
| <input type="checkbox"/> B. Runtime behavior changes are difficult. | <input type="checkbox"/> E. Ducks can't fly and quack at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

## How about an interface?

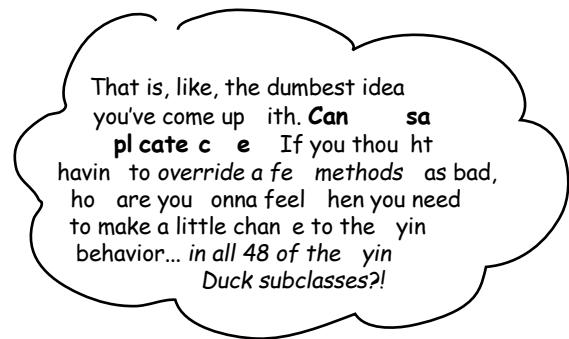
Joe realized that inheritance probably wasn't the answer, because he just got a memo that says that the executives now want to update the product every six months (in ways they haven't yet decided on). Joe knows the spec will keep changing and he'll be forced to look at and possibly override `quack()`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

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

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



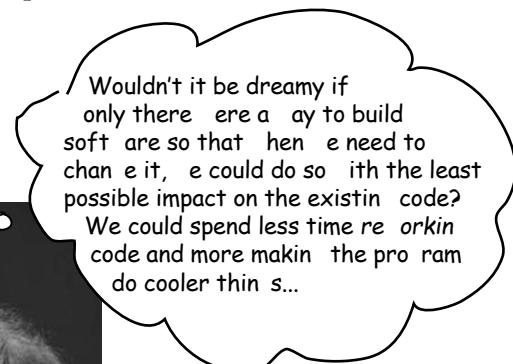
**What do you think about this design?**



## What would you do if you were oe?

We know that not of the subclasses should have ying or uacking behavior, so inheritance isn't the right answer. But while having the subclasses implement Flyable and or uackable solves rt of the problem (no inappropriately ying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a i ere t maintenance nightmare. And of course there might be more than one kind of ying behavior even among the ducks that y...

t this point you might be waiting for a esign Pattern to come riding in on a white horse and save the day. But what fun would that be? So, we're going to figure out a solution the old fashioned way *y ying st re esig ri i es*



# The one constant in software development

a what s the o e th o ca alwa s co t o so tware de elopme t

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?

5

(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 *i.e.*



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!

---

---

---

---

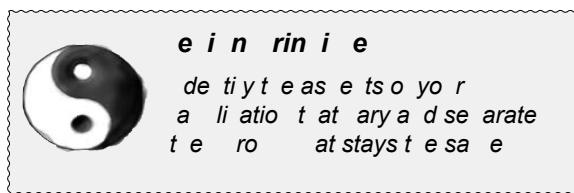
---

---

## eroing in on the problem...

o 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 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 bugs along the way.

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



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

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

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

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 other parts.

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

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

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

*u out hat arie*

## Separating what changes from what stays the same

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

Now, to separate the parts that change from those that stay the same, we are going to create two sets of classes (totally apart from `Duck`), one for `fly` and one for `quack`. Each set of classes will hold all the implementations of their respective behavior. For instance, we might have a `Flyer` class that implements `fly()`, a `Quacker` class that implements `quack()`, and a `Screamer` class that implements `scream()`.

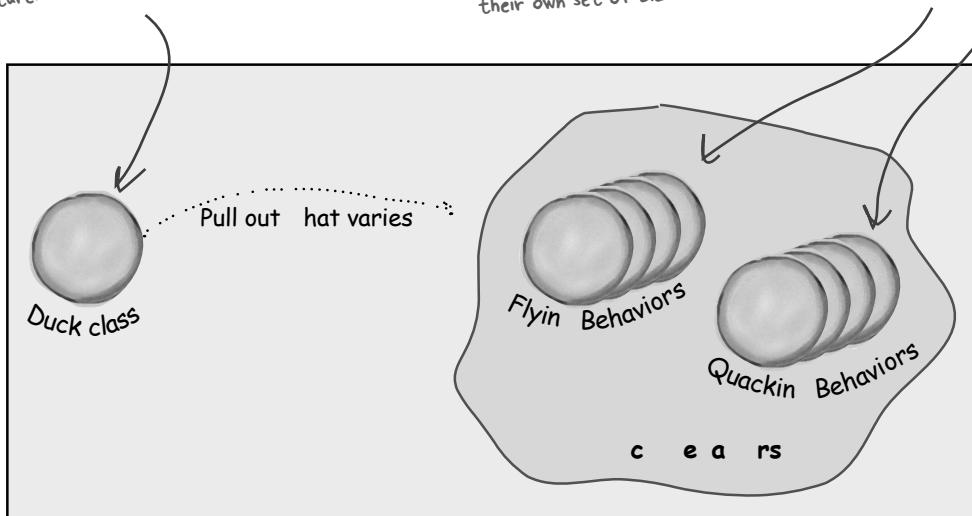
**e now that `fly` and `quack` are the parts of the Duck class that vary across duck's**

**o 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**

The `Duck` class is still the superclass of all ducks, but we are pulling out the `fly` and `quack` behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



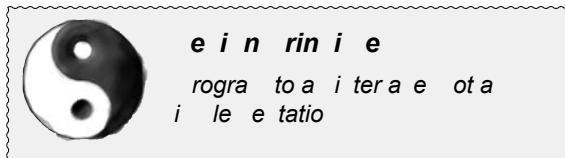
*ha ter*

# Designing the Duck Behaviors

## ~~o how are we o to des the set o classes that mpleme t the a d ac beha ors~~

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

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



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.

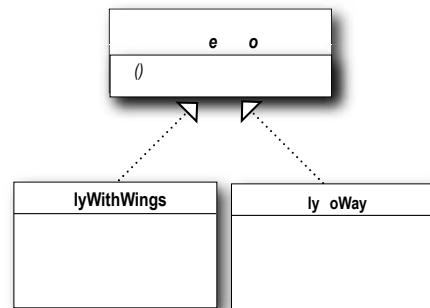
This time it won't be the `Duck` classes that will implement the `flying` and `quacking` interfaces. Instead, we'll make a set of classes whose entire reason for living is to represent a behavior (for example, `swimming`), and it's the `Behavior` class, rather than the `Duck` class, that will implement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

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

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

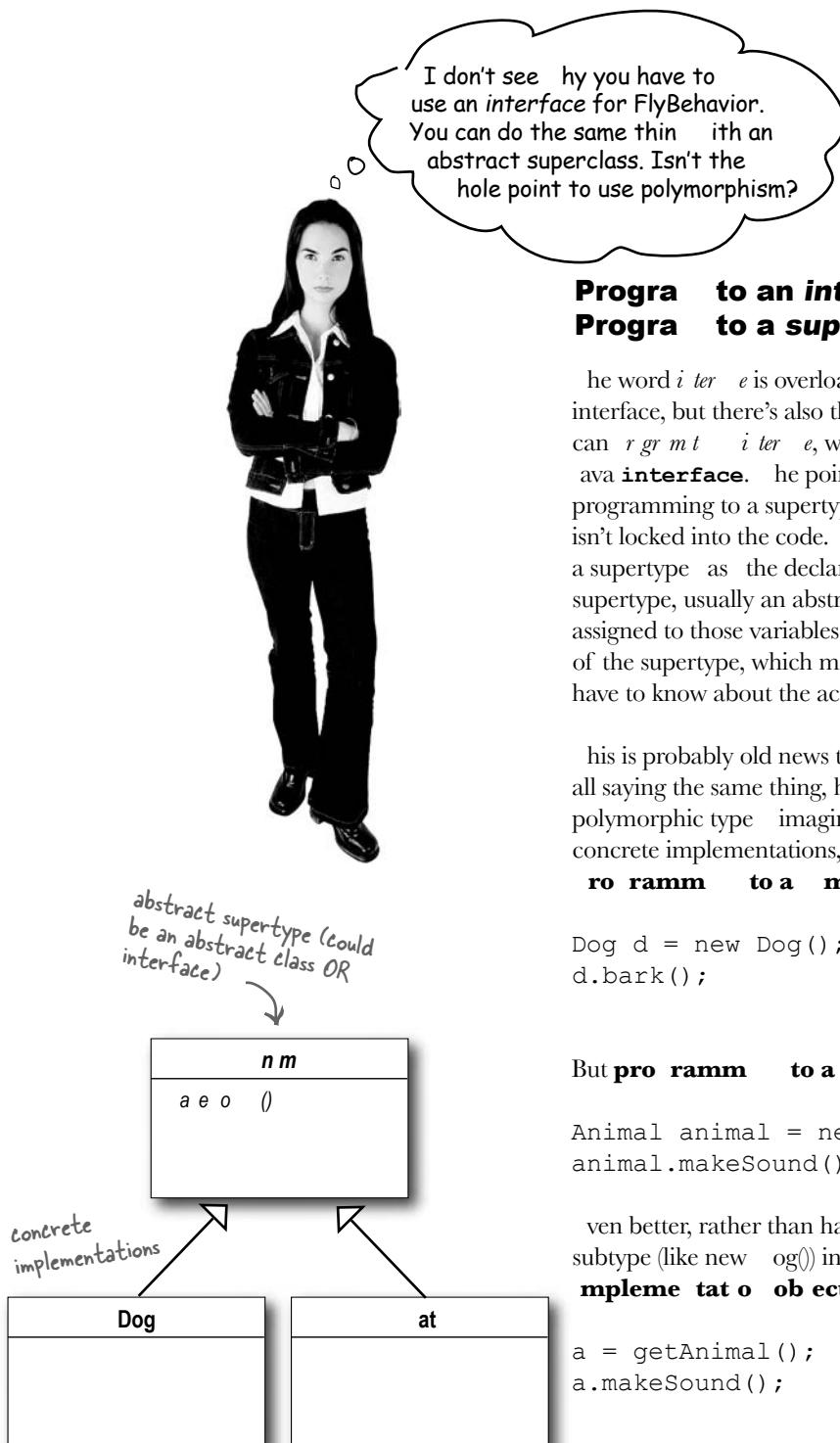
**From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.**

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



*you are here ▶*

**ro ra to an interface**



## **Program to an interface really means Program to a supertype**

The word *interface* is overloaded here. Here's the *real* of interface, but there's also the Java construct **interface**. You can *implement* *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*.

**Program to a *interface* would be**

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

But **Program to a *supertype*/pertype** would be

```
Animal animal = new Dog(); We know it's a Dog, but  
animal.makeSound(); we can now use the animal  
reference polymorphically.
```

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

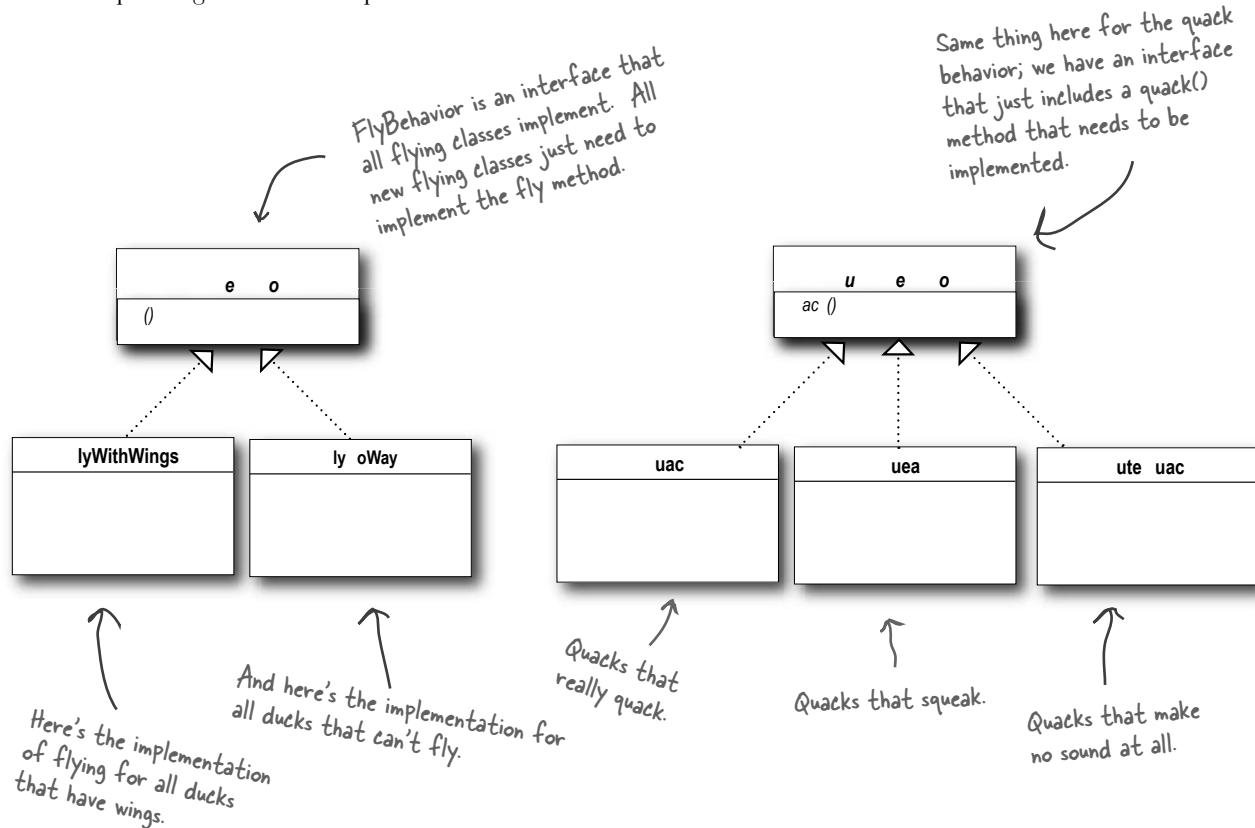
```
a = getAnimal();  
a.makeSound();
```

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

*harder*

# Implementing the Duck Behaviors

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



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

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

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

*ha ter*

## there are no Dumb Questions

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

**A:**

**Q:** It feels a little weird to have a class that just a behavior Aren't classes supposed to represent the *functionality* of the objects?

**A:**

*t*

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

**A:**

## Sharpen your pencil

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

② Can you think of a situation that might want to extend Quack with a short animation?

) Create a Fly object-powered class that implements the FlyBehavior interface.  
( ) Use example, a duck call (a device that makes duck sounds).

Answers

*ha ter*

# Integrating the Duck Behavior

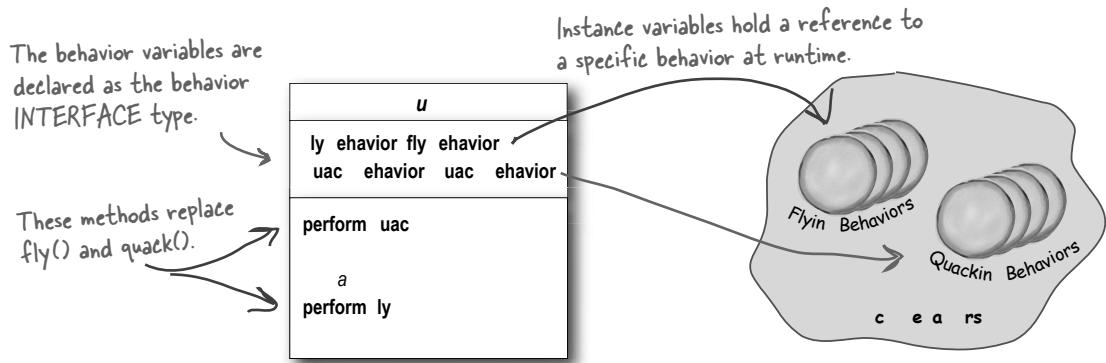
**he *key* is that a *Duck* will now delegate its *ying* and *uac ing* *ehavior*, instead of using *uac ing* and *ying* *ethods de ned in the *Duck* class or su class***

## Here's how

- First we'll add two *sta ce arables* to the *Duck* class called *y e i r* and *e i r*, that are declared as the interface type (not a concrete class implementation type). Each *Duck* object will set these variables polymorphically to reference the *s e i* behavior type it would like at runtime (*FlyWithWings*, *Quack*, etc.).

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

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



- Now we implement *perorm ac* :

```
public class Duck {
    QuackBehavior quackBehavior; ← Each Duck has a reference to something that
    // more implements the QuackBehavior interface.

    public void performQuack() {
        quackBehavior.quack(); ← 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 *uack*, a *Duck* just allows the object that is referenced by *quackBehavior* to *uack* for it.

In this part of the code we don't care what kind of object it is, ***all we care a out is that it nows how to uac !***

*you are here ▶*

## More Integration...

- 3 Okay, time to worry about **how the ~~eha or a d~~ ac eha or sta ce ar ables are set**. Let's take a look at the Mallard Duck class

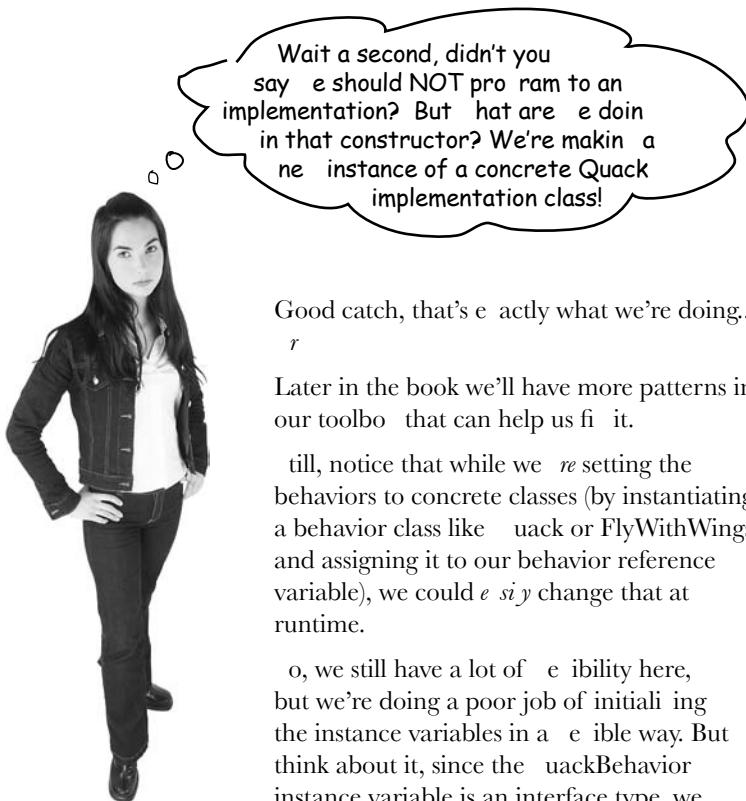
```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

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

o Mallard Duck's quack is a real live duck **ac**, not a **s ea** and not a **m te ac**. o what happens here? When a Mallard Duck is instantiated, its constructor initializes the Mallard Duck's inherited quackBehavior instance variable to a new instance of type Quack (a QuackBehavior concrete implementation class).

nd the same is true for the duck's wing behavior the Mallard Duck'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...

r

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

Still, notice that while we're 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.)

*you are here ▶*

# Testing the Duck code

- ① type and compile the Duck class below Duck.java , and the allardDuck class from two pages ago allardDuck.java**

```
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly(); ← Declare two reference variables
    } ← for the behavior interface types.
    public void performQuack() {
        quackBehavior.quack(); ← All duck subclasses (in the same
    } ← package) inherit these.

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

- ② type and compile the Fly behavior interface FlyBehavior.java and the two behavior implementation classes FlyWithWings.java and FlyNoWay.java**

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

The interface that all flying behavior classes implement.

---

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

Flying behavior implementation for ducks that DO fly...

---

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

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

# Testing the Duck code continued...

## ③ type and compile the QuackBehavior interface

**QuackBehavior.java and the three Behavior implementation classes**

```
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.

## Run the code

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

## Setting behavior dynamically

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

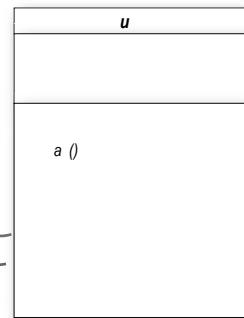
### ① add two new methods to the Duck class

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

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

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

*editor note    ratuitou    un    x*



### ② create a new Duck type ModelDuck

```
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 create a new Fly behavior type FlyRocketPowered

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

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



ha ter

- Change the test class `MiniDuckSimulator`, add the `ModelDuck`, and see the `ModelDuck` rocket enabled

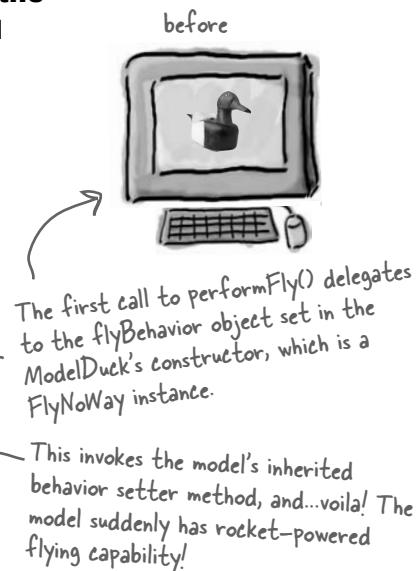
```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
```

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

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

- Run it

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



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

*you are here ▶*

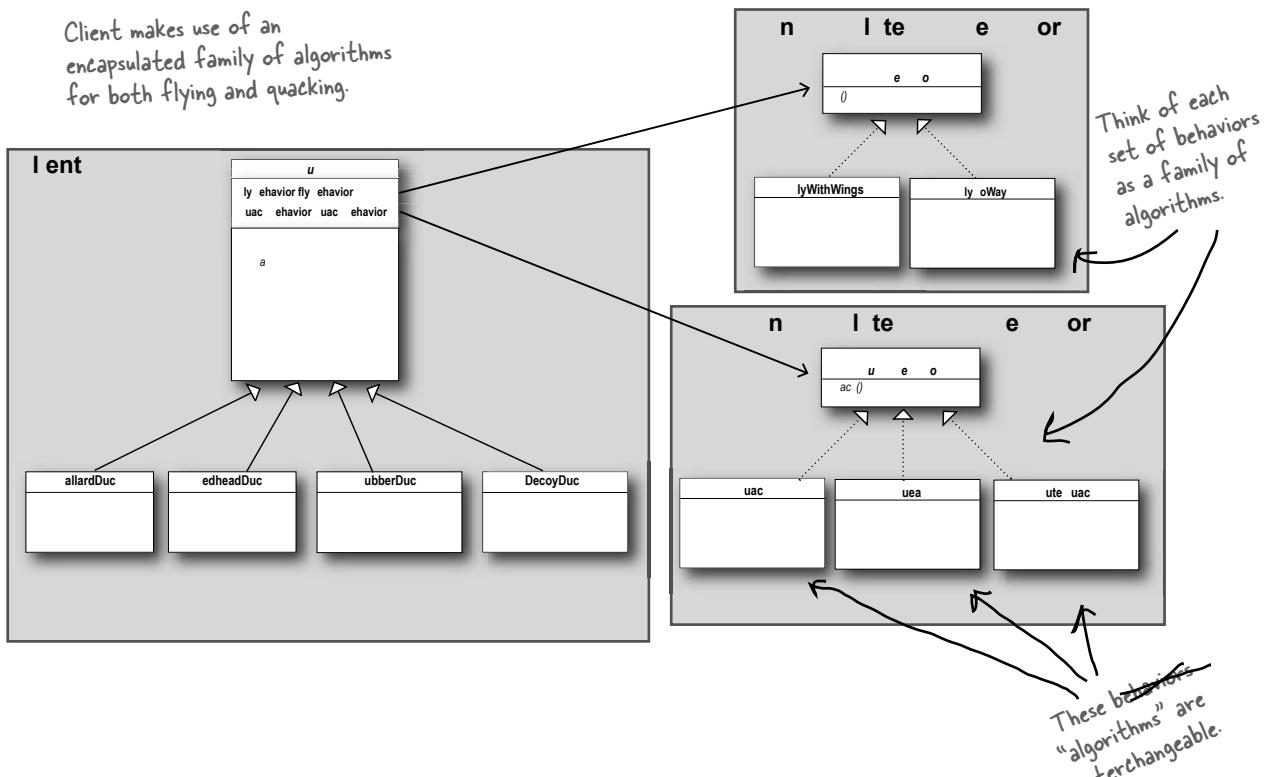
# The Big Picture on encapsulated behaviors

**ay, now that we've done the deep dive on the duc si ulator design, it's ti e to co e ac up for air and ta e a loo at the ig picture**

Below is the entire reworked class structure. We have everything you'd expect ducks attending duck, y behaviors implementing FlyBehavior and uack 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 *items*, we'll start thinking of them as a *family* of *algorithms*. Think about it in the im duck 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 to by different states.

Pay careful attention to the *relationships* between the classes. In fact, grab your pen and write the appropriate relationship (,, H and MPL M ) 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 **co-position**. Instead of *implementing* their behavior, the ducks get their behavior by being *associated* with the right behavior object.

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



e i n r i n i e

Fa or o ositio o eri erita e

So 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.



Ad c call i a de ice t at nter e to mimic t e  
call ( ac )o d c . How wo ld yo implement yo r  
own d c call t at doe ot in erit rom t e D c cla ?



a ter and tudent

a ter rass o er tell e atyo a e leardedo te e t rie ted ays

tudent Master a elear edt at t e ro ise o t eo e torie ted ay isre se

a ter rass o er o ti e

tudent Master tro g i erita e all good t i gs ay ere sed a d so e ill o e to drasti ally t de elo e tti eli e es istly t a oo i t e oods

a ter rass o er is ore ti es e to ode e ore or a ter de elo e tis o lete

tudent e a s er is a ter Master e al ays s e d ore ti e ai tai i g a d a gi g so t are t a i itial de elo e t

a ter o rass o er s o lde ort go i to re se a o e ai tai ta ility a de te si ility

tudent Master elie e t att ere is tr t i t is

a ter a see t at yo still a e to lear o ld li e or yo to go a d editate o i erita e rt er s yo e see i erita e as its ro le s a dt ere are ot er ays o a ie i gre se

## Speaking of Design Patterns...



Congratulations on  
our first pattern

o t le or r t e n ttern te  
ttern t r t o e te  
tr te ttern to re or te  
n tot ttern te l tor re or n  
n e t o e e e t oo onterne t  
ne tr to e  
  
o t t e e e o t et elon ro to l t  
ere t e or l e n t o n o t ttern

**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.



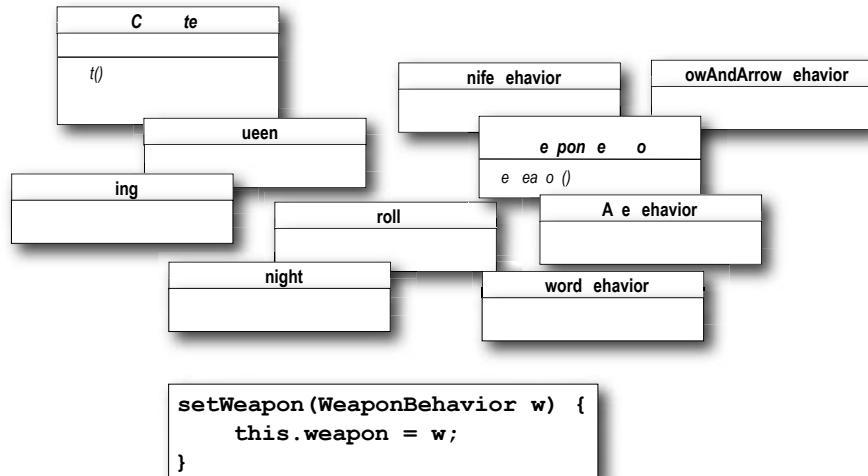
# esign Pu le

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. Our job is to sort it all out...

( Answers are at the end of the chapter.)

## o r t a s :

- ➊ Range the classes.
- ➋ Identify one abstract class, one interface and eight classes.
- ➌ Draw arrows between classes.
  - a. Draw this kind of arrow for inheritance ( e tends ). →
  - b. Draw this kind of arrow for interface ( implements ). .....→
  - c. Draw this kind of arrow for H . →
- ➍ Put the method setWeapon() into the right class.



*you are here ▶*

## Overheard at the local diner...

**lice**

I need a Cream cheese  
ith jelly on hite bread, a  
chocolate soda ith vanilla ice cream, a  
rilled cheese sand ich ith bacon, a tuna  
sh salad on toast, a banana split ith  
ice cream & sliced bananas and a coffee  
ith a cream and t o su ars, ... oh,  
and put a hambur er on the rill!

**Flo**

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



What's the difference between these two orders? ot a thing hey're both  
the same order, e cept lice is using twice the number of words and trying the  
patience of a grumpy short order cook.

What's Flo got that lice doesn't? **sharedocab lar** with the short order  
cook. ot only is it easier to communicate with the cook, but it gives the cook less  
to remember because he's got all the diner patterns in his head.

esign Patterns give you a shared vocabulary with other developers. nce 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. t also elevates your  
thinking about architectures by letting you **th at the pattern le el**, not the  
nitty gritty e t level.

## Overheard in the next cubicle...



**RA POWER**

Can you think of other areas that are beyond OO design and development? (Hint: law office automation, carpenter, ormetec, air traffic control) What abilities are communicated along with this?

Can you think of aspects of OO design that are communicated along with patterns? What abilities are communicated along with the name Strategy Pattern?



*you are here ▶*

# The power of a shared pattern vocabulary

**hen you co unicate using patterns you  
are doing ore than ust sharing N**

**hared patterocab lar es are**

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.

**atter s allow o to sa more w th less.** When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

**al at the patter le el allows o to sta  
the des lo er.** 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.

**haredocab lar es ca t rbo char e o r  
de elopme t team.** A team well versed in design patterns can move more quickly with less room for misunderstanding.

**haredocab lar es e co ra e more or  
de elopers to et p 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.

"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.

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

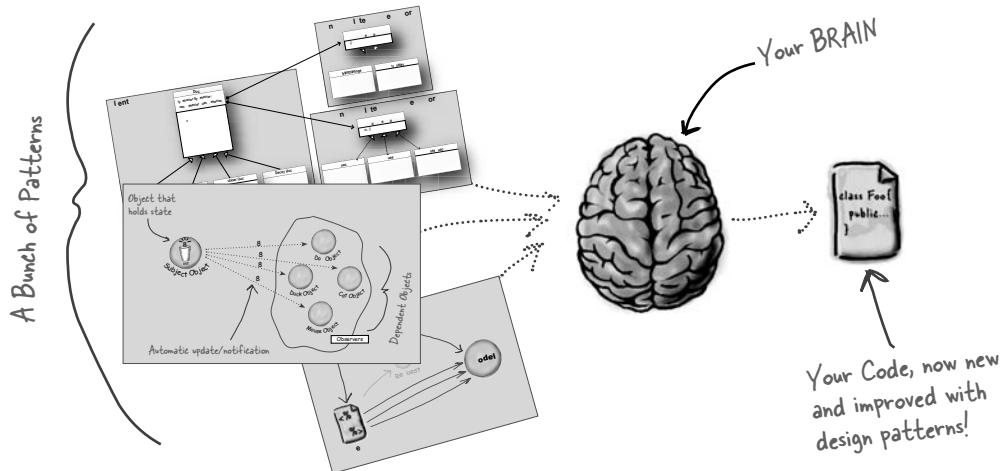
As your team begins to share design ideas and experience in terms of patterns, you will build a community of patterns users.

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, 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 reusable. 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 inreusable mess of spaghetti code.



*there are no  
Dumb Questions*

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

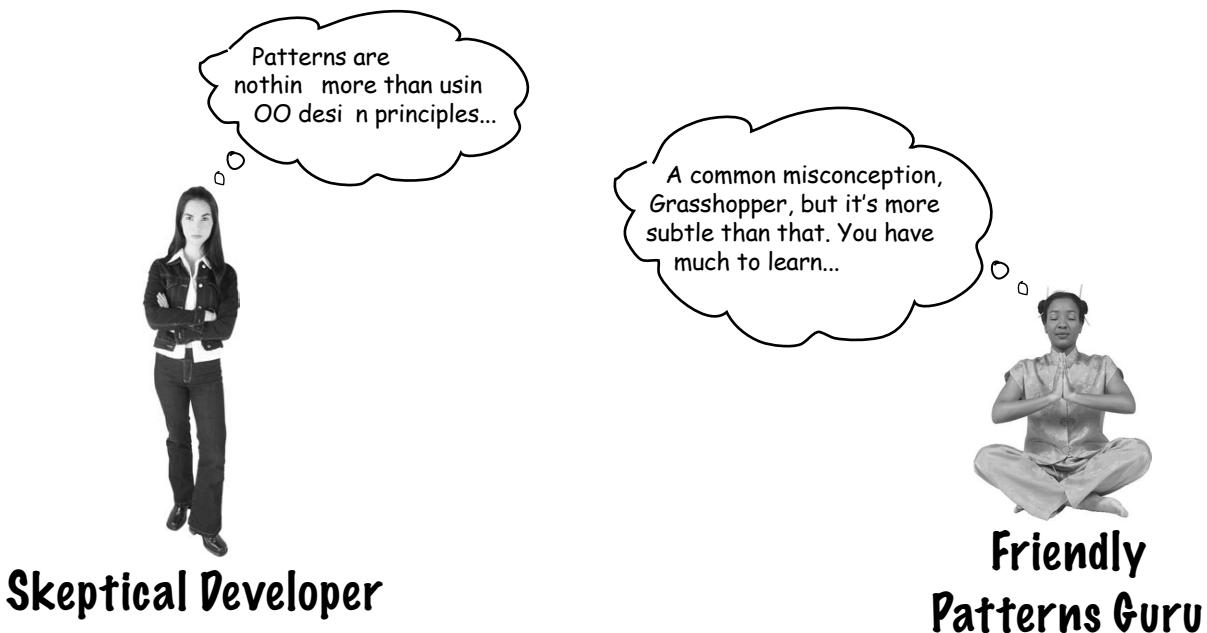
**A:**

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

**A:**

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

**A:**



**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.

**uru** 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?

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

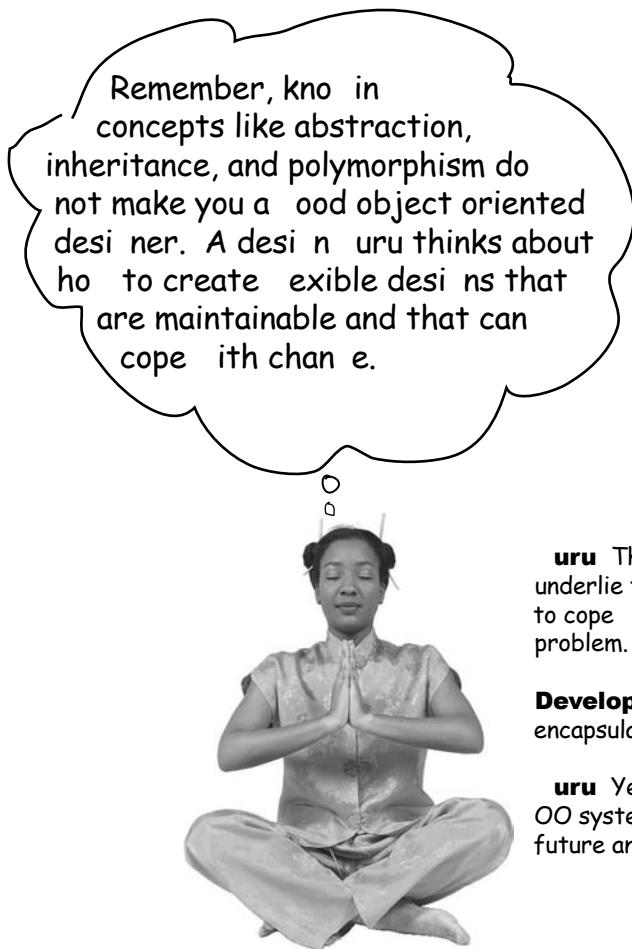
**Developer** I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected...

**uru** ...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?

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

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

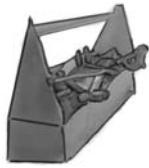


Remember, knowin concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A designer needs to think about how to create flexible designs that are maintainable and that can cope with change.

**uru** 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...

**uru** 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

**ou've nearly ade it through the rst chapter ou've already put a few tools in your tool o let's a e a list of the efore we ove on to hapter**



# 00 Basics

Abstraction  
Encapsulation  
Polymorphism  
Inheritance

# 00 Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.

# OO Patterns

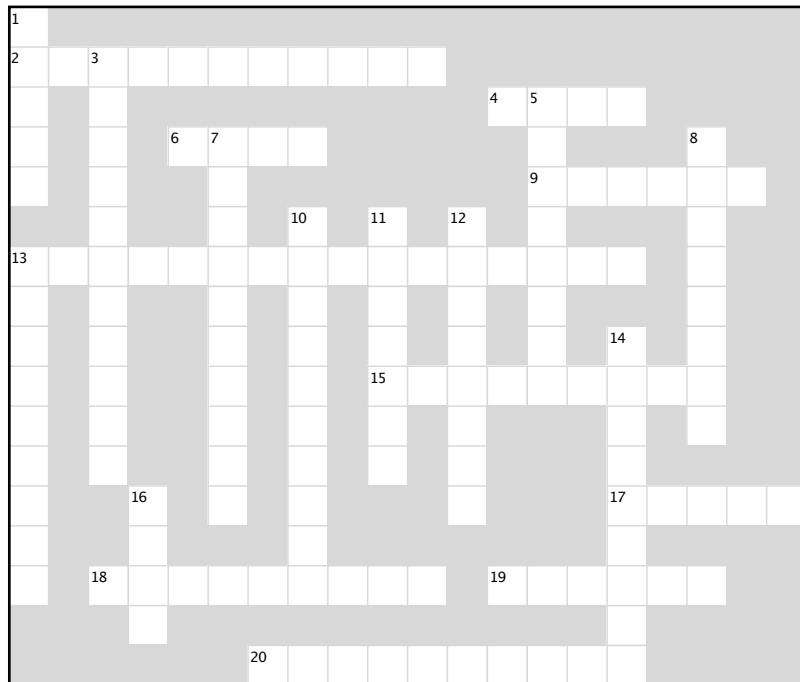
**Strategy** - defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

One down, many to go!

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

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

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

**Across**

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

**Down**

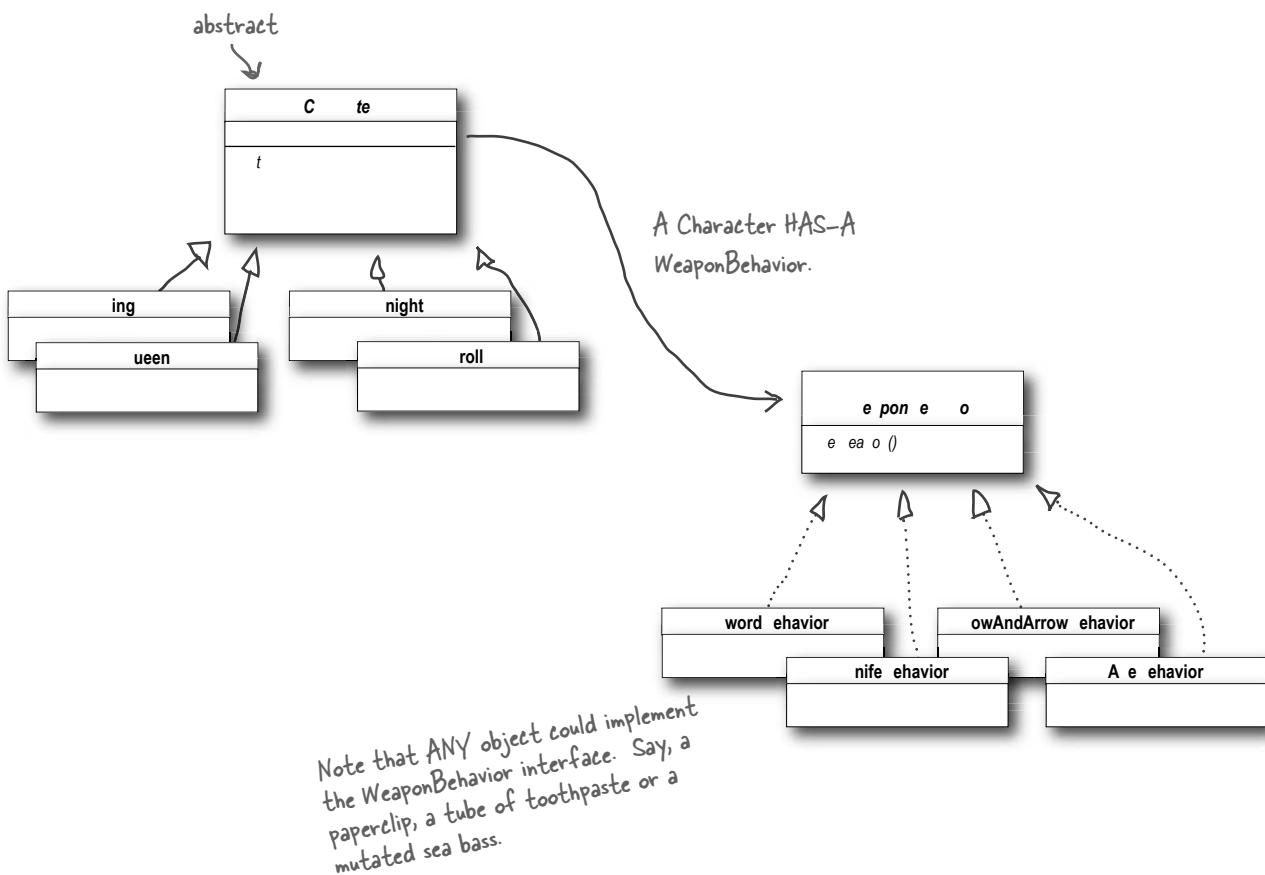
- 1. Patterns \_\_\_\_\_ in many applications
- 3. Favor over inheritance
- 5. Dan was thrilled with this pattern
- 7. Most patterns follow from OO \_\_\_\_\_
- 8. Not your own
- 10. High level libraries
- 11. Joe's favorite drink
- 12. Pattern that fixed the simulator
- 13. Duck that can't quack
- 14.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



# esign Pu le Solution

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

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

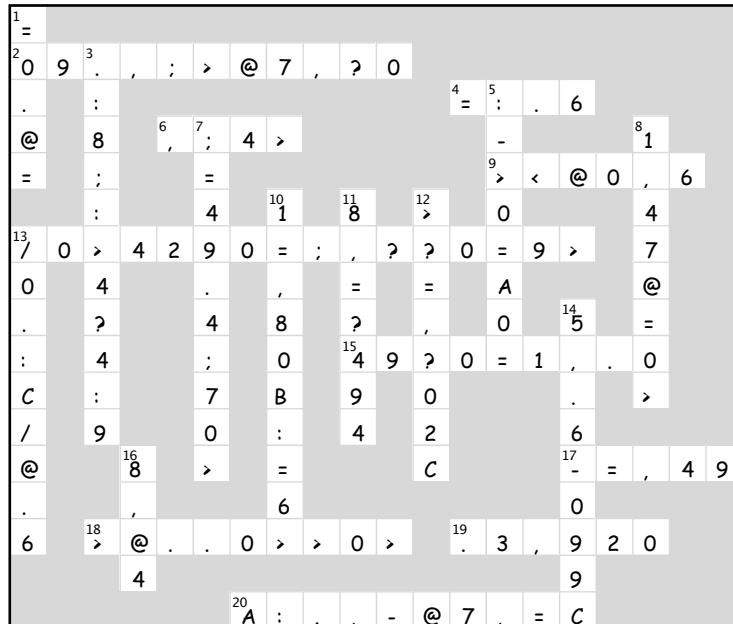


# Solutions

## Sharpen your pencil

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

- A. Code is duplicated across subclasses.
- B. Untime 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

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

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

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

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

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



the bser er attern

# keeping your ects in the now



Hey Jerry, I'm  
notifyin everyone that the  
Patterns Group meetin moved to  
Saturday ni ht. We're oin to be  
talkin about the Observer Pattern.  
That pattern is the best! It's the  
BEST, Jerry!

We e ot a  
pattern t at eep yo ro ect in t e now w en omet in t ey mi t care a o t appen .  
O ect can e en decide at r ntine w et er t ey want to e ept in ormed. T e O er er  
Pattern i one o t e mo t ea ily ed pattern in t e JDK, and it incredi ly e l. Be ore  
we re done, we ll al o loo at one to many relation ip and loo e co plin (yea , t at ri t,  
we aid co plin ). Wit O er er, yo ll e t e lie o t e Pattern Party.

## ongratulations

our tea has ust won the contract to uild  
eather a a, nc 's ne t generation,  
nternet ased eather onitoring tation



Weather-O-rama, Inc.  
Main Street  
ornado lle,

### statement of Work

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

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

Further, this is an expandable weather station. Weather-O-Rama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

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

We look forward to seeing your design and alpha application.

Sincerely,

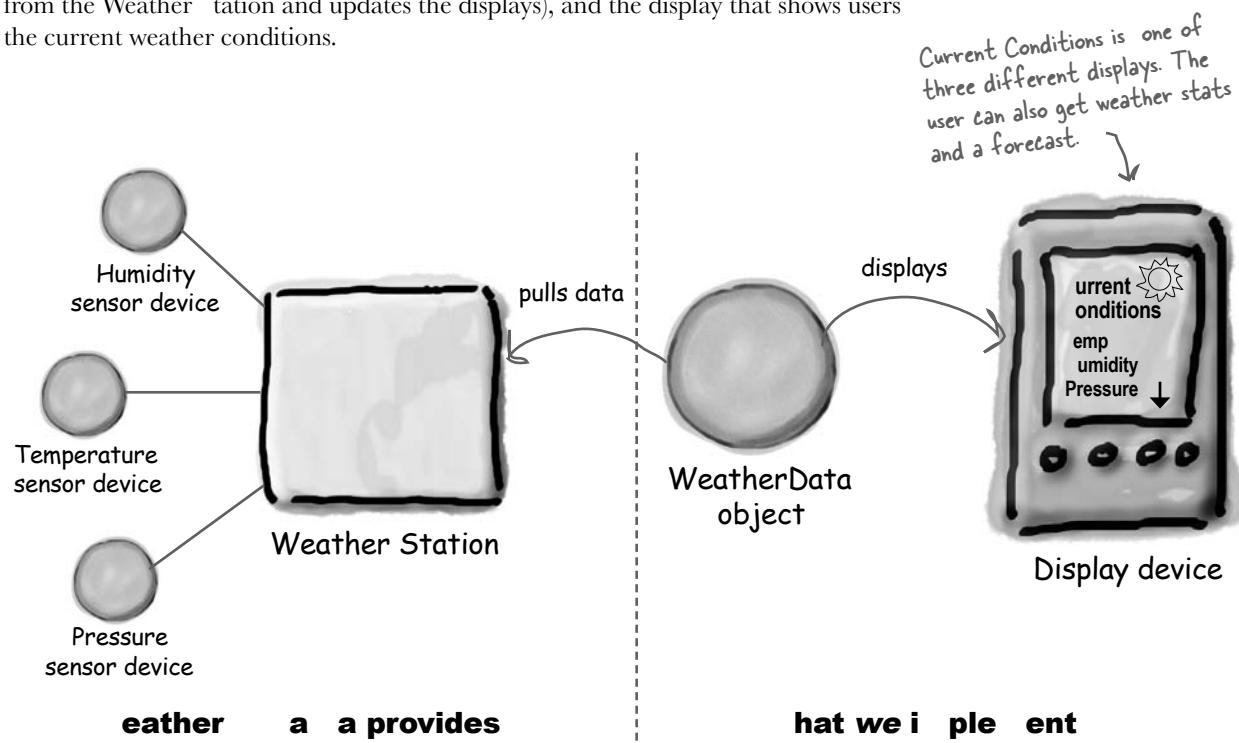
*Johnny Hurricane*

Johnny Hurricane, CEO

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

# The Weather Monitoring application overview

he 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.



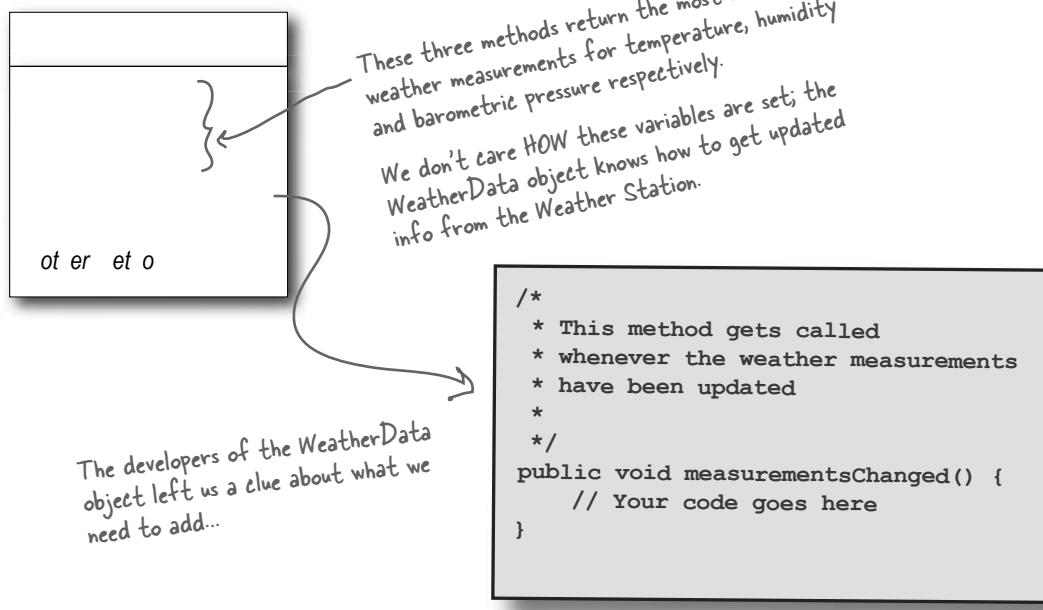
he WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements Current Conditions (shows temperature, humidity, and pressure), Weather statistics, and a simple forecast.

**ur o , 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**

weather data a

## Unpacking the WeatherData class

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



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



Display device

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

later

# What do we know so far?



he spec from Weatherama 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 a method for tree measurement called :temperature, humidity and atmospheric pressure.

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

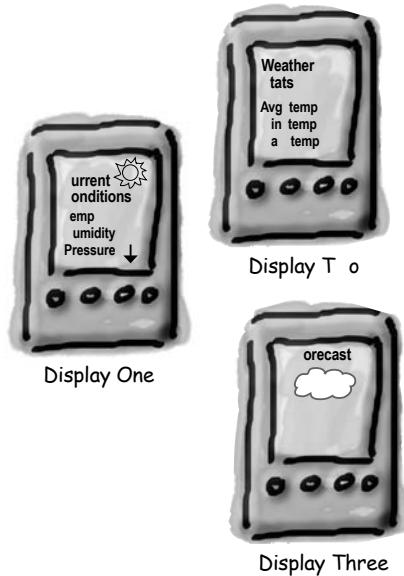
We need to implement three display elements at the weather data: a current conditions display, a statistics display and a forecast display. The displays must be updated each time WeatherData adds a new measurement.

`getTemperature()`

`getHumidity()`

`getPressure()`

`measurementsChanged()`



Future displays

The system needs to be flexible so it can create new custom display elements and so it can add or remove many display elements as they want to the application. Currently, we now about only the initial three display types (current condition, statistic 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 Data 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 Weather Data class.

# What's wrong with our implementation?

hin ac to all those hapter concepts and principles

```
public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();
    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Area of change, we need to encapsulate this.

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

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

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



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

*eet the o er er attern*

## Meet the Observer Pattern

**ou now how newspaper or aga ine  
su scriptions wor**

- ❶ newspaper publisher goes into business and begins publishing newspapers.
- ❷ 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.
- ❸ you unsubscribe when you don't want papers anymore, and they stop being delivered.
- While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

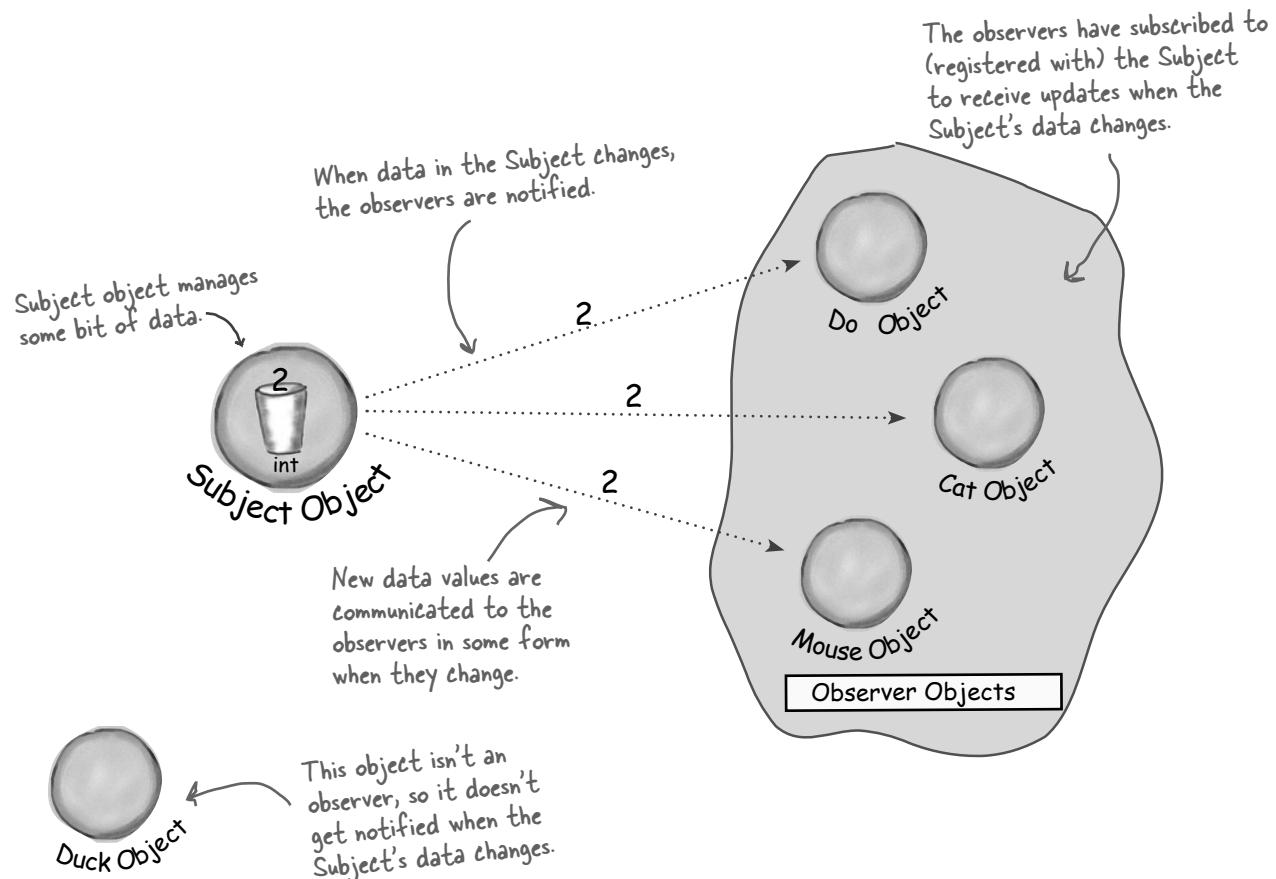


*ha ter*

# 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.

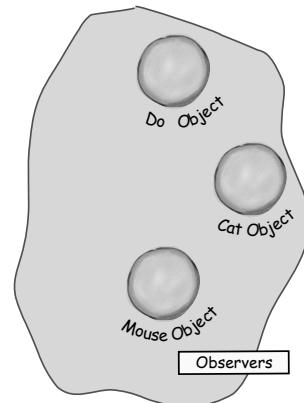
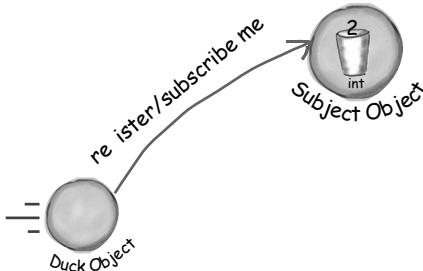


you are here ▶

## A day in the life of the Observer Pattern

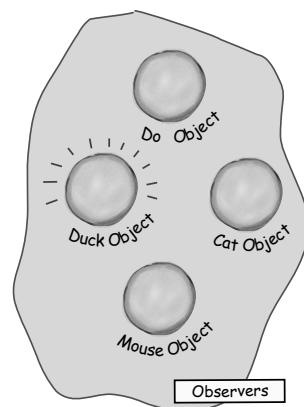
c object c es al n  
an tells t e object t at  
t ants t bec ean  
bser er

Duck really ants in on the  
action; those ints Subject is  
sendin out henever its state  
chan es look pretty interestin ...



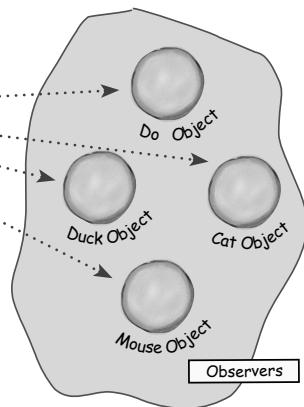
e c object s n an  
cal bser er

Duck is psyched... he's on the  
list and is aitin ith reat  
anticipation for the next  
noti cation so he can et an int.



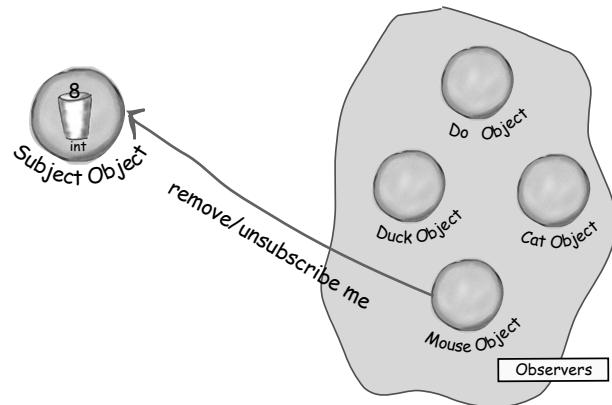
e object ets a ne  
ata al e!

No Duck and all the rest of the  
observers et a noti cation that  
the Subject has chan ed.



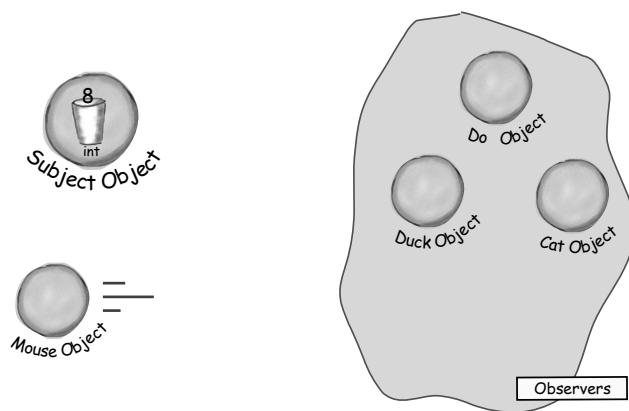
e se bject as s t be  
re e as an bser er

The Mouse object has been  
ettin ints for a es and is tired  
of it, so it decides it's time to  
stop bein an observer.



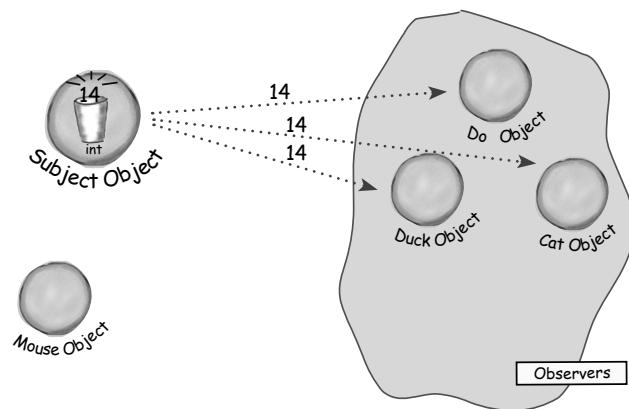
se s tta ere!

The Subject ackno led es the  
Mouse's re uest and removes it  
from the set of observers.



e bject as an t er  
ne nt

All the observers et another  
noti cation, except for the  
Mouse ho is no lon er included.  
Don't tell anyone, but the Mouse  
secretly misses those ints...  
maybe it'll ask to be an observer  
ain some day.



you are here ▶



## Five minute drama: a subject for observation

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



1

Software Developer



3

Software Developer



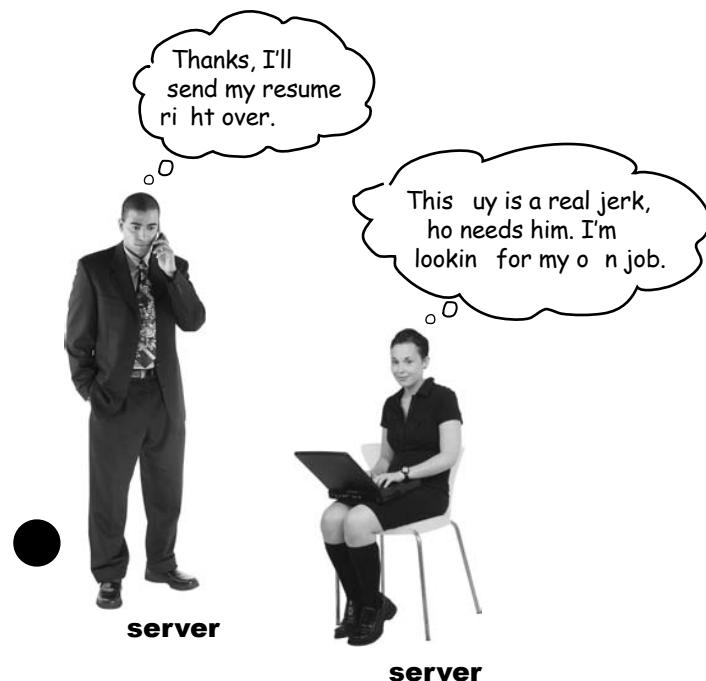
2

Headhunter contact



contact

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



*the o er er attern de ned*

## Two weeks later...



Jill's lovin' life, and no lon'er an observer.  
She's also enjoyin' the nice fat si nin  
bonus that she got because the company  
didn't have to pay a headhunter.



But what has become of our dear Ron? We hear  
he's beatin' the headhunter at his own game.  
He's not only still an observer, he's got his own  
call list now, and he is notifyin' his own observers.  
Ron's a subject and an observer all in one.

*ha ter*

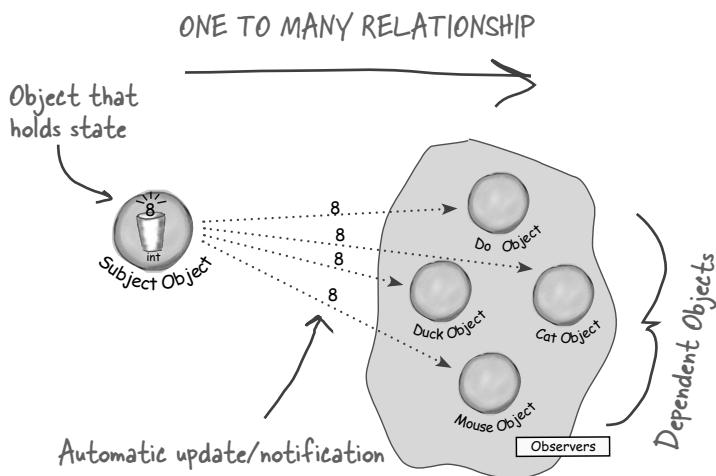
# The Observer Pattern defined

hen ou're tr ing to picture the bserver attern, a newspaper subscription service with its publisher and subscribers is a good wa to visuali e the pattern.

In the real world however, ou'll t picall see the bserver attern defined li e this:

**he bser er atter** defines a one to many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

et's relate this definition to how we've been tal ing about the pattern:



he sub ect and observers define the one to man relationship. he observers are dependent on the sub ect such that when the sub ect's state changes, the observers get notified. Depending on the st le of notification, the observer ma also be updated with new values.

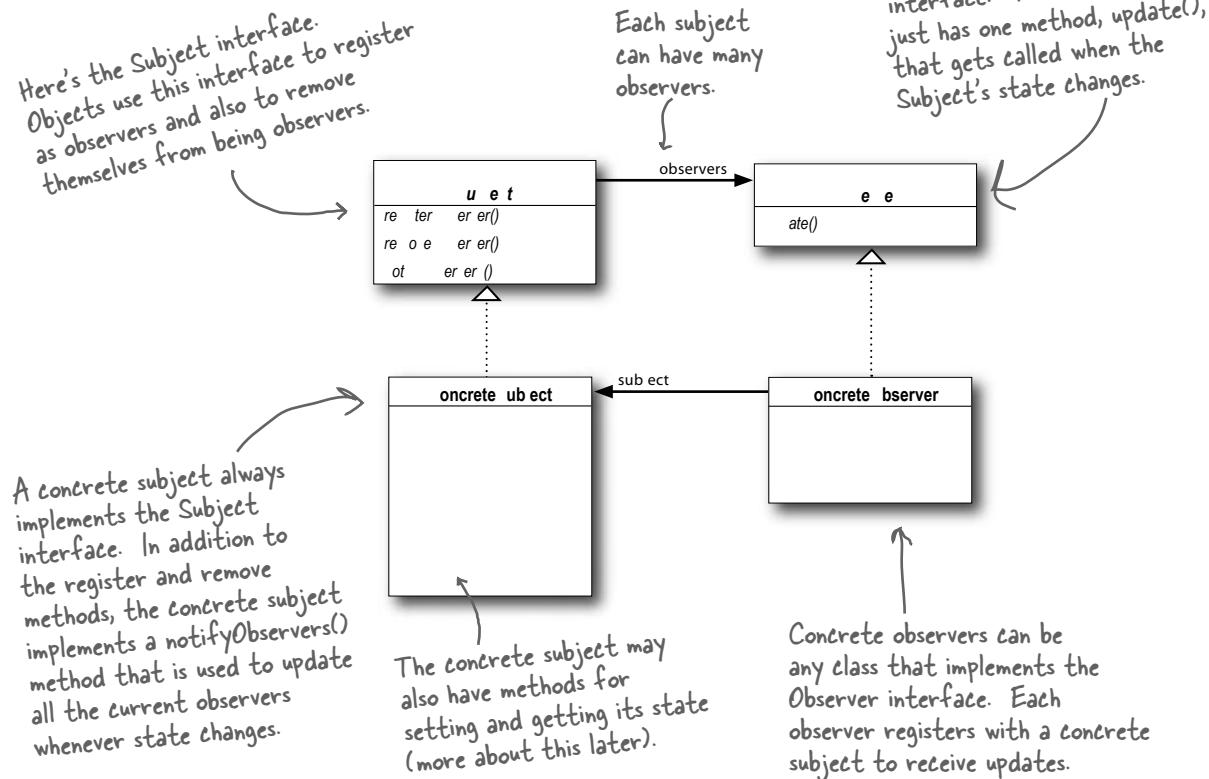
s ou'll discover, there are a few different wa s to implement the bserver attern but most revolve around a class design that includes Sub ect and bserver interfaces.

et's take a loo ...

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

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

# The Observer Pattern defined: the class diagram



*there are no*  
**Dumb Questions**

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

**A:** With the observer pattern, the Subject is the object that contains the state and controls it. So, there is a 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 Subject to the MANY observers.

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

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

# The power of loose Coupling

**hen two o ects are loosely coupled, they can interact, ut have very little nowledge of each other**

**he server Pattern provides an o ect design where su ects and o servers are loosely coupled**

**hy?**

**he o l th the s b ect ows abo t a obser er s that t mpleme ts a certa ter ace** (the bserver interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

**e ca add ew obser ers at a t me.** Because the only thing the subject depends on is a list of objects that implement the bserver 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.

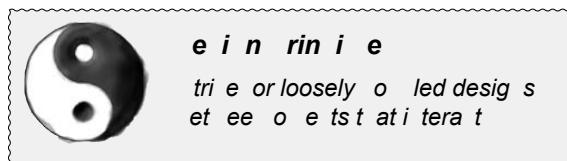
**e e er eed to mod the s b ect to add ew t pes o obser ers.** Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the bserver interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the bserver interface.

**e ca re ses b ects or obser ers depe de tl o 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.

**ha es to e ther the s b ect or a obser er w ll ot a ect 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.



How many different kinds of change can you identify here?



**oosely coupled designs allow us to uild e i le syste s that can handle change ecause they ini i e the interdependency etween o ects**



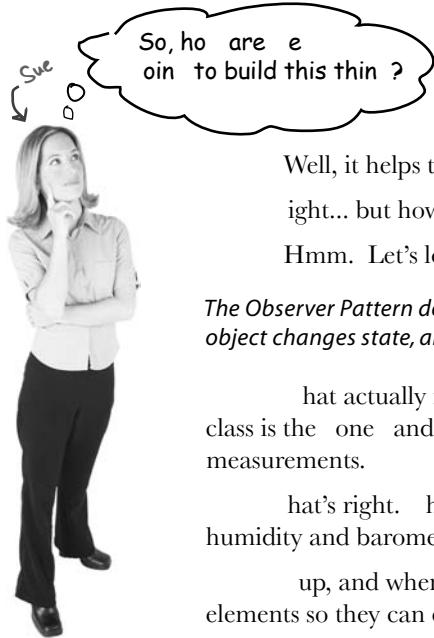
## Sharpen your pencil

Be ore mo in on, try etc in o tt e cla e yo ll need to implement t e Weat er Station, incl din t e Weat erData cla and it di play element . Ma e re yo r dia ram ow ow all t e piece t to et er and al o ow anot er de eloper mi t implement er own di play element.

I yo need a little elp, read t e ne t pa e; yo r teammate are already tal in a o t ow to de i nt e Weat er Station.

## Cubicle conversation

ac to the eather tation pro ect, your tea ates have  
already started thin ing through the pro le



Well, it helps to know we're using the bserver Pattern.

ight... but how do we apply it?

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.*

hat actually makes some sense when you think about it. ur Weather ata class is the one and our many is the various display elements that use the weather measurements.

hat's right. he Weather ata class certainly has state... that's the temperature, humidity and barometric pressure, and those definitely change.

up, 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.

Cool, now think see how the bserver Pattern can be applied to our Weather tation problem.

here are still a few things to consider that 'm not sure understand yet.

Like what?

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

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

es... and once the Weather tation knows about a display element, then it can just call a method to tell it about the measurements.

We gotta remember that every display element can be different... so 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 Weather ata object will know how to send them the measurements.

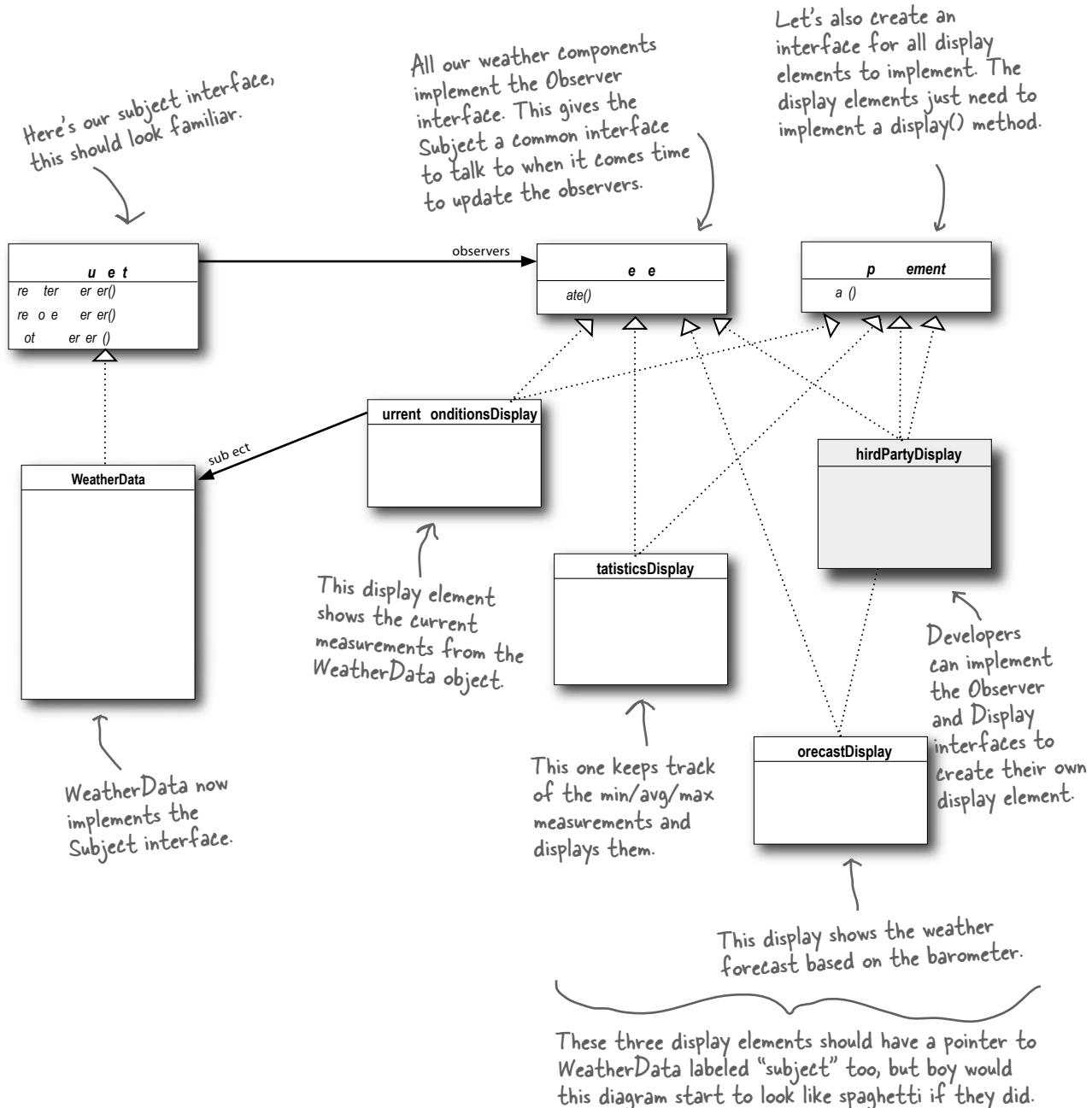
see what you mean. o every display will have, say, an update() method that Weather ata will call.

nd update() is defined in a common interface that all the elements implement

*you are here ▶*

# 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();
}

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

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

*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.*

*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.*

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



Mary and Sue often pass information directly to the observer when a measurement is made. Do you think this is wise? Hint: it is an area of the application that might change often? It did change, wouldn't it 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 updated state to the observer?

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

# le e t t e bject te ace eat e ata

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

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

Here we implement the Subject Interface.

```

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

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

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

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

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

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

    // other WeatherData methods here
}

```

The code is annotated with several arrows pointing to specific parts of the implementation:

- An arrow points from the opening brace of the class definition to the line "Here we implement the Subject Interface." with the text: "WeatherData now implements the Subject interface."
- An arrow points from the line "observers = new ArrayList();" to the text: "We've added an ArrayList to hold the Observers, and we create it in the constructor."
- An arrow points from the line "observers.add(o);" to the text: "When an observer registers, we just add it to the end of the list."
- An arrow points from the line "observers.remove(i);" to the text: "Likewise, when an observer wants to un-register, we just take it off the list."
- An arrow points from the line "observer.update(temperature, humidity, pressure);" to the text: "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."
- An arrow points from the line "measurementsChanged()" to the text: "We notify the Observers when we get updated measurements from the Weather Station."
- An arrow points from the line "measurementsChanged();" to the text: "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. WeatherData has ordered three: the current conditions display, the statistics display and the forecast display. Let's take a look at the current conditions display; once you have a good feel for this display element, check out the statistics and forecast displays in the head first code directory. You'll see they are very similar.

```
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");
    }
}
```

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.

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.

### <sup>there are no</sup> Dumb Questions

**Q:** Is update the best place to call display?

**A:** In this simple example it made sense to call display when the values changed. However, you are right, there are much better ways to design

the way the data gets displayed. We are going to see this when we get to the model view controller pattern.

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

**A:** True, but in the future we may want to unregister 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();

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

        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.*

*Create the three displays and pass them the WeatherData object.*

*Simulate new weather measurements.*

## 2 Run the code and let the server Pattern do its magic

```
File Edit Window Help StormyWeat er
%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

Jo nny H rricane, Weat er-O-Rama CEO t called, t ey can't po i ly ip wit o ta Heat Inde di play element. Here are t e detail :

T e heat inde i an inde t at com ine temperat re and midity to determine t e apparent temperat re ( ow of it act ally eel ). To comp te t e heat inde , yo ta et e temperat re, T, and t e relati e midity, RH, and et i orm la:

**heatindex =**

```
16.923 + 1.85212 * 10-1 * T + 5.37941 * RH - 1.00254 * 10-1 * T
* RH + 9.41695 * 10-3 * T2 + 7.28898 * 10-3 * RH2 + 3.45372 * 10-4
* T2 * RH - 8.14971 * 10-4 * T * RH2 + 1.02102 * 10-5 * T2 * RH2 -
3.8646 * 10-5 * T3 + 2.91583 * 10-5 * RH3 + 1.42721 * 10-6 * T3 * RH
+ 1.97483 * 10-7 * T * RH3 - 2.18429 * 10-8 * T3 * RH2 + 8.43296 *
10-10 * T2 * RH3 - 4.81975 * 10-11 * T3 * RH3
```

So et typin !

J t iddin . Don t worry, yo wont a e to type t at orm la in; t create yo r own HeatInde Di play. a a le and copy t e orm la rom eatinde .t t into it.

You can get heatindex.txt from headfirstlabs.com

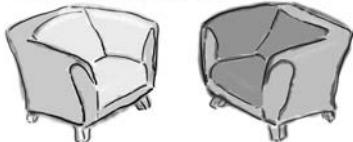
How doe it wor ? o d a e to reer to *Head First Meteorology*, or try a in omeone at t e National Weat er Ser ice (or try a Goo le earc ).

W en yo ni , yo ro tp t o ld loo li et i :

Here's what changed  
in this output

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

## Fireside Chats



Tonight's talk: **A S ect and ser er spar o er the r ght  
ay to get state n ormat on to the ser er.**

### S ect

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

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

h yeah, like what?

Well e se me. have to send my state with my notifications so all you la y bservers will know what happened

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

### ser er

really? I thought you didn't care much about us bservers.

Well yeah, but that's just a small part of who am. Anyway, know a lot more about you...

Well, you're always passing your state around to us bservers so we can see what's going on inside you. Which gets a little annoying at times...

h, 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?

## S ect

es, could let you **p ll** 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 like **p sh** better... then you have everything you need in one notification.

Well, can see the advantages to doing it both ways. I have noticed that there is a built in `ava bserver` Pattern that allows you to use either push or pull.

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

## ser er

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

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

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

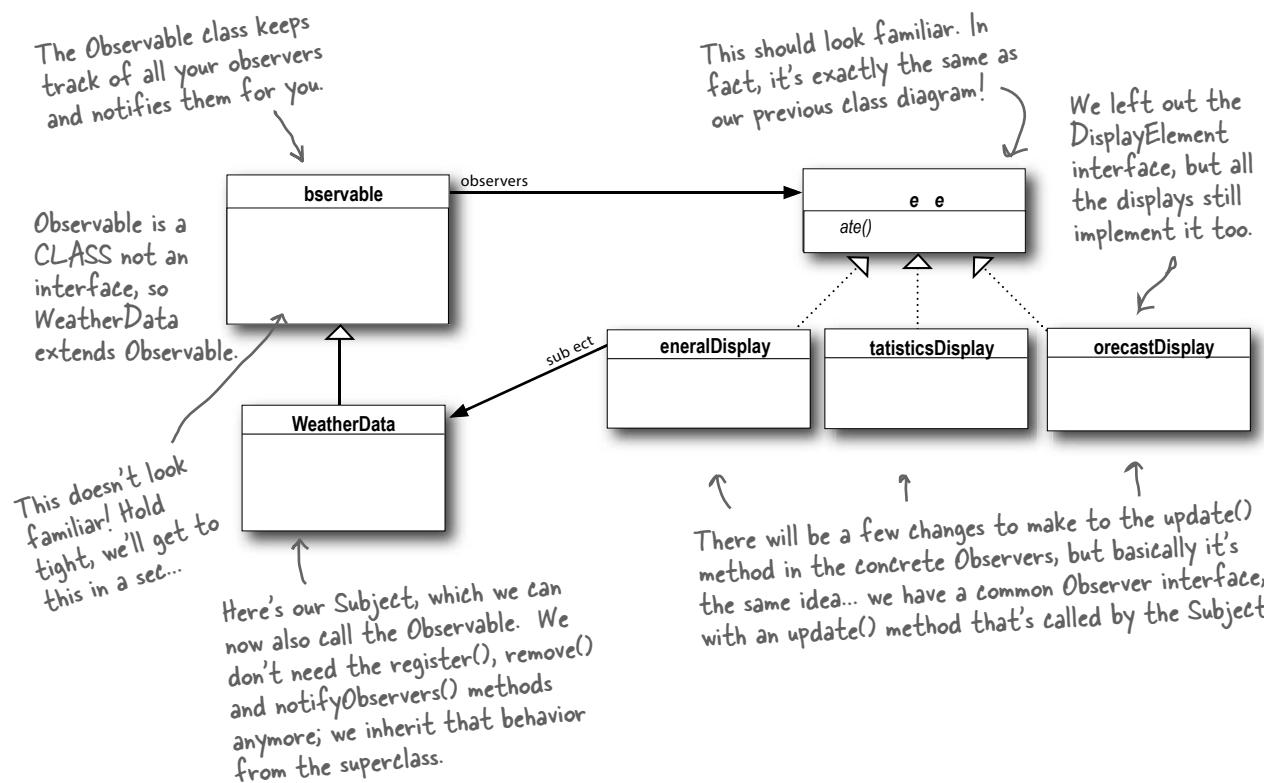
What, us agree on something? I guess there's always hope.

# Using Java's built-in Observer Pattern

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

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

With Java's built-in support, all you have to do is extend Observable and tell it when to notify the Observers. The API does the rest for you.



# How Java's built-in Observer Pattern works

The built-in Observable 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 `Subject` 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 just call the `setChanged` method to signify that the state has changed in your object**

**② Then, call one of two `notify` servers methods**

either `notifyObservers()` or `notifyObservers(Object arg)`

This version takes an arbitrary data object that gets passed to each `Observer` when it is notified.

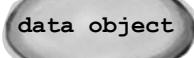
## For an `Observer` to receive notifications

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

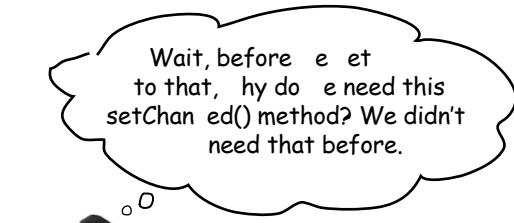
`update(Observable o, Object arg)`

The Subject that sent the notification is passed in as this argument.

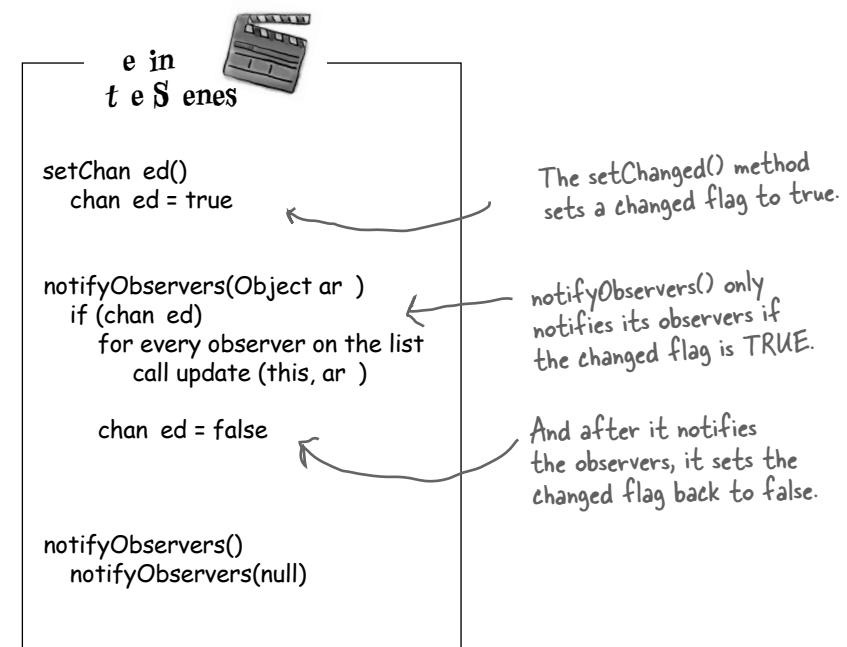
This will be the data object that was passed to `notifyObservers()`, or null if a data object wasn't specified.



If you want to push data to the observers you can pass the data as a `data object` to the `notifyObserver(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.



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

- 1 Make sure we are importing the right Observer/Observable.

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

- 2 We are now subclassing Observable.

```
public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers();
    }

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

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

- 3 We don't need to keep track of our observers anymore, or manage their registration and removal; (the superclass will handle that) so we've removed the code for register, add and notify.

- Our constructor no longer needs to create a data structure to hold Observers.

Notice we aren't sending a data object with the `notifyObservers()` call. That means we're using the PULL model.

- We now first call `setChanged()` to indicate the state has changed before calling `notifyObservers()`.

These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

urrent condition re or

## Now, let's rework the CurrentConditionsDisplay

- ① Again, make sure we are importing the right Observer/Observable.

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

public class CurrentConditionsDisplay implements Observable, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

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

- ② We now are implementing the Observer interface from java.util.

- ③ Our constructor now takes an Observable and we use this to add the current conditions object as an Observer.

- We've changed the update() method to take both an Observable and the optional data argument.

- In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().



## 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;
```

te t dri e

## Running the new code

**ust to e sure, let's run the new code**

```
File Edit Window Help TryTi i AtHome
%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
%
```

**H , do you notice anything different? oo again**

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

### Never depend on order of evaluation of the server 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? either; we just chose to implement things in different ways.

What would be incorrect, however, is if we wrote our code to `expect` 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.

ha ter



## The dark side of `java.util.Observable`

es, 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.

### **servable 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 *inherit* 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` P. Or do you have the option of swapping out the `java.util` implementation for another (say, a new, multi-threaded implementation).

### **servable protects crucial methods**

If you look at the `Observable` P, 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: *from site to interface*.

### **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 P that makes use of the pattern.

## Other places you'll find the Observer Pattern in the DK

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

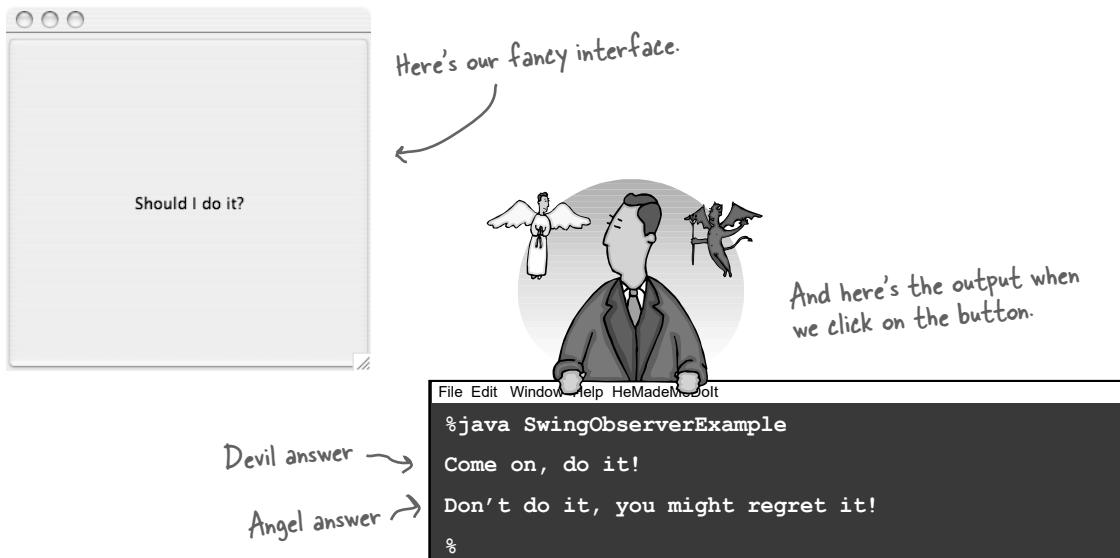
If you're curious about the Observer Pattern in JavaBeans check out the `PropertyChangeListener` interface.

### little ac ground

Let's take a look at a simple part of the swing P, the `Button`. If you look under the hood at `Button`'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 P.

### 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...

his life changing application requires very little code. All we need to do is create a Button object, add it to a Frame 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 Started 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());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // Set frame properties here
    }

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

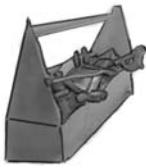
    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

*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.*

*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.*



# Tools for your Design Toolbox

**elco e to the end of chapter  
ou've added a few new things to your  
tool o**



**OO Basics**

- Abstraction
- Iteration
- Hism
- Cee

**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**

- Strâ  
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!



Exercise



## Design Principle Challenge

For each design principle, describe how the `bserver` pattern makes use of the principle.

e i n r i n i e

de tiyt eas e ts o yo ra li atio t at ary  
a d se arate t e ro at stays t e sa e

---

---

---

---

---

e i n r i n i e

rogra to a i tera e ota i le e tatio

---

---

---

---

---

e i n r i n i e

Fa or o ositio o eri erita e

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

---

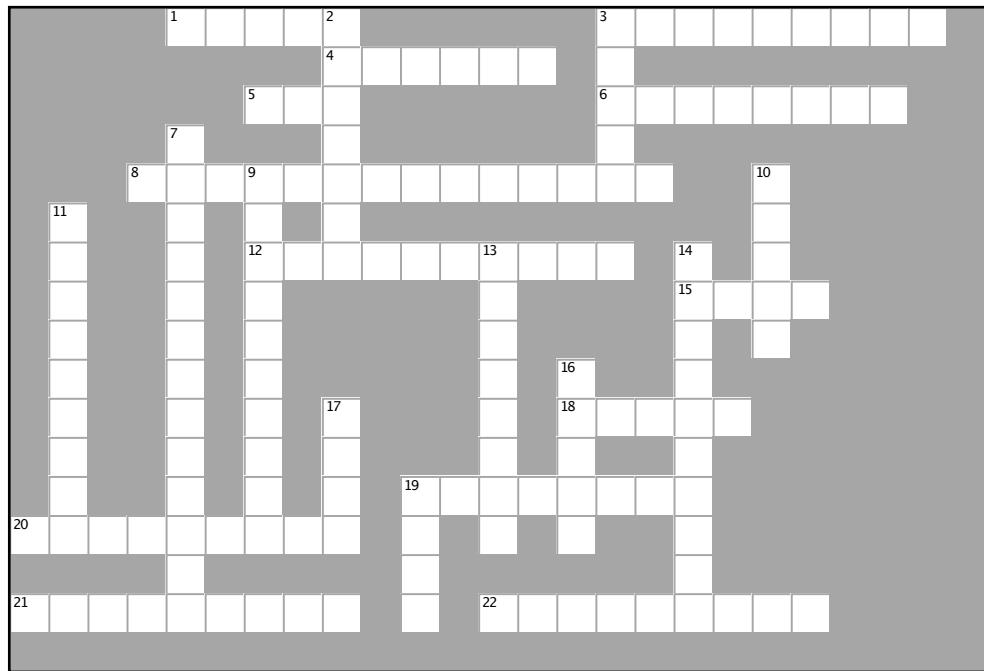
---

---

---

---

you are here ▶



#'\$\$

\$" 4 =M@P<=@M<;;;;;; ; I JN<I DND@>@  
 &" / @D<I ? - I B@<L@ ;;;;; NJ N@=ONI  
 ' " 2H K@H @NNDM H@CJ? NJ B@N JNA@  
 (" 3@BJNJI @JAC@JQI  
 ) ". C@N JI ?D@I M DNGS D@K@ @N NND  
 DND@>@  
 +" 1JQ N B@SJCLM@J AND@4 =M@P@ @N  
 \$%": JOA@LB@NN@MASJOL@ JNB@ND B@ JNA@  
 QG@ SJON@F SJOMCJ@3 =@  
 \$(\$ 41@T O=E@N@G@IN N@G N ;;;; J=M@P@M  
 \$+/" JI N>J@NJI NNDMAILI JNA@>NJI  
 \$, " 8@H K@<N@O@OCH D@<I ? ;;;;;;  
 %%" 4 =M@P@M<L@ ;;;;;; JI N@O=E@N  
 %%" 5WBL@H N < ;;;;;; I JN<I  
 D@K@ @NND@I  
 %% - 7O=E@NNDMD@BLN < ;;;;;;

%"!

% 6 JI Q<M=JNC <I 4 =M@P@ <I ? < ;;;;;;  
 &" : JOQ<I NN F@X SJCL>JOK@B ;;;;;;  
 \* " 1 @MSMSJOMCJ@3 BJ AIL@N  
 , " ;;;; >I H<I <B@SJCLJ=M@P@MAILSJO  
 \$%" 3<H @QJLF Q@DC GNMJA4 =M@P@M  
 \$\$" 9 @N@4 !6 <H < M. 04 I <H @ <A@ NND  
 FD? JAMNLH  
 \$&" 4 =M@P@M@&@N =@ ;;;;;; QC@  
 MJH @ODB I @C<KK@M  
 \$' " 8C@9 @N@<N>GMM; ;;;;;; N@  
 7O=E@NND@A>@  
 \$) " 1 @?D@I NQ<I N@I S H JL@DNM MJ C@L@H JP@  
 CDH M@A  
 \$\* ". 04 <GJMN@LB@N@ ;;;; D?@R?DNGS  
 \$, " 7O=E@NND@B@Q<I N@<N@?<N@  
 N 4 =M@P@



## er ise solutions

### 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 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 Weather data class.



### Design Principle Challenge

#### e i n r i n i e

de tiyt eas e ts o yo ra li atio t at  
ary a dse arate t e ro at stayst e  
sa e

#### 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!

#### e i n r i n i e

rogra to a i tera e ota i le e tatio

#### 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.

#### e i n r i n i e

Fa or o ositio o eri erita e

#### 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!

you are here ▶



# Code Magnets

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

public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

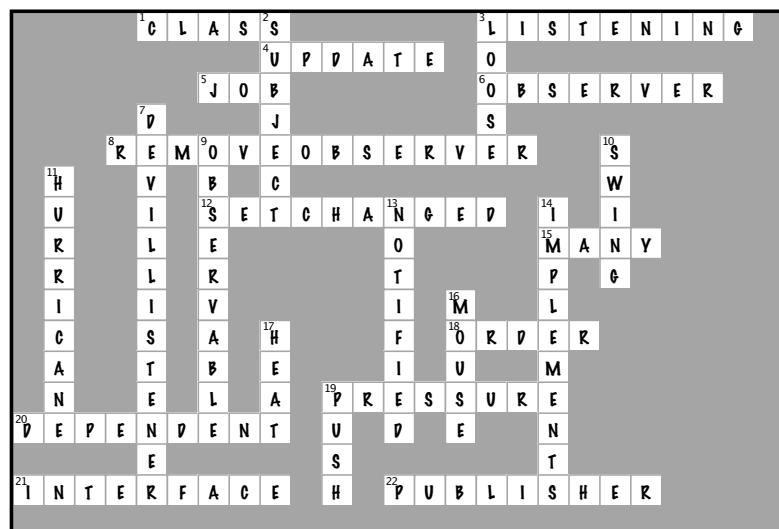
    public ForecastDisplay(Observable observable) {
        observable.addObserver(this);
    }

    public void update(Observable observable,
Object arg) {
        if (observable instanceof WeatherData) {
            WeatherData weatherData =
(WeatherData)observable;
            lastPressure = currentPressure;
            currentPressure = weatherData.getPr
            display();
        }
    }

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



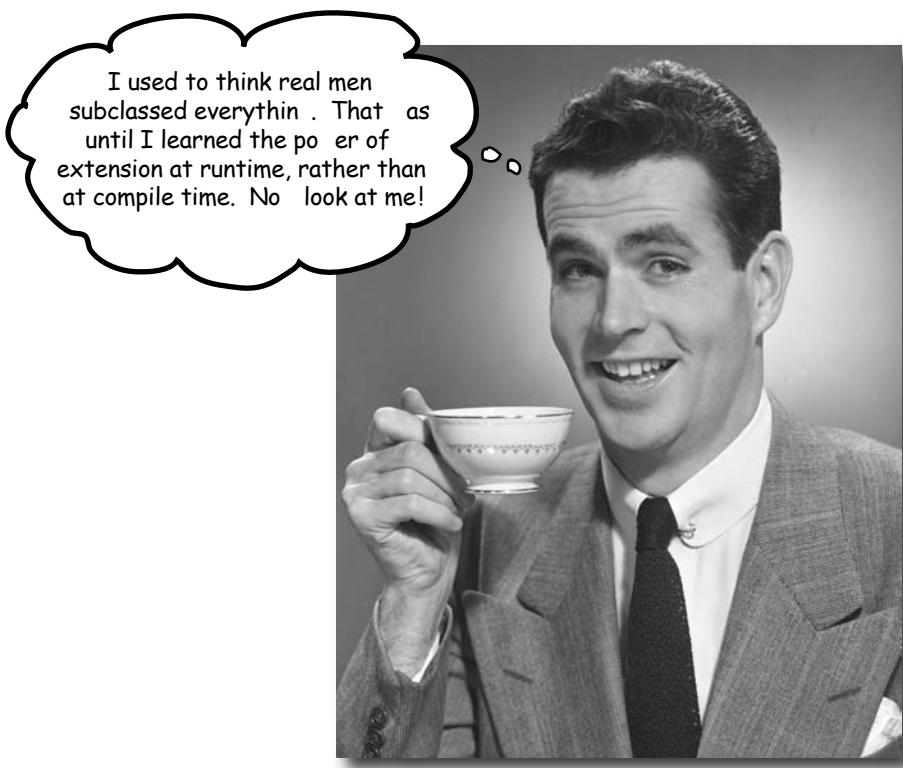
er ise  
solutions



the De orator attern



# Decorating Effects



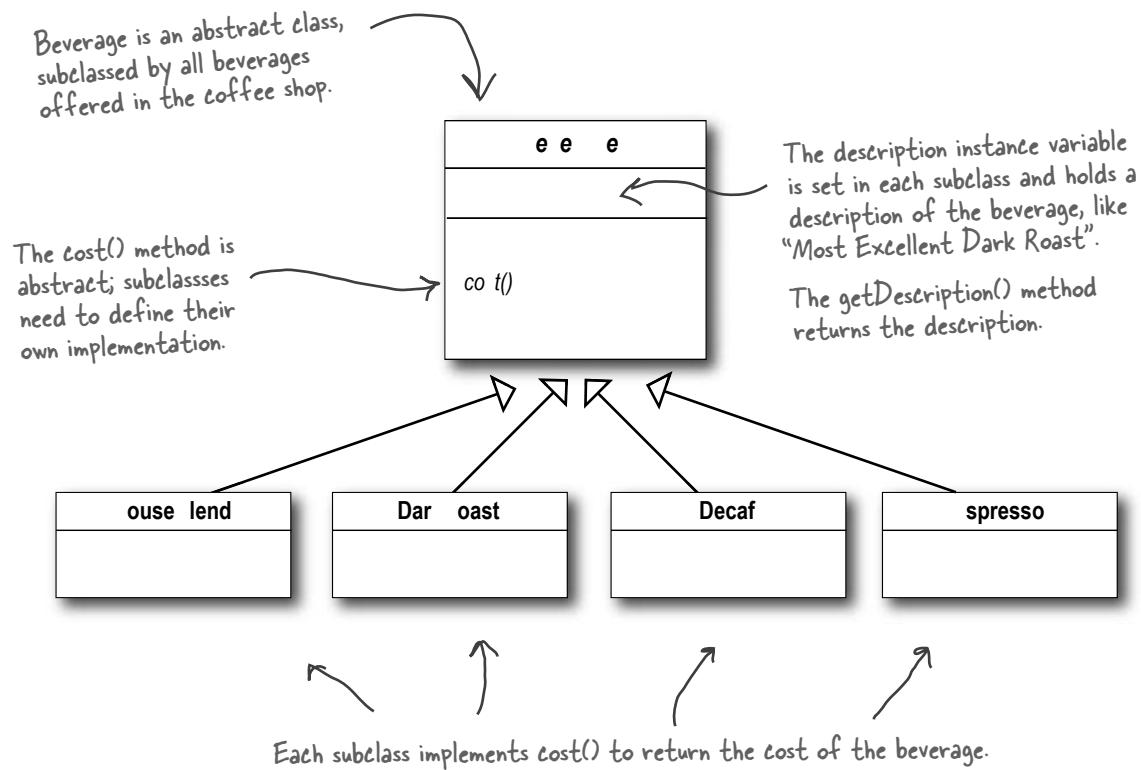
We'll re-examine the typical order of inheritance and you'll learn how to decorate your class at runtime in a form of object composition. Why? Once you know the technique of decoration, you'll be able to impress (or someone else) by creating new representations without changing a gesture or adding a dependency.

## Welcome to Starbuzz Coffee

tar u 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.

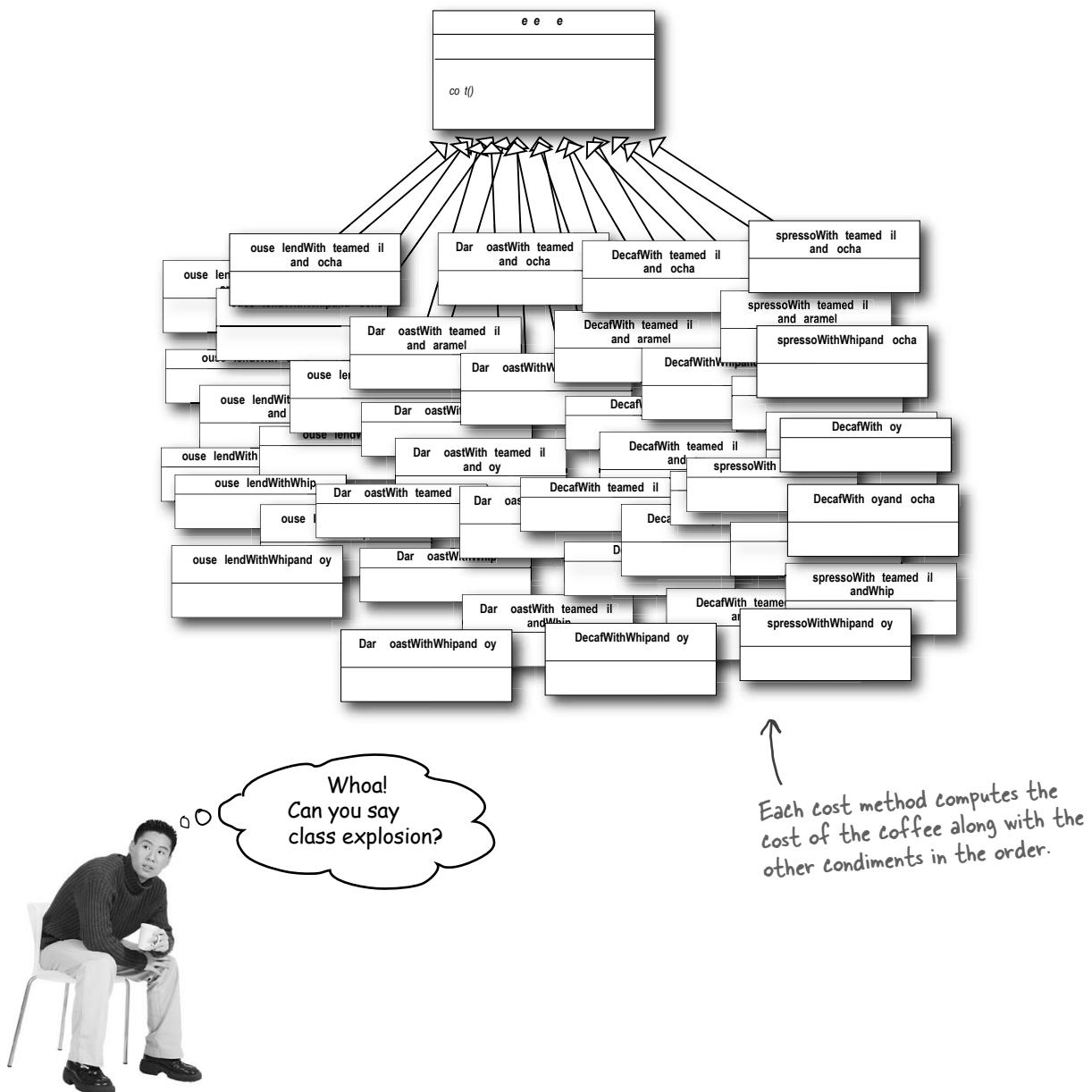
Because they've grown so quickly, they're scrambling to update their ordering system to match their beverage offerings.

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



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

Here's their first attempt:



you are here ▶

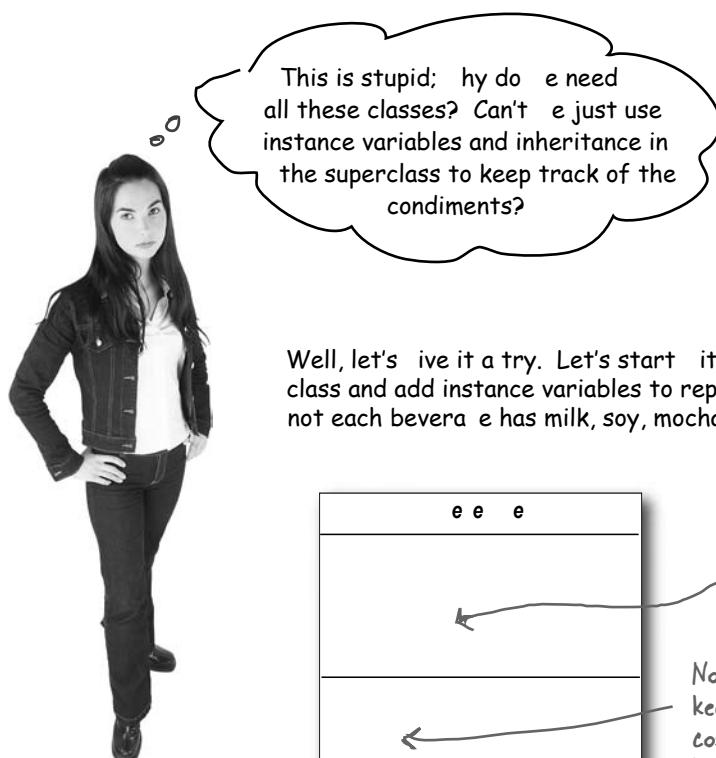


## RA POWER

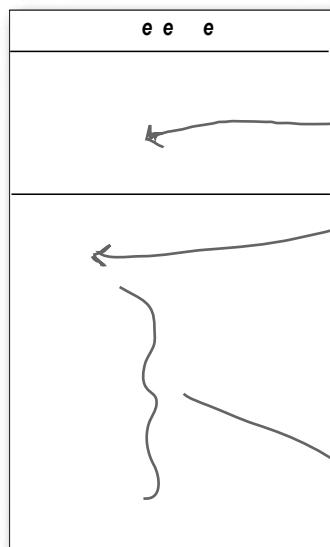
It pretty o io t at Star a created a maintenance ni tmare or t em el e . W at appen w ent e price o mil oe p? W at do t ey do w ent ey add a new caramel toppin ?

T in in eyond t e maintenance pro lem, w ic o t e de i n principle t at we e co ered o ar are t ey iolatin ?

Hi t e yre iolati g t o o t e i a ig ay



Well, let's give it a try. Let's start ith the Bevera e base class and add instance variables to represent hether or not each bevera e has milk, soy, mocha and hip...



New boolean values for each condiment.

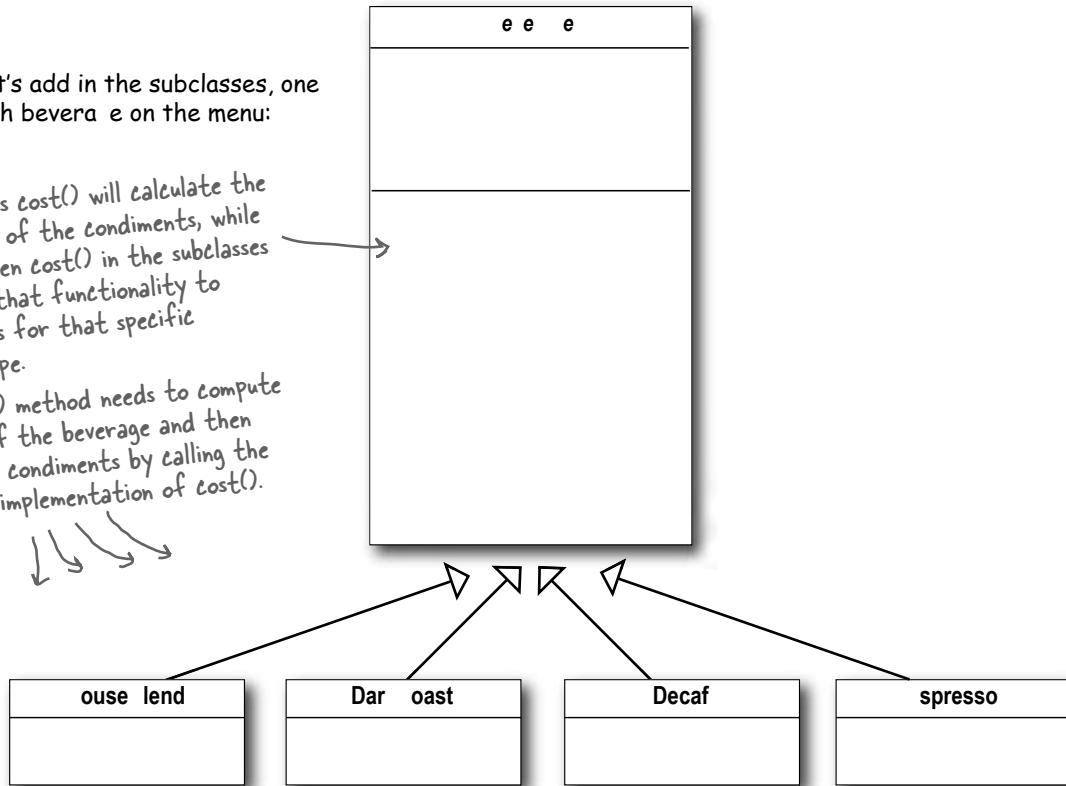
Now we'll implement `cost()` in `Beverage` (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override `cost()`, but they will also invoke the super version so that they can calculate the total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

These get and set the boolean values for the condiments.

No let's add in the subclasses, one for each beverage on the menu:

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

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

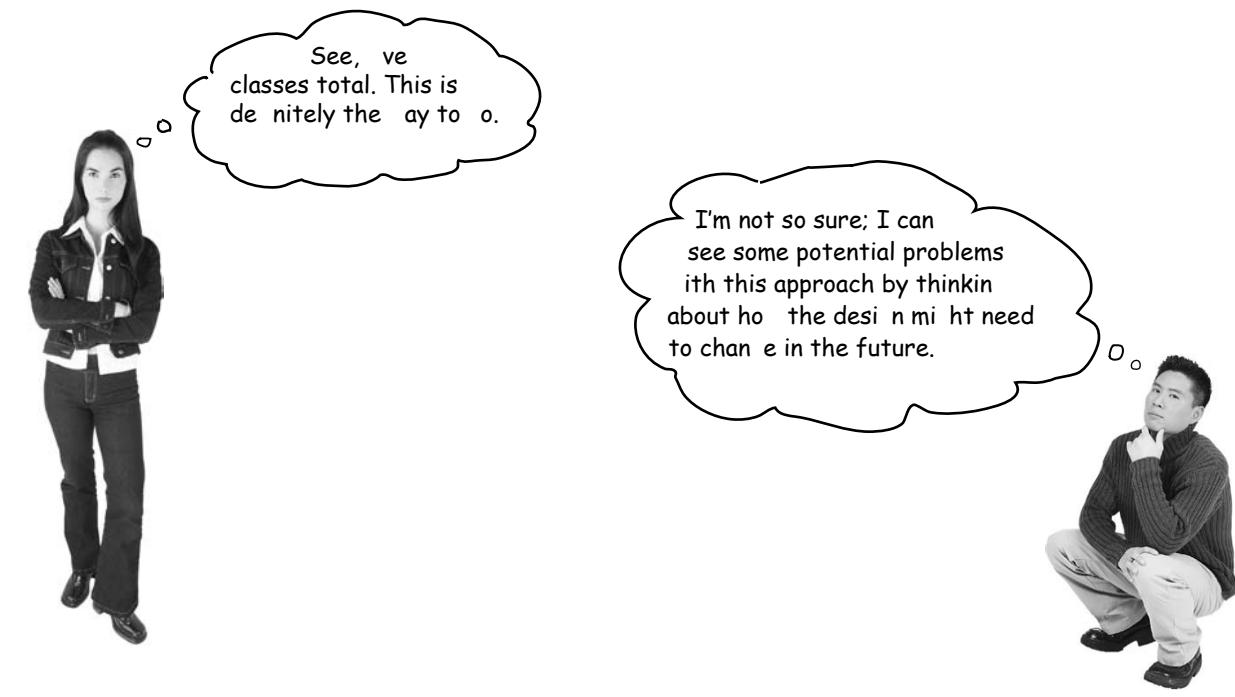


## Sharpen your pencil

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

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

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



## Sharpen your pencil

What requirement or other actor might cause a class to 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:

Answer



**a ter and tudent**

**a ter** rass o er it as ee so eti es i eo r last  
eeti g Ha e yo ee dee i editatio o i erita e  
**tudent** es Master ile i erita e is o er l a e  
lear ed t at it does tal ays lead to t e ost e i le or  
ai tai a le desig s

**a ter** yes yo a e ade so e rogress o tell e yst de t o  
t e ill yo a ie e re se i ott ro g i erita e

**tudent** Master a elear ed t ere are ays o i eriti g e a ior at  
r ti et ro g o ositio a d delegatio

**a ter** lease go o

**tudent** e i erit e a ior ys lassi g t at e a ior is set stati ally  
at o illet i e additio all s lasses sti eritt e sa e e a ior  
o e er a e te da o e ts e a iort ro g o ositio t e a do  
t is dy a i ally at r ti e

**a ter** ery good rass o er yo are egi i g to see t e o ero  
o ositio

**tudent** es it is ossi le or e to add Iti le e res o si ilities to o e ts  
t ro g t iste i e i l di gres o si ilities t at ere ote e t o g to  
y t e desig ero t e s er lass d do t a e to to t eir ode

**a ter** at a e yo lear ed a o tt e e e to o ositio o ai tai i g  
yo r ode

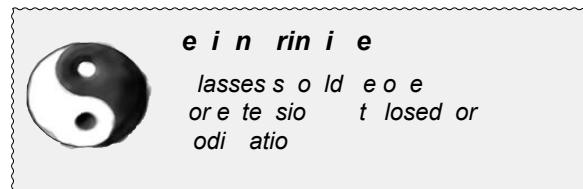
**tudent** ell t at is at as getti g at ydy a i ally o osi go e ts  
a add e tio ality y rit i g e o de rat ert a alteri g e isti g  
ode e a se ot a gi g e isti g ode t e a es o i trod i g gs  
or a si g i te ded side e e ts i re e isti g ode are red ed

**a ter** ery good o g or today rass o er o ld li e or yo to  
go a d ed itate rt ero t is to i e e er odes o ld e losed to  
a ge li et elot s o eri t e e e i g yet o e to e te sio li e t e  
lot s o eri t e or i g

you are here ▶

## The Open-Closed Principle

**rasshopper is on to one of the most important design principles**



Come on in; we're *open*. Feel free to *attend* 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 *expectations*.



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.

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

## there are no Dumb Questions

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

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

As it turns out, though, there are some clever techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the observer pattern in chapter ... 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 design techniques.

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

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

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

**A:** Usually, you can't. Making a 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 taking 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 our designs and apply the principles there.

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

**A:** This is partly a matter of experience in designing 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. Extending code is wasteful, 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, 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

Okay, enough of the Object Oriented Design Club. We have real problems here! Remember us? Starbucks Coffee? Do you think you could use some of those design principles to actually help us?



**① Create a Beverage abstract class**

**② Decorate it with a MochaDecorator**

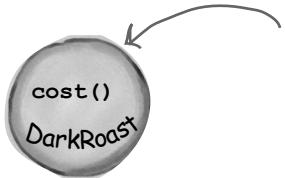
**③ Decorate it with a WhippedCreamDecorator**

**● calculate total 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? Hint: think of decorator objects as wrappers. Let's see how this works...

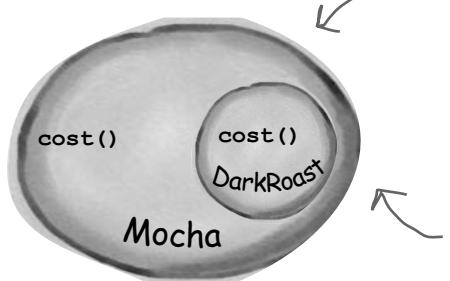
# Constructing a drink order with Decorators

- 1 We start with our DarkRoast object**



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

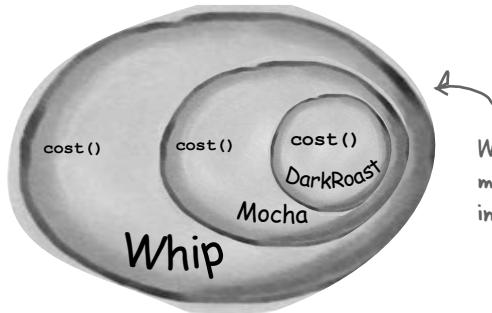
- 2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast**



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

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

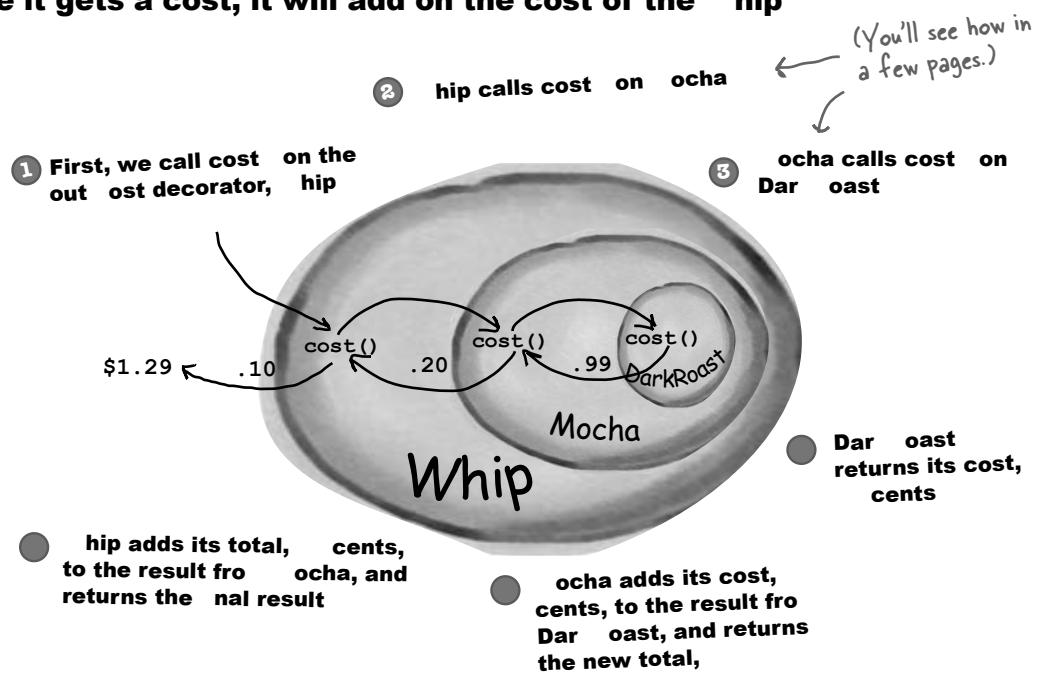
- 3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it**



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

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

- 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 `hip`.



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

- 
- 
- 
- 
- 

Key Point!

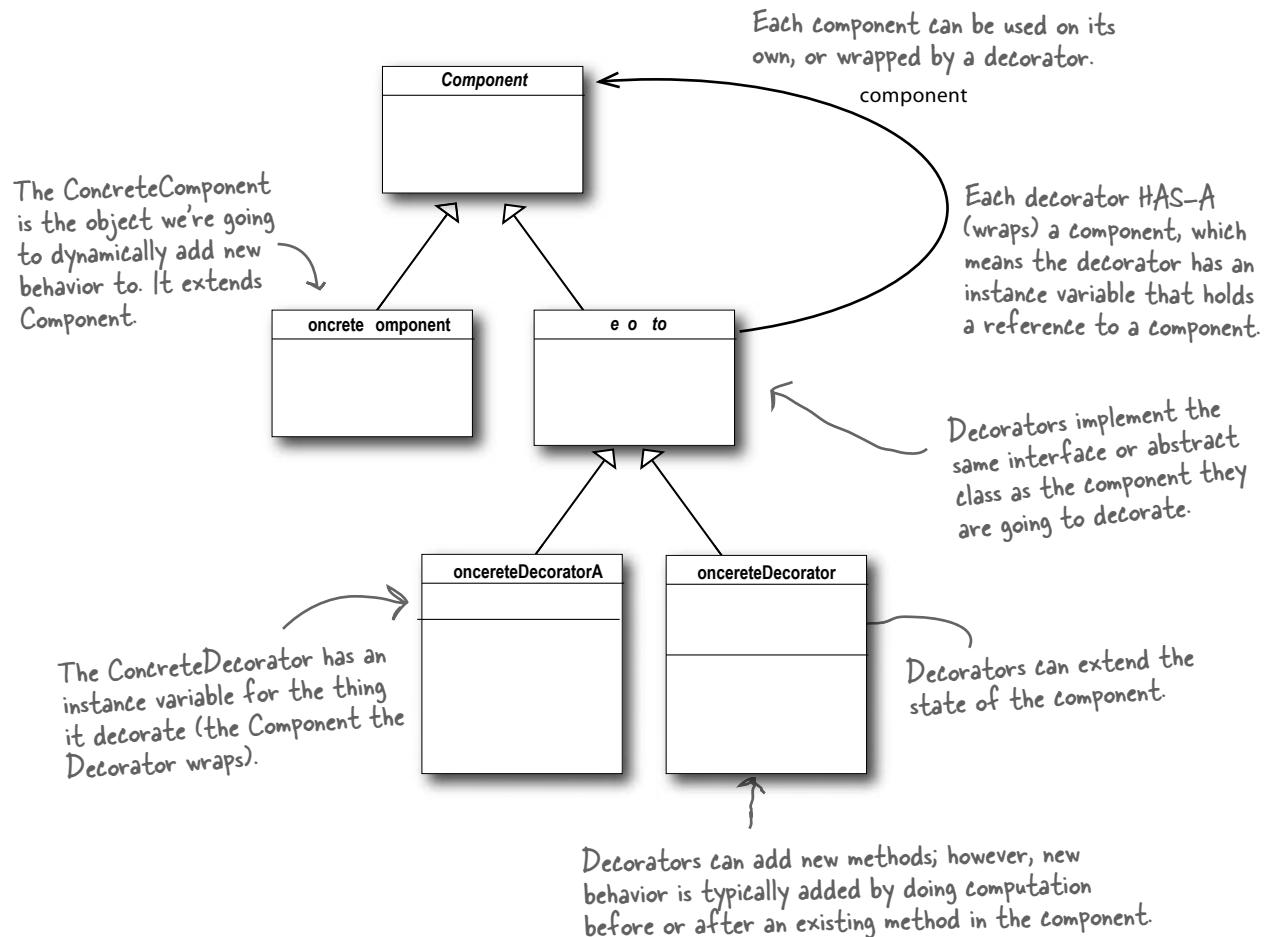
Now let's see how this all really works by looking at the **Decorator Pattern** definition and writing some code.

# The Decorator Pattern defined

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

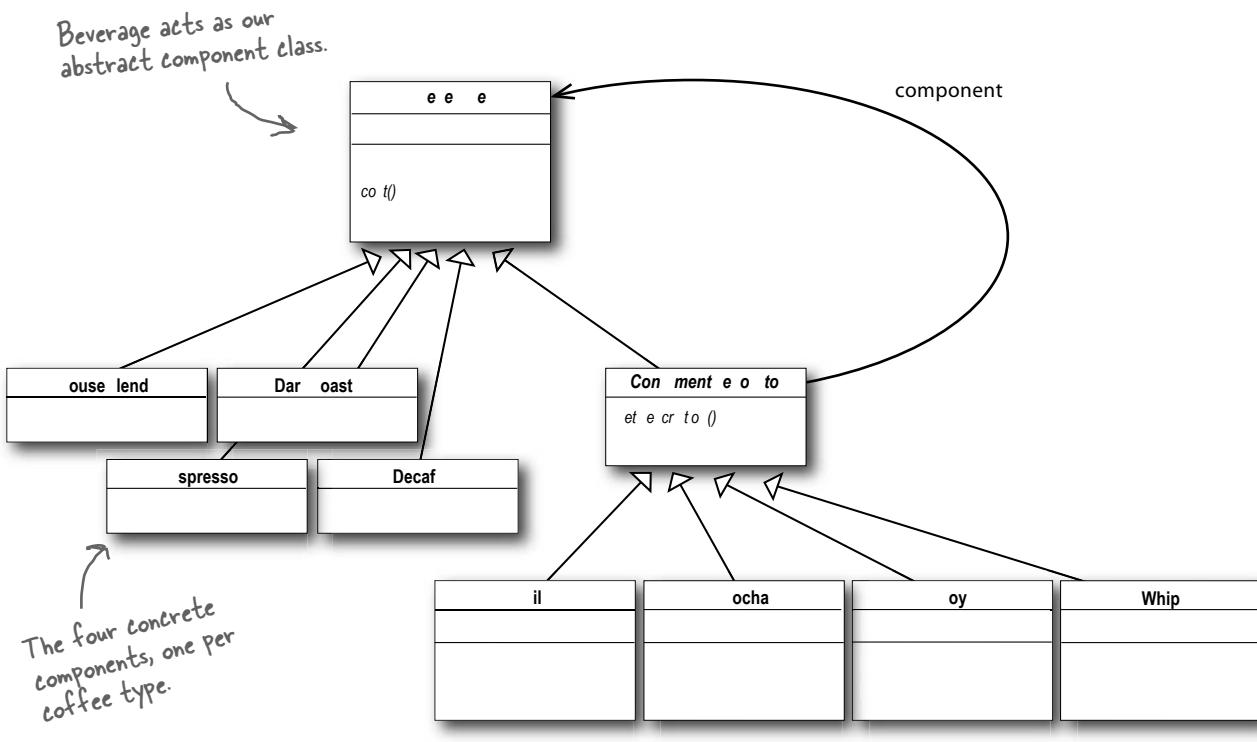
**The Decorator pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

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



## Decorating our Beverages

ay, let's wor our tar u everages into this fra ewor



The four concrete components, one per coffee type.

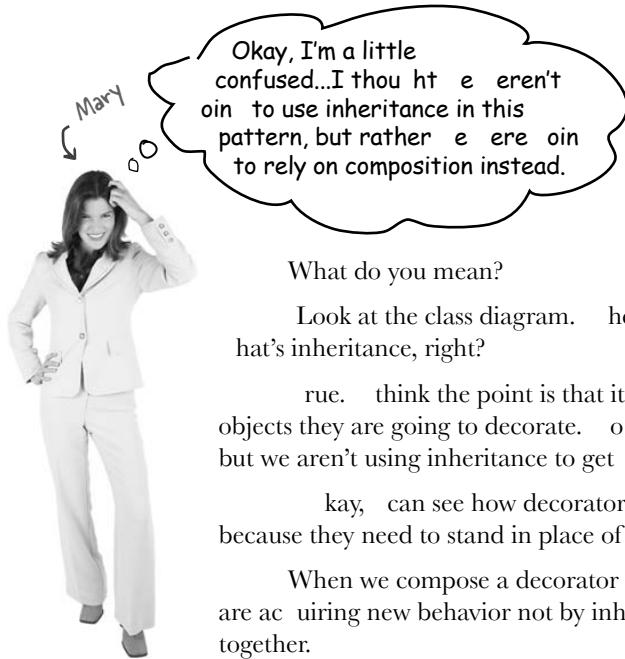
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...



Before we implement the cost() method for coffee and the condiment. All in all, we need to implement the getDescription() method for the condiment.

# Cubicle Conversation

## o e confusion over nheritance versus o position



What do you mean?

Look at the class diagram. The Condiment Decorator is extending the Beverage class. That's inheritance, right?

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 *fly weight*, but we aren't using inheritance to get *efficiency*.

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?

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.

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.

That's right.

So, 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.

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*

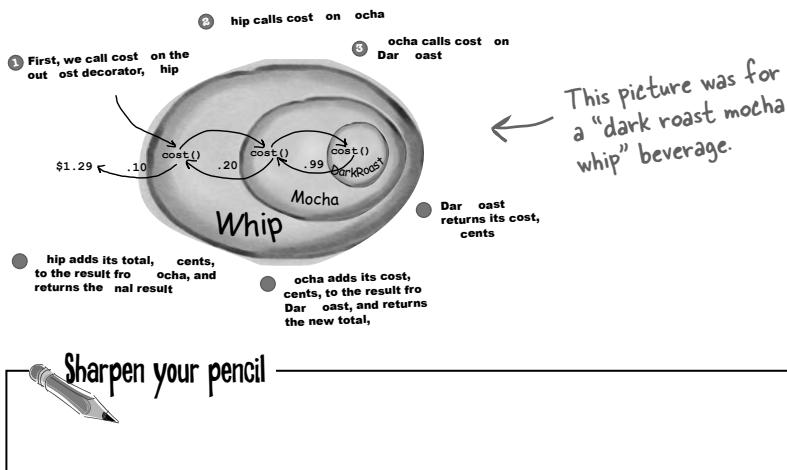
And as I understand it, we can implement new decorators at any time to add new behavior. If we relied on inheritance, we'd have to go in and change existing code any time we wanted new behavior.

Exactly.

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?

Well, remember, when we got this code, it was already 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



Sharpen your pencil

Okay, I need for you to make me a double mocha, soy latte ith hip.



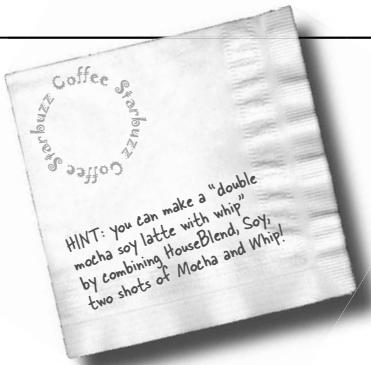
## Starbuzz Coffee

### Coffees

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

### Condiments

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



# Writing the Starbuzz code

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



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

```
public abstract class Beverage {
    String description = "Unknown Beverage";
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
```

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

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

Beverage is simple enough. Let's implement the abstract class for the condiments Decorator as well.

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

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

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

## Coding beverages

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

```
public class Espresso extends Beverage {  
  
    public Espresso() {  
        description = "Espresso";  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

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

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 classes (DarkRoast and Decaf) in exactly the same way.

Starbuzz Coffee		
<u>Coffees</u>		
House Blend		.89
Dark Roast		.99
Decaf		1.05
Espresso		1.99
<u>Condiments</u>		
Steamed Milk		.10
Mocha		.20
Soy		.15
Whip		.10

## Coding condiments

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

```
Mocha is a decorator, so we
extend CondimentDecorator.
```

```
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 .20 + beverage.cost();
    }
}
```

Remember, CondimentDecorator  
extends Beverage.

We're going to instantiate Mocha with  
a reference to a Beverage using:  
 (1) An instance variable to hold the  
beverage we are wrapping.  
 (2) A way to set this instance  
variable to the object we are  
wrapping. Here, we're going to pass  
the beverage we're wrapping to the  
decorator's constructor.

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

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

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



Sharpen your pencil

## Serving some coffees

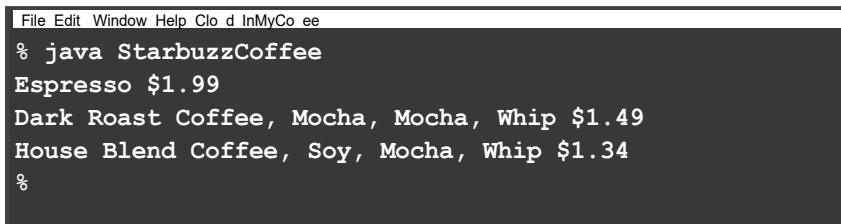
Congratulations. It's time to sit back, order a few coffees and marvel at the flexible design you created with the Decorator Pattern.

### Here's some test code\* to see orders

```
public class StarbuzzCoffee {  
  
    public static void main(String args[]) {  
        Beverage beverage = new Espresso(); ← Order up an espresso, no condiments  
        System.out.println(beverage.getDescription()  
            + " $" + beverage.cost());  
  
        Beverage beverage2 = new DarkRoast(); ← Make a DarkRoast object.  
        beverage2 = new Mocha(beverage2); ← Wrap it with a Mocha.  
        beverage2 = new Mocha(beverage2); ← Wrap it in a second Mocha.  
        beverage2 = new Whip(beverage2); ← Wrap it in a Whip.  
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());  
  
        Beverage beverage3 = new HouseBlend(); ← Finally, give us a HouseBlend  
        beverage3 = new Soy(beverage3); ← with Soy, Mocha, and Whip.  
        beverage3 = new Mocha(beverage3);  
        beverage3 = new Whip(beverage3);  
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());  
    }  
}
```

\* We're going to see a much better way of creating decorated objects when we cover the Factory Pattern (and the Builder Pattern, which is covered in the appendix).

### Now, let's get those orders in



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

## there are no Dumb Questions

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

**A:**

**Q:** Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? i.e if I had a `Dar oast` with `ocha oy` and `Whip` it would be easy to write code that somehow ended up with a reference to `oy` instead of `Whip` which means it would not include `Whip` in the order

**A:**

**Q:** an decorators now about the other decorations in the chain? say I wanted my `getDescription` method to print `Whip Double ocha` instead of `ocha Whip ocha`? what would require that my outermost decorator now all the decorators it is wrapping

**A:**



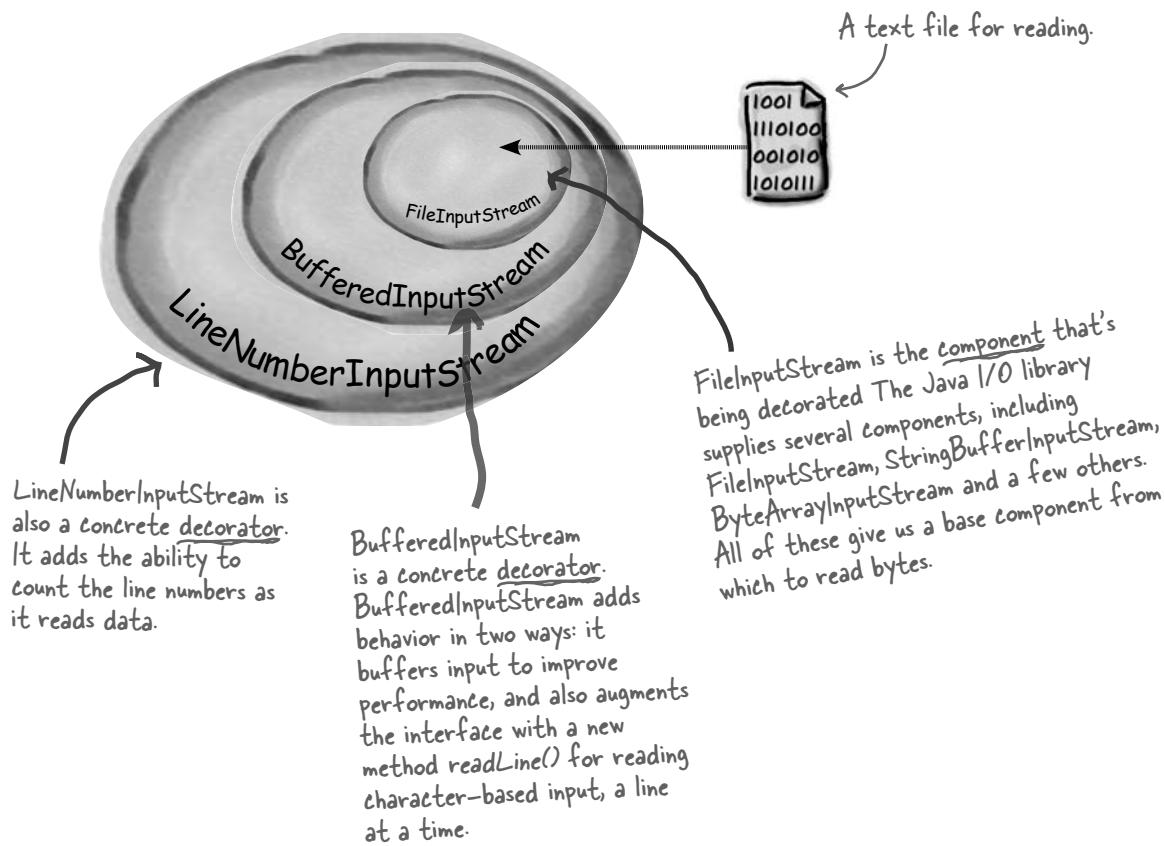
### Sharpen your pencil

ur friends at Tarbu have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation small, medium, and large). Tarbu saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: `set size()` and `get size()`. They'd also like for the condiments to be charged according to size, so for instance, `oy` costs `0.00`, `and 0.10` respectively for tall, grande, and venti coffees.

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

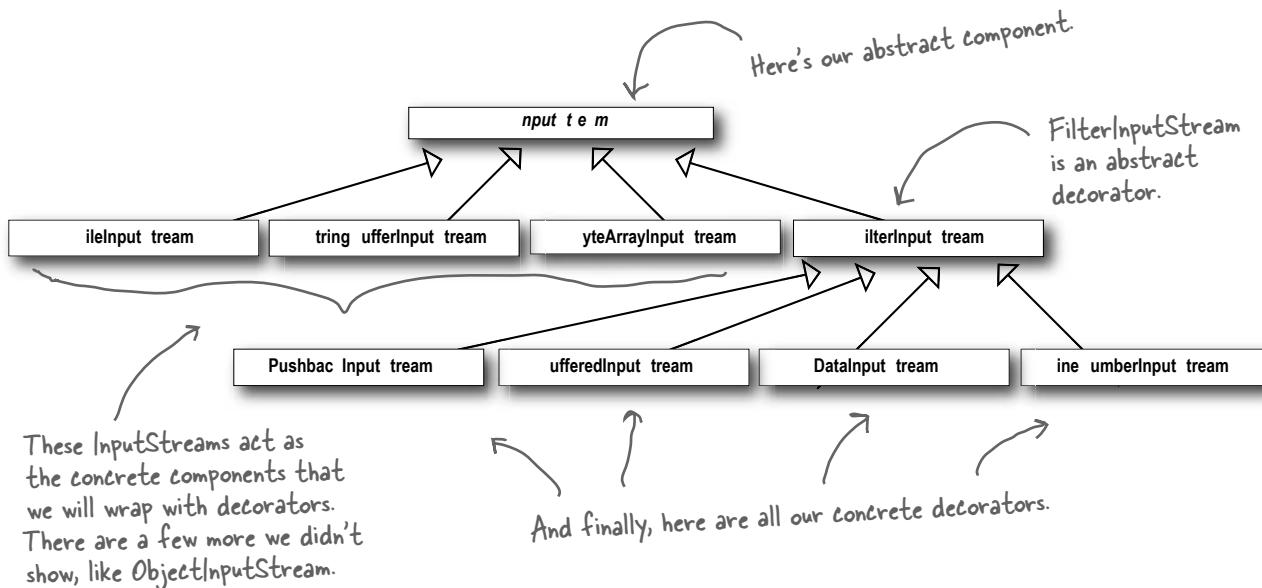
## Real World Decorators: `aval 0`

The large number of classes in the `java.io` package is... *er e mi g.* Don't feel alone if you said "whoa" the first (and second and third) time you looked at this P. But now that you know the Decorator Pattern, the 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



**Buffere** Input Stream and **line u er** Input Stream both extend **Iterator** Input Stream, which acts as the abstract decorator class.

## Decorating the java.io classes



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

And you'll see that the `it` streams have the same design. And you've probably already found that the `eader 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).

But `ava` also points out one of the `si es` of the `ecorator 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 `ecorator based P`. But now that you know how `ecorator` works, you can keep things in perspective and when you're using someone else's `ecorator heavy P`, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

# t a a O ec at

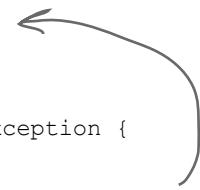
O a t e ec at atte e  
see t e O class a a s l be ea f  
te tec at  
  
ab tt s tea ec at t atc e ts  
all e casec a acte st l e case t e  
t st ea t e s e ea  
t e ec at atte t ee e  
t e ec at c e t st st t e  
ec at atte t ee e le

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 = super.read();  
        return (c == -1 ? c : Character.toLowerCase((char)c));  
    }  
  
    public int read(byte[] b, int offset, int len) throws IOException {  
        int result = super.read(b, offset, len);  
        for (int i = offset; i < offset+result; i++) {  
            b[i] = (byte)Character.toLowerCase((char)b[i]);  
        }  
        return result;  
    }  
}
```

No problem. I just have to  
extend the FilterInputStream class  
and override the read() methods.



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

ha ter

# Test out your new `avaI O Decorator`

Write some slick 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();
        }
    }
}
```

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

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!

te t.t le

You need to make this file.

Give it a spin:

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



Welcome ecorator Pattern. We've heard that you've been a bit down on yourself lately?

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

Can you perhaps share some of your troubles with us?

ure. Well, you know 've got the power to add e ability to designs, that much is for sure, but also have a *r si e*. ou see, 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.

Can you give us an e ample?

ake the ava libraries. hese are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an nput tream, life would be much easier.

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

here's more, 'm afraid. '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. ow, one great thing about me is that ***you can usually insert decorators transparently and the client never has to now it's dealin with a decorator*** But like said, some code is dependent on specific types and when you start introducing decorators, boom Bad things happen.

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

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

'll be interviewing the Factory and Builder patterns ne t week hear they can be very helpful with this?

hat's true; should talk to those guys more often.

Well, we all think you're a great pattern for creating e able designs and staying true to the pen Closed Principle, so keep your chin up and think positively

'll do my best, thank you.



# Tools for your Design Toolbox

**ou've got another chapter under your belt and a new principle and pattern in the tool box**



**OO Basics**

- 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.

**OO Patterns**

- Strategy
- Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Facade
- Observer
- Singleton
- Factory Method
- Abstract Factory
- Builder
- Composite
- Adapter
- Bridge
- Proxy

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.

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?

*you are here* ►

# Exercise solutions

```

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

    public double cost() {
        double condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
}

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

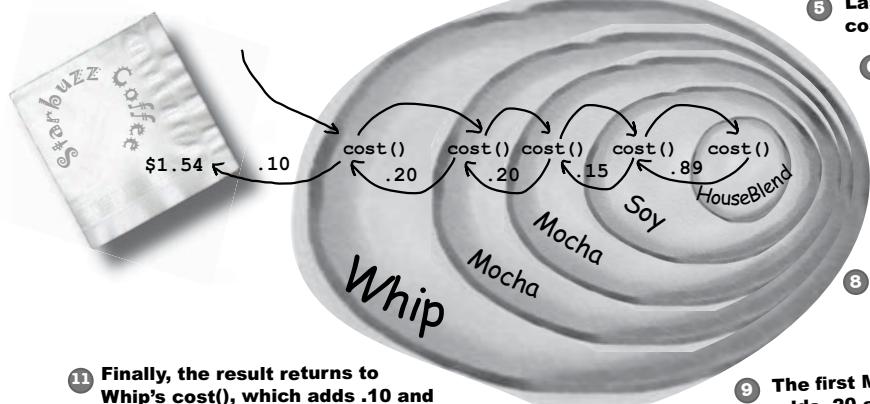
    public double cost() {
        return 1.99 + super.cost();
    }
}

```

## New barista training



"double mocha soy latte with whip"



# er ise solutions

ur friends at tarbu have introduced si es to their menu. ou can now order a coffee in tall, grande, and venti si es (for us normal folk small, medium, and large). tarbu saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class set i e() and get i e(). hey'd also like for the condiments to be charged according to si e, so for instance, oy costs , , and respectively for tall, grande, and venti coffees.

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

```
public class Soy extends CondimentDecorator {
    Beverage beverage;

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

    public int getSize() {
        return beverage.getSize();
    }

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

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

Now we need to propagate the getSize() method to the wrapped beverage. We should also move this method to the abstract class since it's used in all condiment decorators.

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

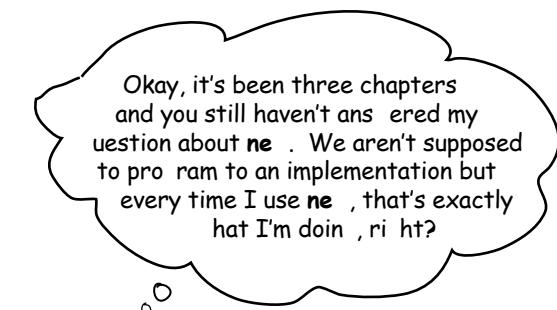


the a tor attern

oodness



There is more to  
main object than the **operator**. You will learn that instantiation is an activity that  
shouldn't always be done in public and can often lead to **coupling**. And you don't want  
that, do you? Find out how Factory Pattern can help you from memory dependencies.



### **When you see new , thin concrete**

es, 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!

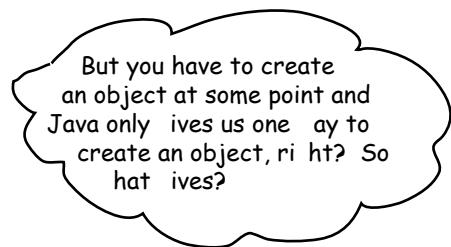
When you have a whole set of related concrete classes, often you're forced to write code like this

```
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.



### hat'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 CHG 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. In other words, your code will not be closed for modification. If you add new concrete types, you'll have to reopen it.

So what can you do? It's times like these that you can fall back on Design Principles to look for clues. Remember, our first principle deals with change and guides us to *interface segregation, separate interfaces, and styles to reuse.*

Remember that designs should be "open for extension but closed for modification" – see Chapter 1 for a review.



How might you take all the parts of your application to 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;
}

```

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Clearly, dealing with *i* 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

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.

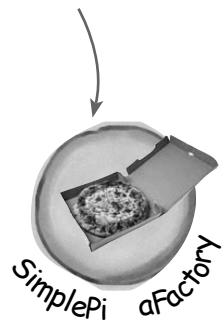
```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

First we pull the object creation code out of the `orderPizza` Method

What's going to go here?

```
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();  
}
```

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.



### e've got a n a e for this new o ect 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, which 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...

```

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 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.

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.

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

**A:**

There are no Dumb Questions

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

**A:**

you are here ▶

## 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

```
Now we give PizzaStore a reference
to a SimplePizzaFactory.
```

```
public class PizzaStore {
    SimplePizzaFactory factory;

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

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

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

    // other methods here
}
```

PizzaStore gets the factory passed to it in the constructor.

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

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



We now take advantage of composition to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to do that in the PizzaStore? What factory implementation might we be able to swap in and out?

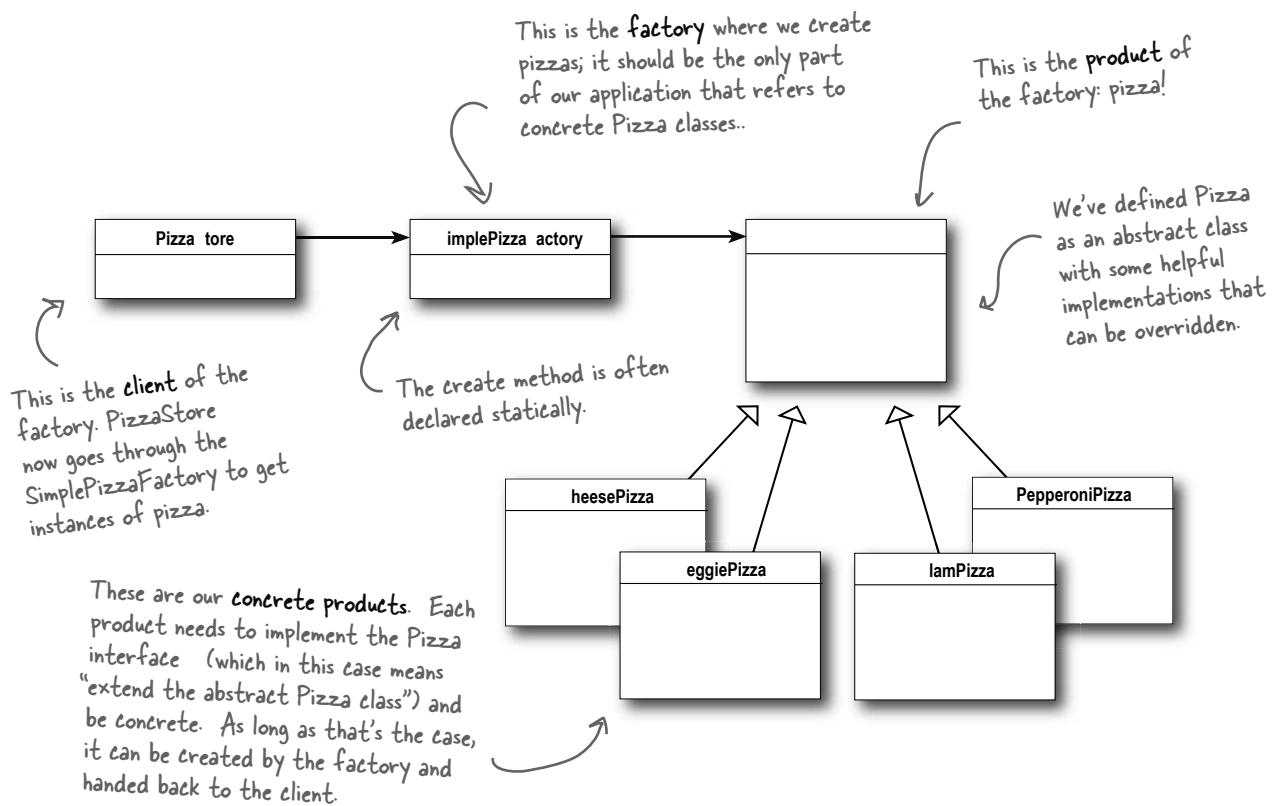
We don't have a factory, we're still in New York, California, or Tokyo, so let's not get New Haven, too)

# The Simple Factory defined

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 **L** pattern doesn't mean we shouldn't check out how it's put together. Let's take a look at the class diagram of our new Pizza store



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

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

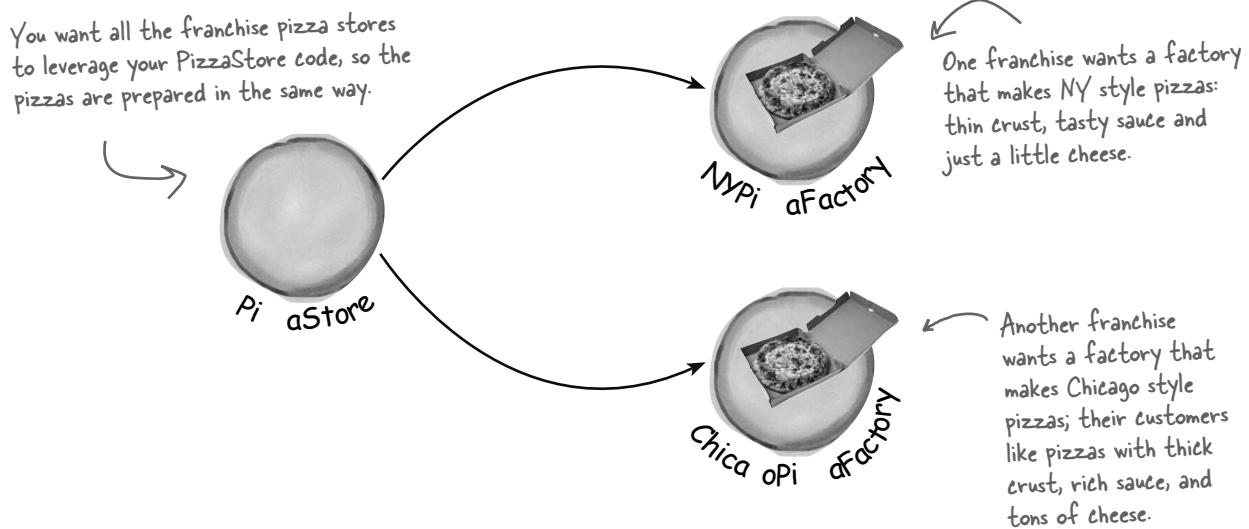
you are here ▶



## Franchising the pizza store

our 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 pizza (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 `SimplePi aFactory` and create three different factories, `NYPi aFactory`, `ChicagoPi aFactory` and `CaliforniaPi aFactory`, then we can just compose the `PizzaStore` with the appropriate factory and a franchise is good to go. That's one approach.

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

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");
```

Here we create a factory for making NY style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-styled pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago flavored ones

## But you'd like a little more quality control...

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.

Thinking 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 Pizza store, but it wasn't flexible. So, how can we have our pizza and eat it too?

I've been making pizza for years so I thought I'd add my own improvements to the PizzaStore procedures...



Not what you want in a good franchise. You do NOT want to know what he puts on his pizzas.

## A framework for the pizza store

here 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.

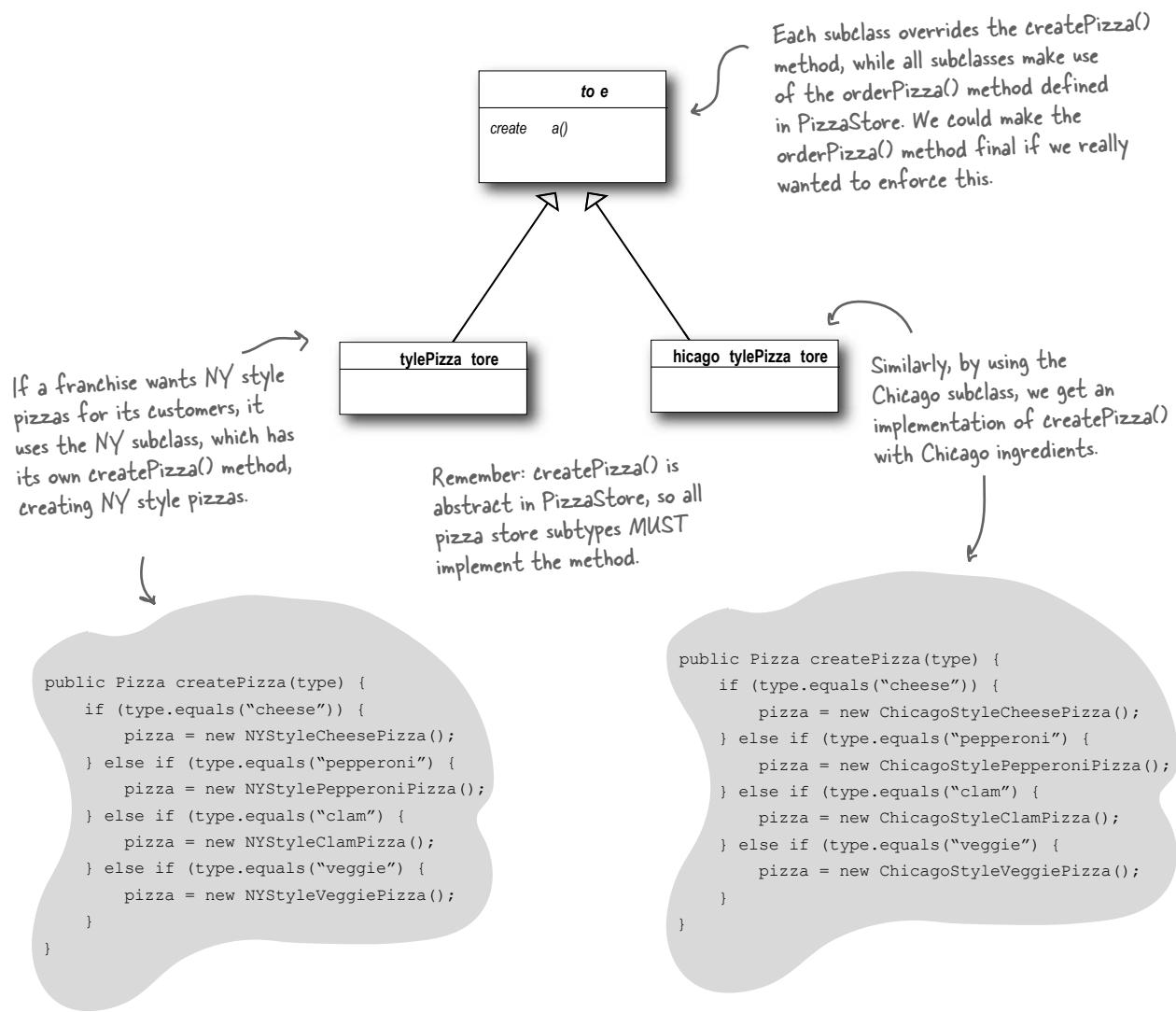
Our "factory method" is now abstract in PizzaStore.

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

## Allowing the subclasses to decide

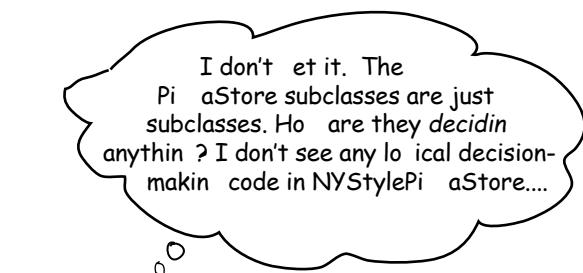
Remember, the Pizza store already has a well-honed order system in the `orderPizza()` method and you want to ensure that it's consistent across all franchises.

What varies among the regional Pizza stores is the style of pizza as they make New York Pizza has thin crust, Chicago Pizza has thick, and so on and we are going to push all these variations into the `createPizza()` method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of Pizza store define what the `createPizza()` method looks like. So, we will have a number of concrete subclasses of Pizza store, each with its own pizza variations, all fitting within the Pizza store framework and still making use of the well-tuned `orderPizza()` method.

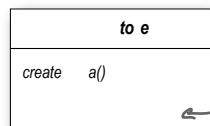


you are here ▶

ho do u a e de ide

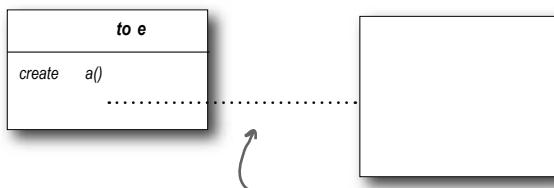


Well, think about it from the point of view of the PizzaStore's `orderPizza()` method: it is defined in the abstract `PizzaStore`, but concrete types are only created in the subclasses.



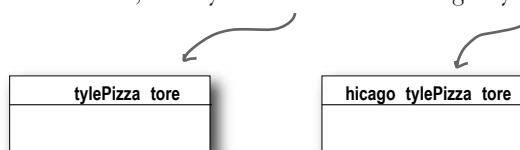
`orderPizza()` is defined in the abstract `PizzaStore`, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.

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



`orderPizza()` calls `createPizza()` to actually get a pizza object. But which kind of pizza will it get? The `orderPizza()` method can't decide; it doesn't know how. So who does decide?

When `orderPizza()` calls `createPizza()`, one of your subclasses will be called into action to create a `Pizza`. Which kind of `Pizza` will be made? Well, that's decided by the choice of `PizzaStore` 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.

ha ter

## Let's make a PizzaStore

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

Here's the New York regional style

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

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

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

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

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

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

Once we've got our Pizza store 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

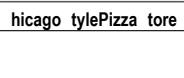
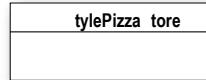
We've knocked out the Pi a tote, just two more to go and we'll be ready to franchise  
Write the Chicago and California Pi a tote implementations here

## 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.

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    protected abstract Pizza createPizza(String type);
    // other methods here
}
```

The subclasses of `PizzaStore` handle object instantiation for us in the `createPizza()` method.



All the responsibility for instantiating Pizzas has been moved into a method that acts as a factory.



### Code Up Close

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.

`abstract Product factoryMethod(String type)`

A factory method is abstract so the subclasses are counted on to handle object creation.

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method may be parameterized (or not) to select among several variations of a product.

A factory method isolates the client (the code in the superclass, like `orderPizza()`) from knowing what kind of concrete Product is actually created.

## et's see how it works: ordering pizzas with the pizza factory method



## So how do they order?

- ❶ First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a New York PizzaStore.
- ❷ 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).
- ❸ To create the pizzas, the `createPizza()` method is called, which is defined in the two subclasses PizzaStore and ChicagoPizzaStore. As we defined them, the New York PizzaStore instantiates a New York style pizza, and the ChicagoPizzaStore instantiates a Chicago style pizza. In either case, the pizza is returned to the `orderPizza()` method.
- ❹ 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 serves it for Ethan and Joel.

et's check out how these pizzas are really made to order...

e in  
t e S enes



1

**et's follow than's order rst we need a N Pi a tore**

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates a instance of  
NYPizzaStore.

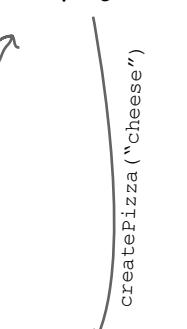


2

**Now that we have a store, we can ta e an order**

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on  
the nyPizzaStore instance (the method  
defined inside PizzaStore runs).



3

**he orderPi a etod then calls the createPi a etod**

```
Pizza pizza = createPizza("cheese");
```

Remember, createPizza(), the factory  
method, is implemented in the subclass. In  
this case it returns a NY Cheese Pizza.



4

**Finally we have the unprepared pi a in hand and the  
orderPi a etod nishes preparing it**

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

The orderPizza() method gets  
back a Pizza, without knowing  
exactly what concrete class it is.

All of these methods are defined  
in the specific pizza returned  
from the factory method  
createPizza(), defined in the  
NYPizzaStore.

you are here ▶

## e ej st ss et

O at est t be e la  
t ts e ass lets le e tte



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

```
public abstract class Pizza {  
    String name;  
    String dough;  
    String sauce;  
    ArrayList toppings = new ArrayList();  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        System.out.println("Tossing dough...");  
        System.out.println("Adding sauce...");  
        System.out.println("Adding toppings: ");  
        for (int i = 0; i < toppings.size(); i++) {  
            System.out.println("    " + toppings.get(i));  
        }  
    }  
  
    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;  
    }  
}
```

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.

REMEMBER: we don't provide import and package statements in the  
code listings. Get the complete source code from the headfirstlabs  
web site. 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?

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

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

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

And one topping, reggiano cheese!

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

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

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

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

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

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

# You've waited long enough, time for some pizzas

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

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

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

First we create two different stores.

Then use one store to make Ethan's order.

And the other for Joel's.

```
File Edit Window Help o WantMoot OnT atPi a?
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
    Grated Regiano 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

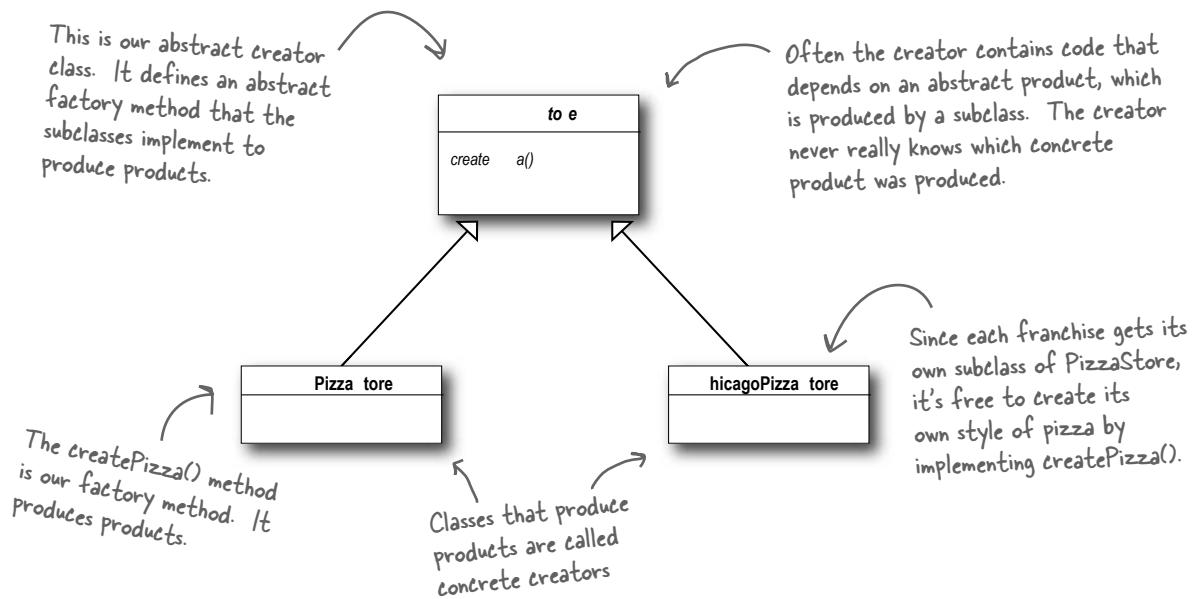
Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
    Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

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.

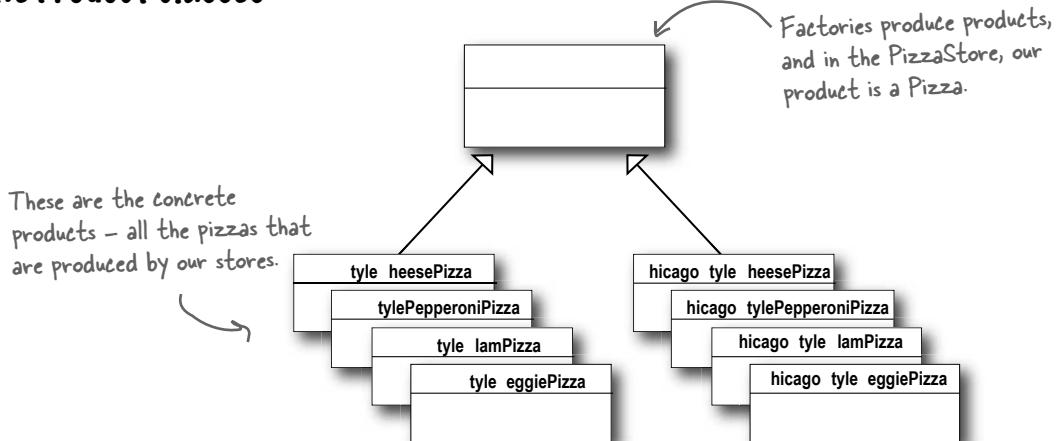
# It's finally time to meet the Factory Method Pattern

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



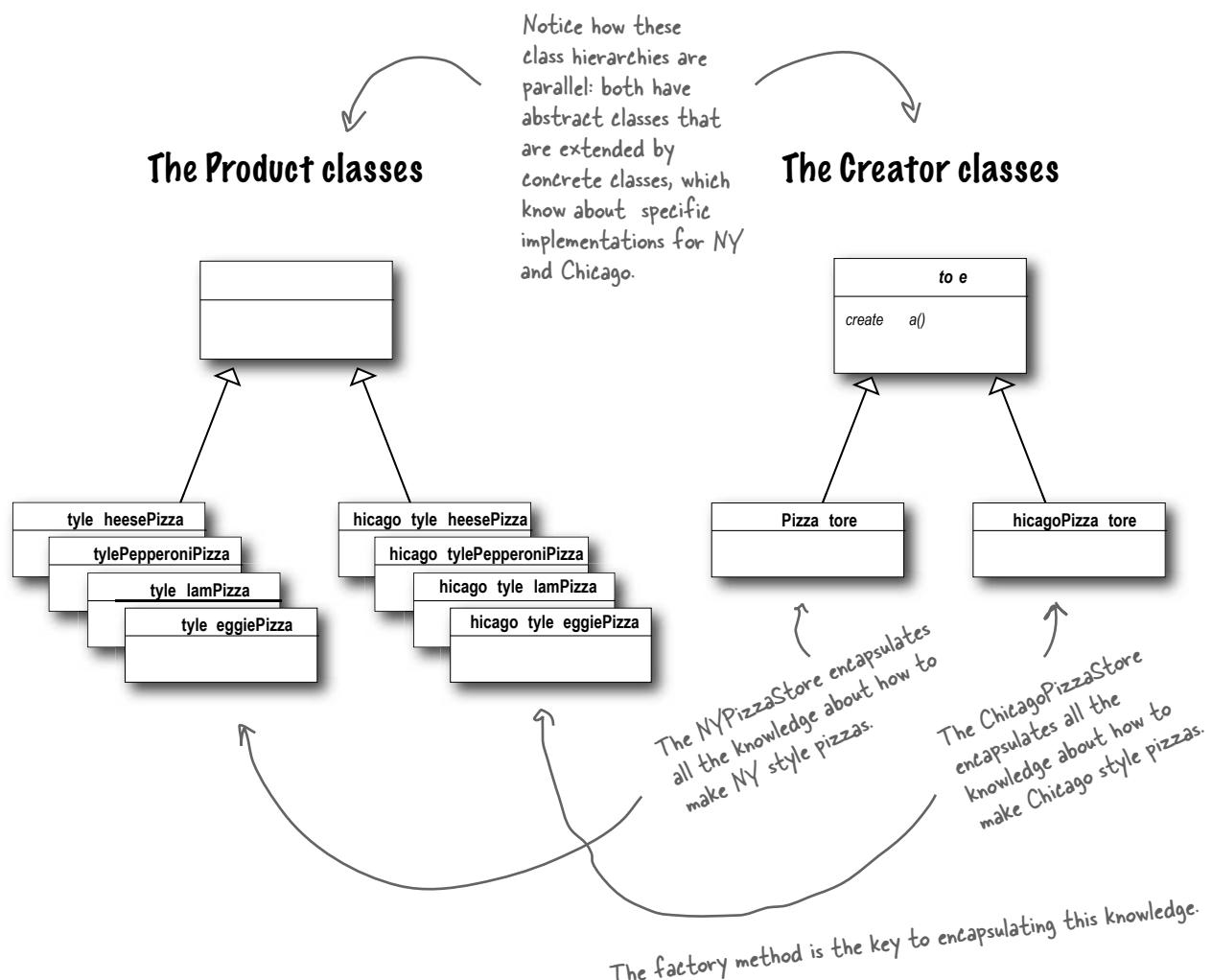
## The Product classes



## 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.

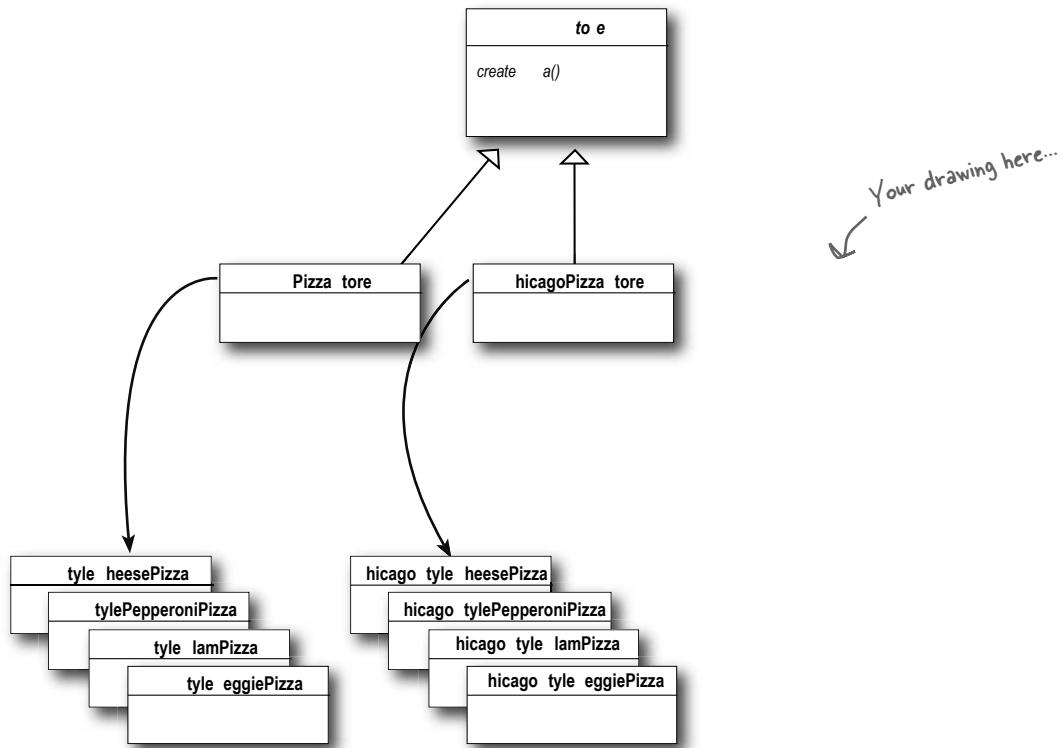
Let's look at the two parallel class hierarchies and see how they relate





# esign Pu le

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 Pizza store.



Okay, now write the five most interesting things you can think of to put on a pizza. When you're ready to go into business making pizza in California,

---



---



---



---



---

## Factory Method Pattern defined

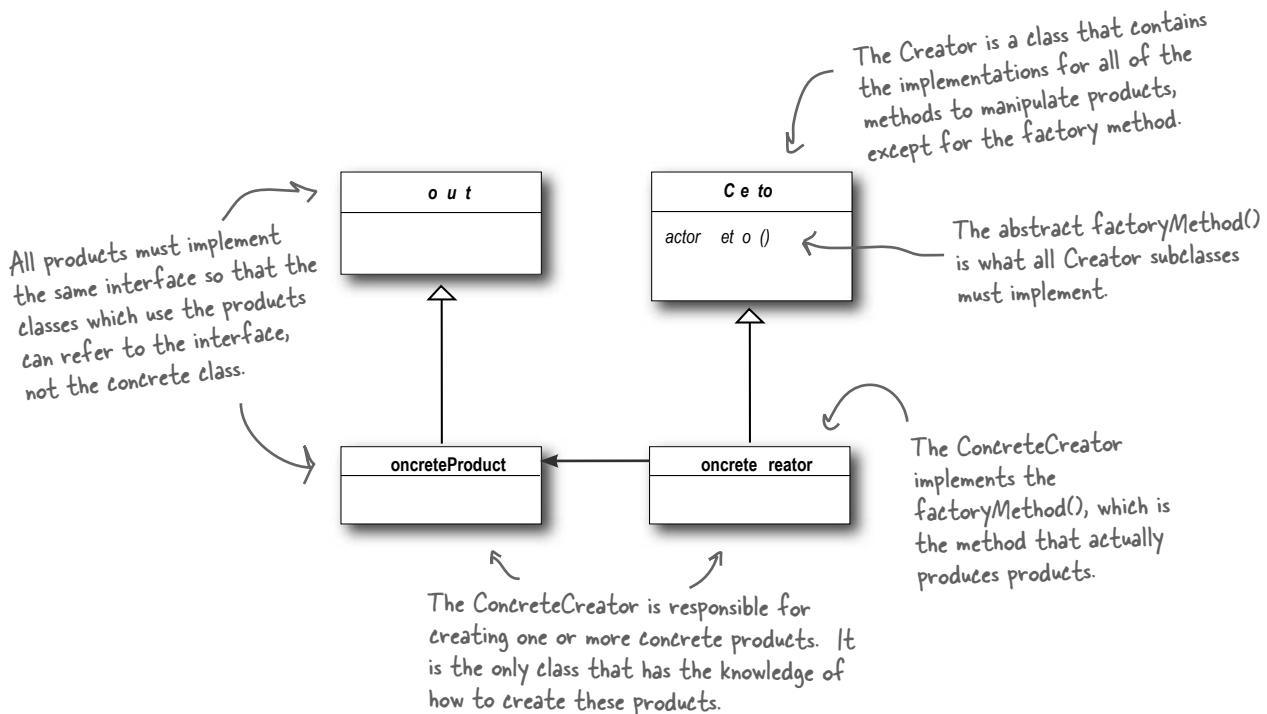
It's time to roll out the official definition of the Factory Method Pattern

**he actor method after** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

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.

Since in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" 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.

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



## Dumb Questions<sup>there are no</sup>

**Q:** What's the advantage of the factory method Pattern when you only have one concrete creator?

**A:**

**A:**

**Q:** our parameterized types don't seem type safe I'm just passing in a string! What if I asked for a calmPizza ?

**Q:** Would it be correct to say that our and chicago stores are implemented using factory? They look just like it

**A:**

**A:**

e

**Q:** I'm still a bit confused about the difference between 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?

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

**A:**

**A:**

**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 ter and tudent**

**a ter** rass o er tell e o yo rtrai i g is goi g  
**tudent** Master a eta e yst dyo e a s late at  
aries rt er

**a ter** oo

**tudent** a elear edt ato e a e a s late t e odet at  
reates o e ts e yo a e odet ati sta tiates o rete  
lasses t is is a area o re e t a ge elear ed a  
te i e alled a tories t at allo syo to e a s late t is  
e a ior o i sta tiatio

**a ter** dt ese a tories o at e e tare t ey

**tudent** ere are a y y la i gall y reatio ode i o e  
o e tor et od a oidd li atio i y ode a d ro ideo e  
la e to eror ai te a e at also ea s lie ts de e d  
o ly o i tera esrat ert a t e o rete lasses re ired to  
i sta tiate o e ts s a elear ed i yst dies t is allo s e  
to rogra to a i tera e ota i le e tatio a dt at a es  
y ode ore e i lea de te si le i t e t re

**a ter** es rass o er yo r i sti ts are gro i g o  
yo a e a y estio s or yo r aster today

**tudent** Master o t at ye a s lati go e t reatio  
a odi g to a stra tio sa dde o li g y lie t ode ro  
a t al i le e tatio s t y a tory ode st still se  
o rete lasses to i sta tiate real o e ts ot lli gt e  
ool o er yo eyes

**a ter** rass o er o e t reatio is a reality o lie e st  
reate o e ts or e ill e er reate a si gle a a rogra t  
it o ledge o t is reality e a desig o r odesot at e  
a e orralled t is reatio ode li e t e s ee ose ool yo  
o ld ll o eryo reyes e orralled e a rote ta d  
are ort e reatio ode e let o r reatio ode r ild  
t e e ill e er olle tits ool

**tudent** Master see t e tr t i t is

**a ter** s e yo o ld o lease go a d editate o  
o e tde e de ies

# A very dependent PizzaStore



Let's pretend you've never heard of an **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:

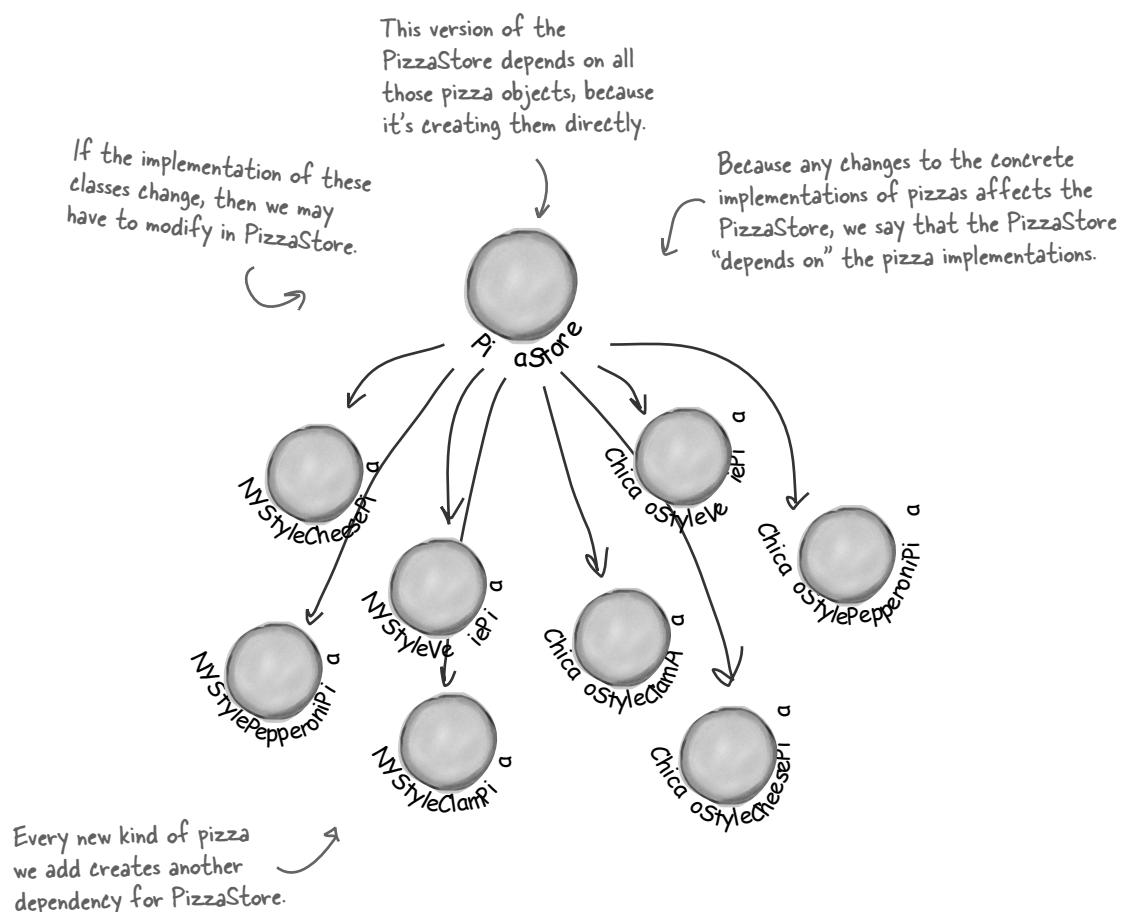
number

number with California too

## ooking at object dependencies

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

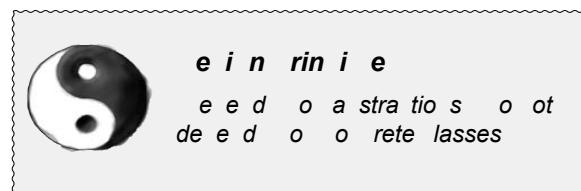
If we draw a diagram representing that version of the `PizzaStore` and all the objects it depends on, here's what it looks like



# The Dependency Inversion Principle

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

Here's the general principle



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

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 → depend on abstractions.

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...

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, prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

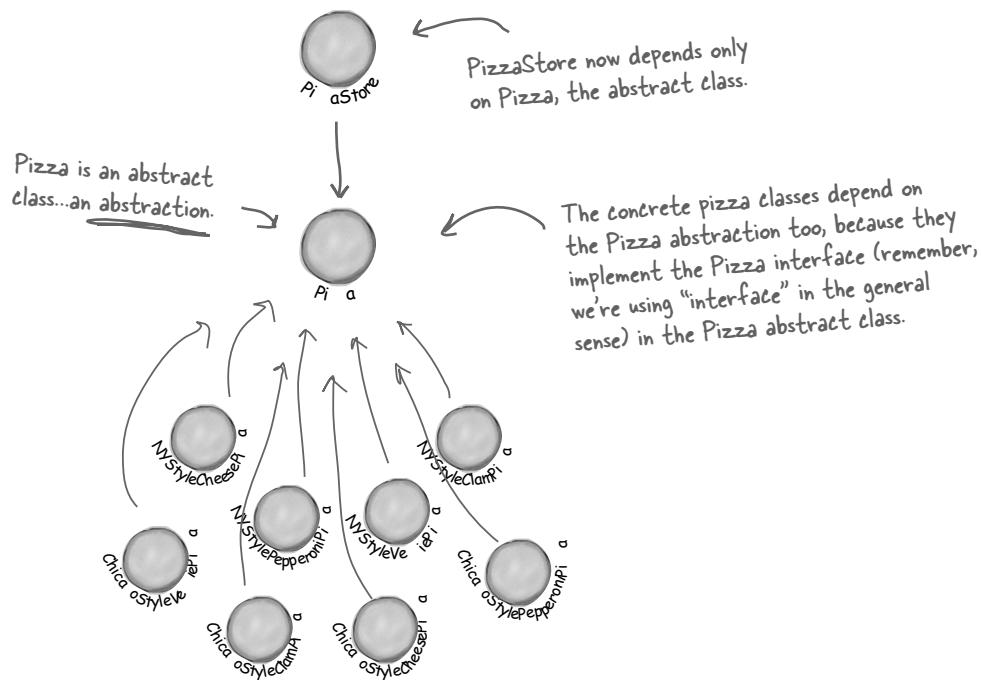
## Applying the Principle

ow, the main problem with the `orderPi a` tore is that it depends on every type of `Pi a` because it actually instantiates concrete types in its `orderPi a()` method.

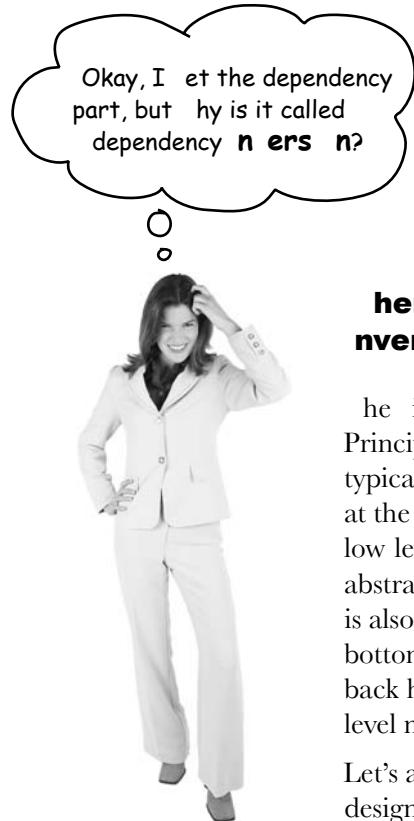
While we've created an abstraction, `Pi a`, we're nevertheless creating concrete `Pi a`s in this code, so we don't get a lot of leverage out of this abstraction.

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

o, after we've applied the Factory Method, our diagram looks like this



fter applying the Factory Method, you'll notice that our high level component, the `Pi a` tore, and our low level components, the `pi as`, both depend on `Pi a`, 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.

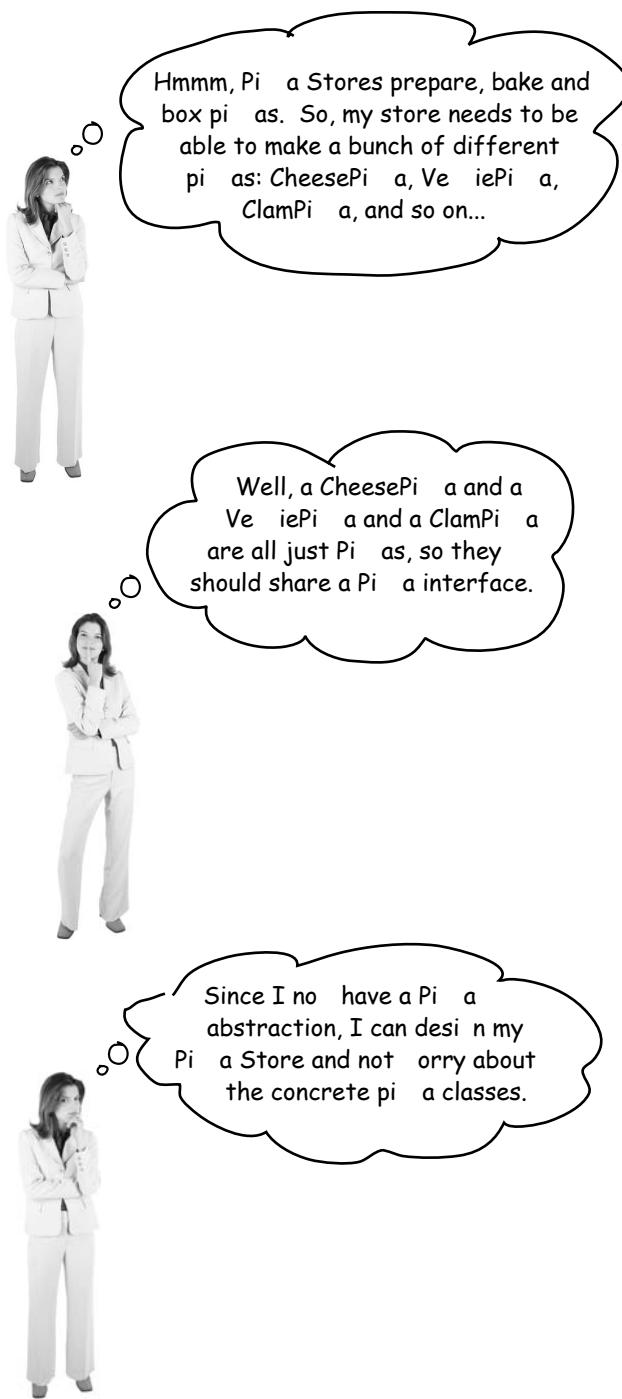


### **here's the inversion in Dependency Inversion Principle?**

The inversion in the name dependency inversion Principle is there because it inverts the way you typically might think about your 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...



kay, so you need to implement a Pi a tore. What's the first thought that pops into your head?

ight, you start at top and follow things down to the concrete classes. But, as you've seen, you don't want your store to know about the concrete pi a types, because then it'll be dependent on all those concrete classes

ow, let's invert your thinking.. instead of starting at the top, start at the Pi as and think about what you can abstract.

ight you are thinking about the abstraction i . o now, go back and think about the design of the Pi a tore again.

Close. But to do that you'll have to rely on a factory to get those concrete classes out of your Pi a tore. Once you've done that, your different concrete pi a types depend only on an abstraction and so does your store. We've taken a design where the store depended on concrete classes and inverted those dependencies (along with your thinking).

# A few guidelines to help you follow the Principle...

The following guidelines can help you avoid dependency inversion Principle

- A variable should hold a reference to a concrete class.
- A class should derive from a concrete class.
- A method should override an implemented method of any of its base classes.

designs that violate

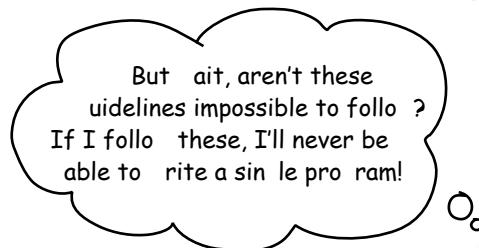
If you use new, you'll be holding a reference to a concrete class. Use a factory to get around that!



If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.



If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.



You're exactly right. Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines.

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate strings objects all the time without thinking twice.

Does that violate the principle? No. Is that okay? Yes. Why? Because strings are very unlikely to change.

If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.



a i ie o in redient

## Meanwhile, back at the Pizza Store...

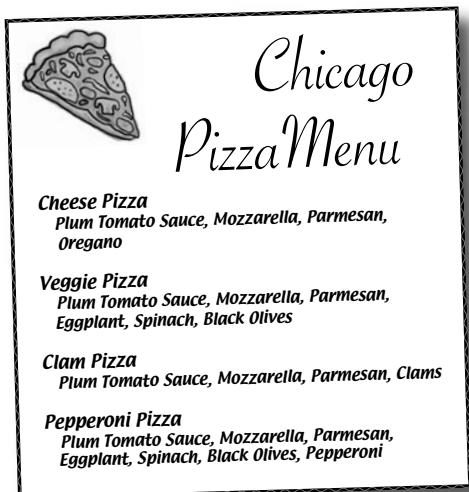
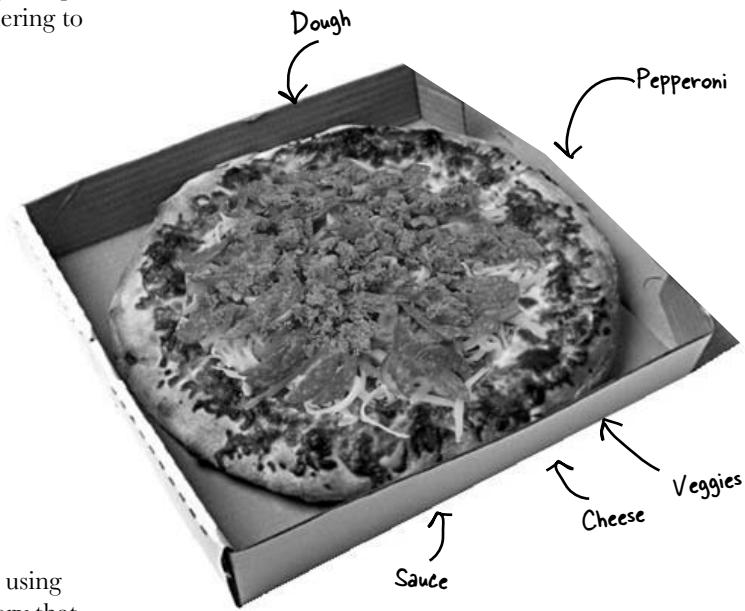
The design for the Pizza store is really shaping up. It's got a flexible framework and it does a good job of adhering to design principles.

Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *rules*, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand.

### Ensuring consistency in your ingredients

How are you going to ensure each franchise is using quality ingredients? You're going to build a factory that produces them and ships them to your franchises.

Now there is only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. You have one set of ingredients that need to be shipped to New York and another set that needs to be shipped to Chicago. Let's take a closer look.



**Chicago Pizza Menu**

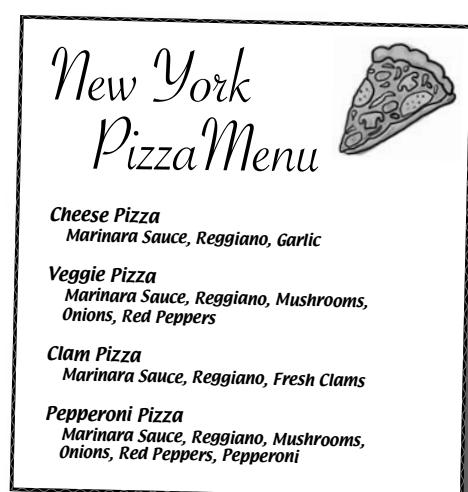
**Cheese Pizza**  
Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

**Veggie Pizza**  
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives

**Clam Pizza**  
Plum Tomato Sauce, Mozzarella, Parmesan, Clams

**Pepperoni Pizza**  
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.



**New York Pizza Menu**

**Cheese Pizza**  
Marinara Sauce, Reggiano, Garlic

**Veggie Pizza**  
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

**Clam Pizza**  
Marinara Sauce, Reggiano, Fresh Clams

**Pepperoni Pizza**  
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

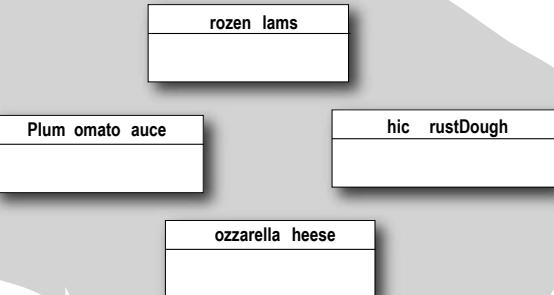
ha ter

## Families of ingredients...

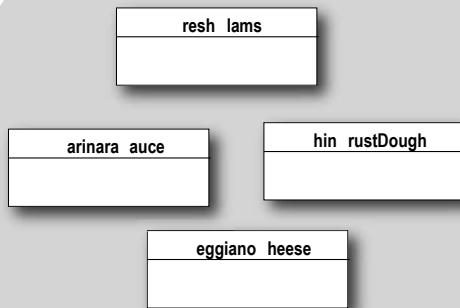
New or uses one set of ingredients and hicago another given the popularity of ectville. Pi a it won't e long efore you also need to ship another set of regional ingredients to alifornia, and what's ne t? eattle?

For this to wor , you are going to have to gure out how to handle fa ilies of ingredients

### Chicago



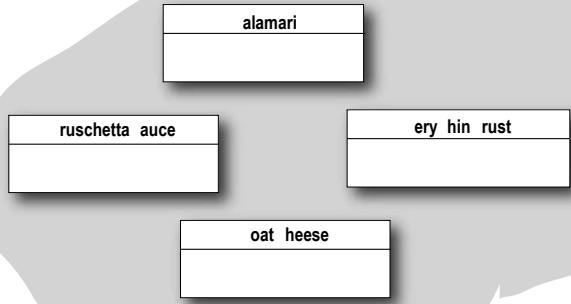
### New York



Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

All Objectville's Pizzas are made from the same components, but each region has a different implementation of those components.

### California



In total, these three regions make up ingredient families, with each region implementing a complete family of ingredients.

*you are here ▶*

## Building the ingredient factories

**N**ow we're going to build a factory to create our ingredients the factory will be responsible for creating each ingredient in the ingredient family in other words, the factory will need to create dough, sauce, cheese, and so on you'll see how we are going to handle the regional differences shortly

**L**et's start by defining an interface for the factory that is going to create all our ingredients

```
public interface PizzaIngredientFactory {  
    public Dough createDough();  
    public Sauce createSauce();  
    public Cheese createCheese();  
    public Veggies[] createVeggies();  
    public Pepperoni createPepperoni();  
    public Clams createClam();  
}
```

Lots of new classes here,  
one per ingredient.



For each ingredient we define a  
create method in our interface.

If we'd had some common "machinery"  
to implement in each instance of  
factory, we could have made this an  
abstract class instead...

### Here's what we're going to do

- ➊ Build a factory for each region. To do this, you'll create a subclass of `PizzaIngredientFactory` that implements each create method
- ➋ Implement a set of ingredient classes to be used with the factory, like `eggianoCheese`, `redPeppers`, and `hickCrustDough`. These classes can be shared among regions where appropriate.
- ➌ Then we still need to hook all this up by working our new ingredient factories into our old `Pizza` code.

# Building the New York ingredient factory

**ay, here's the implementation for  
the New York ingredient factory. His  
factory specializes in marinara sauce,  
reggiano cheese, fresh clams.**

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FreshClams();
    }
}
```

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

The NY ingredient factory implements the interface for all ingredient factories

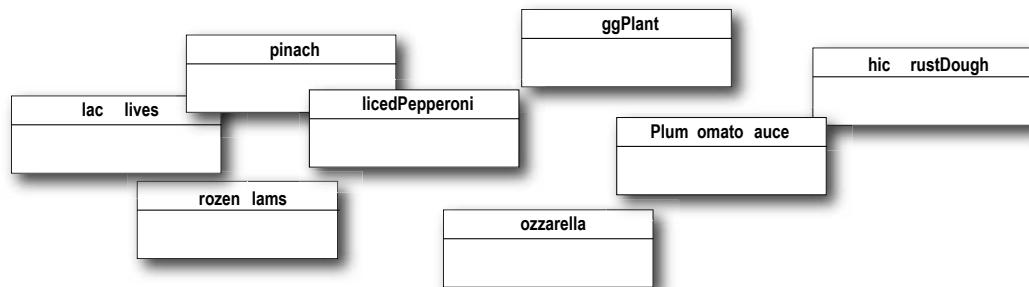
For each ingredient in the ingredient family, we create the New York version.

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

## Sharpen your pencil

Write the ChicagoPizzaIngredientFactory. You can reference the classes below in your implementation



## Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizza as so they only use factory produced ingredients. We'll start with our abstract Pizza class

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

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

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

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

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

de ou in in redient

## Reworking the pizzas, continued...

ow that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas; only this time around they will get their ingredients straight from the factory. The franchisees' days of skimping on ingredients are over.

When we wrote the Factory Method code, we had a CheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough, sauce, cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the Cheese Pizza:

```
public class CheesePizza extends Pizza {  
    PizzaIngredientFactory ingredientFactory;  
  
    public CheesePizza(PizzaIngredientFactory ingredientFactory)  
        this.ingredientFactory = ingredientFactory;  
    }  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        dough = ingredientFactory.createDough();  
        sauce = ingredientFactory.createSauce();  
        cheese = ingredientFactory.createCheese();  
    }  
}
```



To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

← Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

ha ter



## Code Up Close

The Pizza class uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizza as it is. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Rockies, the Pacific Northwest, and beyond.

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Let's check out the ClamPizza as well

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

If it's a New York factory, the clams will be fresh; if it's Chicago, they'll be frozen.

ClamPizza also stashes an ingredient factory.

To make a clam pizza, the prepare method collects the right ingredients from its local factory.

use the right ingredient factory

## Revisiting our pizza stores

We're almost there we just need to add a quick trip to our franchise stores to make sure they are using the correct Pizza as well as also need to give them a reference to their local ingredient factories

```
public class NYPizzaStore extends PizzaStore {  
  
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;  
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory();  
  
        if (item.equals("cheese")) {  
  
            pizza = new CheesePizza(ingredientFactory);  
            pizza.setName("New York Style Cheese Pizza");  
  
        } else if (item.equals("veggie")) {  
  
            pizza = new VeggiePizza(ingredientFactory);  
            pizza.setName("New York Style Veggie Pizza");  
  
        } else if (item.equals("clam")) {  
  
            pizza = new ClamPizza(ingredientFactory);  
            pizza.setName("New York Style Clam Pizza");  
  
        } else if (item.equals("pepperoni")) {  
            pizza = new PepperoniPizza(ingredientFactory);  
            pizza.setName("New York Style Pepperoni Pizza");  
  
        }  
        return pizza;  
    }  
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.



Compare this implementation of the createPizza() method to the one in the Factory Method implementation earlier in this chapter.

# What have we done?

**hat was quite a series of code changes what exactly did we do?**

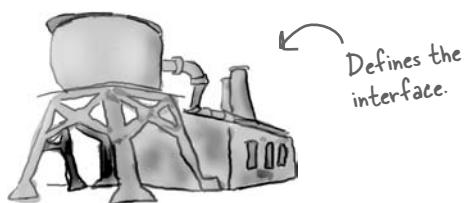
**e provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory**

**n Abstract Factory gives us an interface for creating a family of products by writing code that uses this interface, we decouple our code from the actual factory that creates the products**

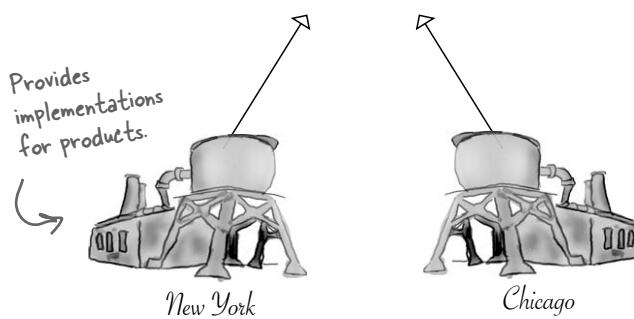
**hat allows us to implement a variety of factories that produce products meant for different contexts such as different regions, different operating systems, or different looks and feels**

**ecause our code is decoupled from the actual products, we can substitute different factories to get different behaviors like getting marinara instead of plain tomatoe**

**n abstract actor provides an interface for a family of products. That's a family. In our case it's all the things we need to make a pizza: dough, sauce, cheese, meats and veggies.**



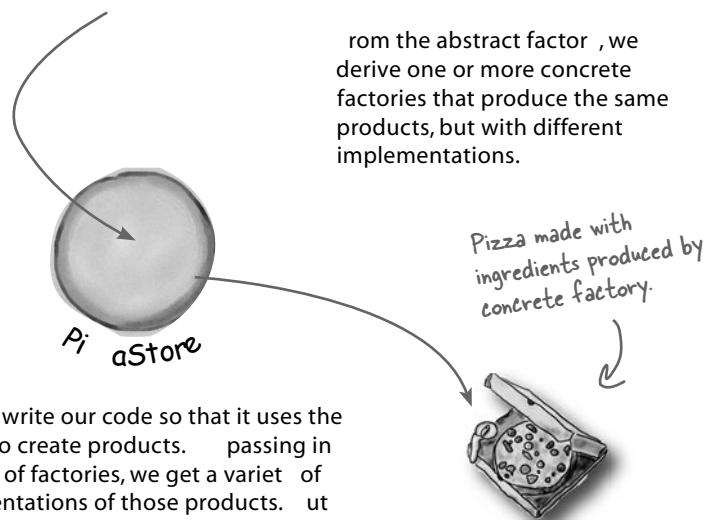
Objectville Abstract Ingredient Factory



New York

Chicago

**From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.**



**e then write our code so that it uses the factor to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.**

order o e ore i a

## More pizza for Ethan and Oel...

Ethan and Oel can't get enough Objectville Pizza. What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...



The first part of the order process hasn't changed at all. Let's follow Ethan's order again:

### 1 First we need a N Y Pi a store

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates an instance of NYPizzaStore.

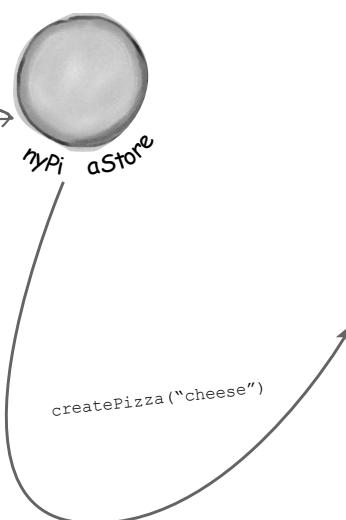
### 2 Now that we have a store, we can take an order

```
nyPizzaStore.orderPizza("cheese");
```

the orderPizza() method is called on the nyPizzaStore instance.

### 3 The orderPizza method first calls the createPizza method

```
Pizza pizza = createPizza("cheese");
```



ha ter

From here things change, because we are using an ingredient factory

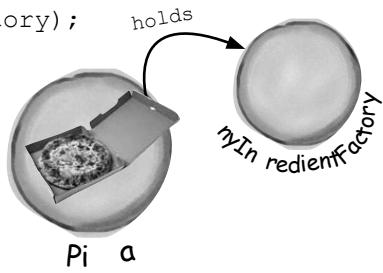


- When the `createPi` method is called, that's when our ingredient factory gets involved

The ingredient factory is chosen and instantiated in the `PizzaStore` and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(newIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.



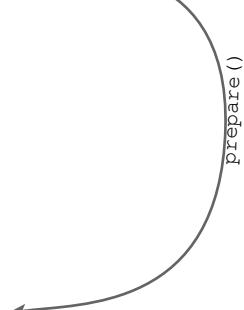
Pi  
za

- Next we need to prepare the pizza once the `prepare` method is called, the factory is asked to prepare ingredients

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Thin crust  
Marinara  
Reggiano

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.



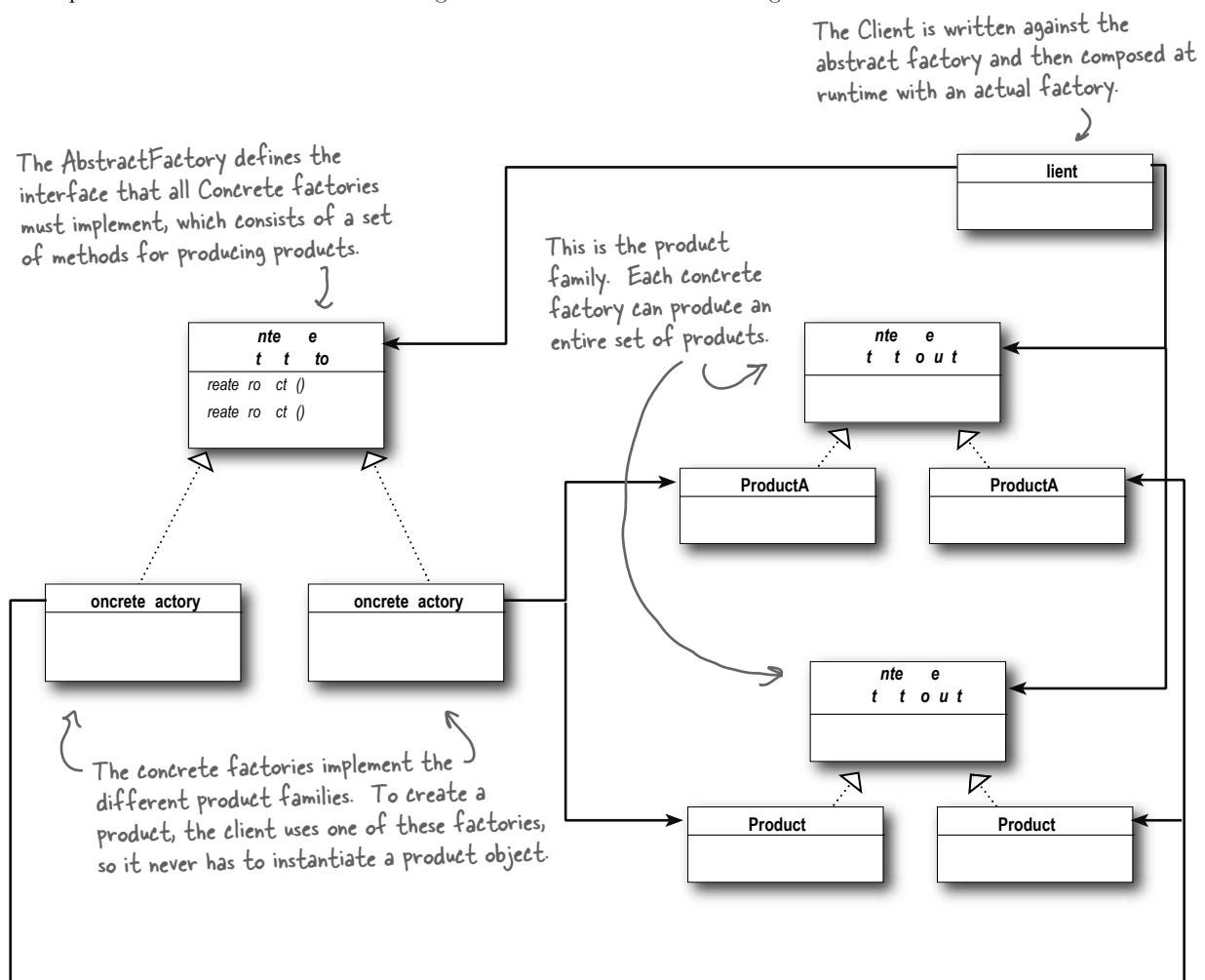
- Finally we have the prepared pizza in hand and the `orderPizza` method makes, cuts, and serves the pizza

# Abstract Factory Pattern defined

We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern

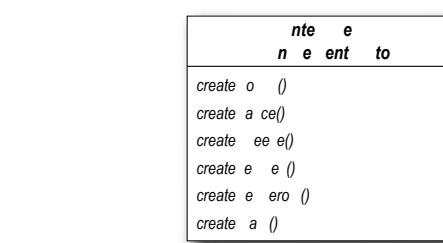
**the abstract actor patter** provides an interface for creating families of related or dependent objects without specifying their concrete classes.

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together

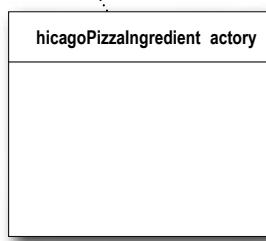
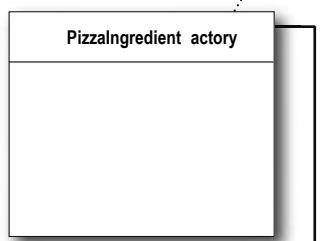


## That's a fairly complicated class diagram let's look at it all in terms of our Pizza store

The abstract `PizzaIngredientFactory` is the interface that defines how to make a family of related products – everything we need to make a pizza.



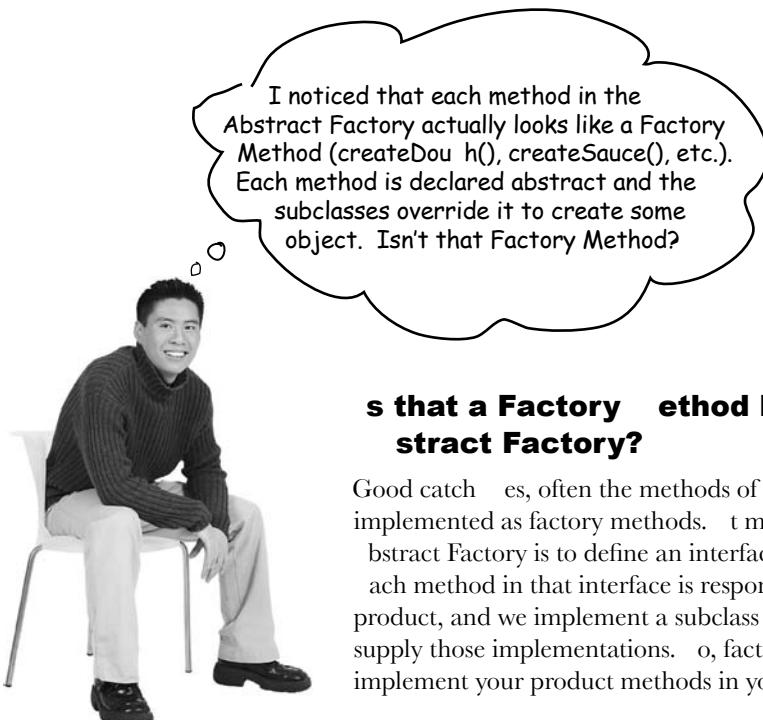
The clients of the Abstract Factory are the concrete instances of the Pizza abstract class.



The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.

Each factory produces a different implementation for the family of products.

*you are here ▶*



### **s that a Factory method lur king inside the abstract Factory?**

Good catch! Often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.



Wow, an interview with two patterns at once – this is a first for us.

Yeah, I'm not so sure – like being lumped in with Abstract Factory, you know. Just because we're both factory patterns doesn't mean we shouldn't get our own interviews.

Don't be miffed, we wanted to interview you together so we could help clear up any confusion about who's who for the readers. You do have similarities, and I've heard that people sometimes get you confused.

It is true, there have been times I've been mistaken for Factory Method, and I know you've had similar issues, Factory Method. We're both really good at decoupling applications from specific implementations; we just do it in different ways. So I can see why people might sometimes get us confused.

Well, it still ticks me off. After all, I use classes to create and you use objects; that's totally different.

Can you explain more about that, Factory Method?

ure. Both abstract Factory and create objects that's our jobs. But do it through inheritance...

...and do it through object composition.

ight. So that means, to create objects using Factory Method, you need to extend a class and override a factory method.

nd that factory method does what?

t creates objects, of course mean, the whole point of the Factory Method Pattern is that you're using a subclass to do your creation for you. In that way, clients only need to know the abstract type they are using, the subclass worries about the concrete type. So, in other words, keep clients decoupled from the concrete types.

nd do too, only do it in a different way.

Go on, abstract Factory... you said something about object composition?

provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type. So, like Factory Method, my clients are decoupled from the actual concrete products they use.

h, see, so another advantage is that you group together a set of related products.

hat's right.

What happens if you need to extend that set of related products, to say add another one? Doesn't that require changing your interface?

hat's true; my interface has to change if new products are added, which know people don't like to do....

snicker

What are you snickering at, Factory Method?

h, come on, that's a big deal. Changing your interface means you have to go in and change the interface of every subclass. That sounds like a lot of work.

eah, but need a big interface because am used to create entire families of products. ou're only creating one product, so you don't really need a big interface, you just need one method.

bstract Factory, heard that you often use factory methods to implement your concrete factories?

es, 'll admit it, my concrete factories often implement a factory method to create their products. In my case, they are used purely to create products...

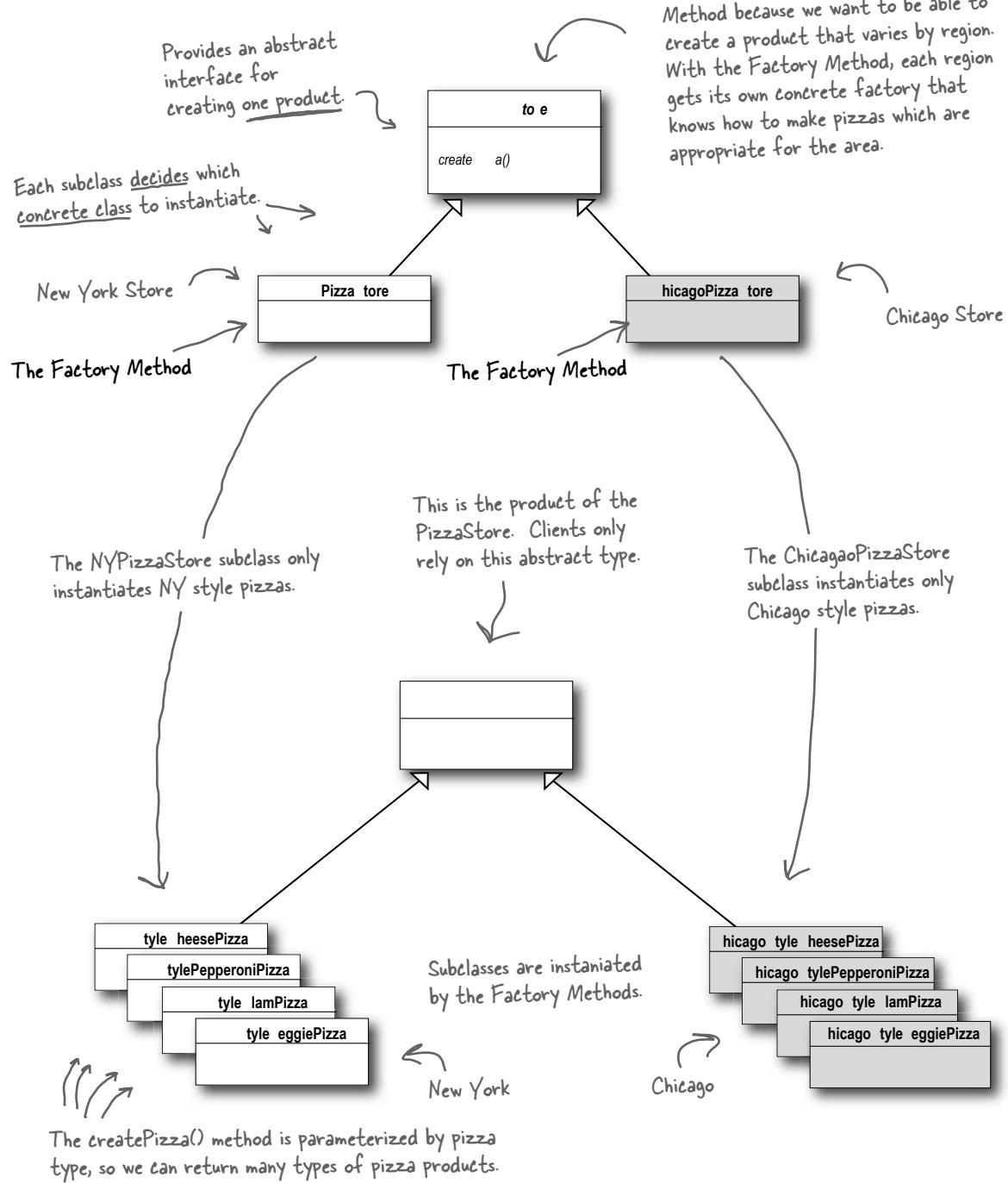
...while in my case usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

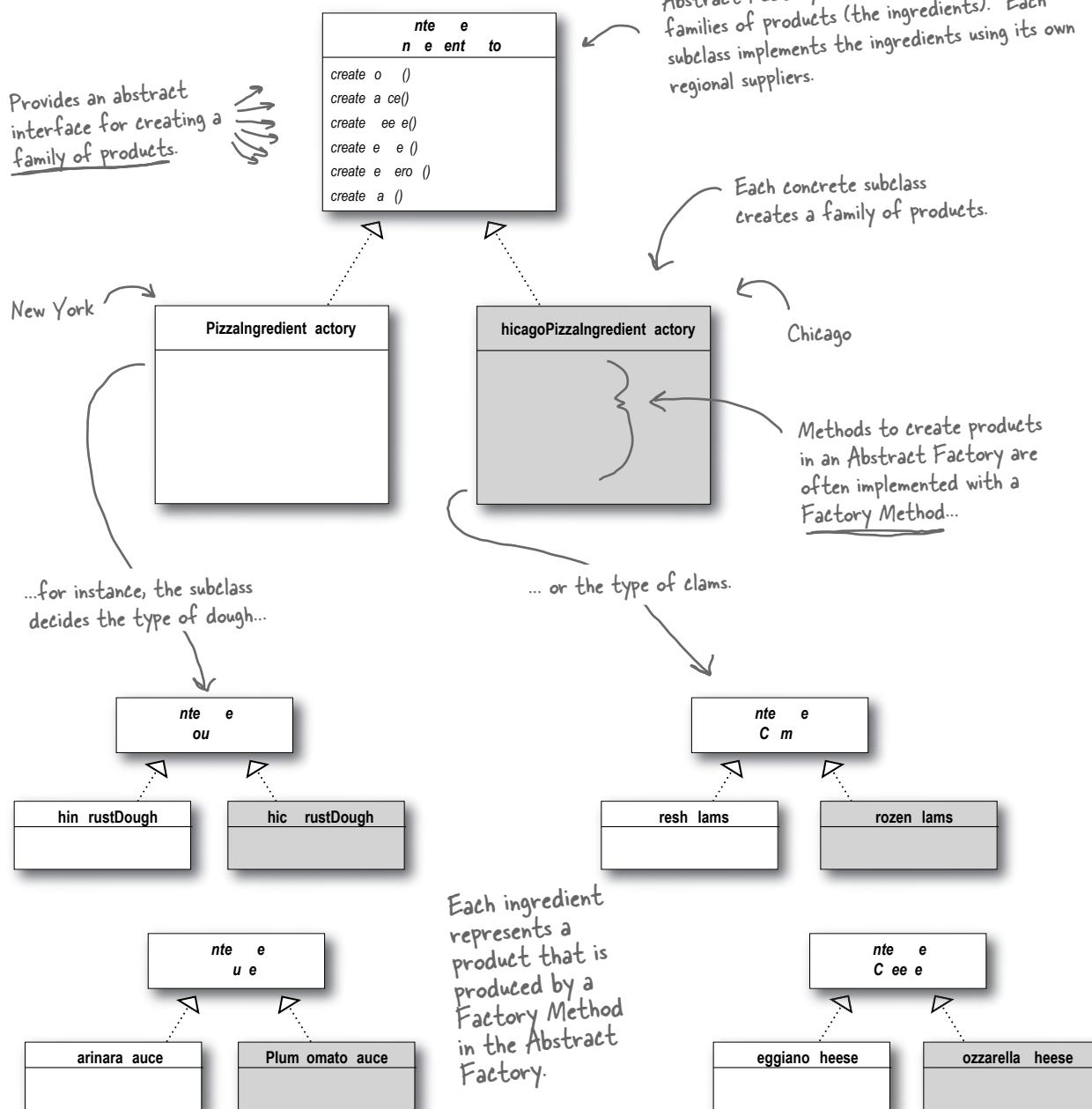
t sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. ou both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, which is really great, whether you're using Factory Method or bstract Factory. May allow you each a parting word?

anks. emember me, bstract Factory, and use me whenever you have families of products you need to create and you want to make sure your clients create products that belong together.

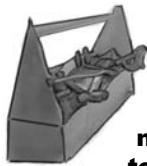
nd I'm Factory Method; use me to decouple your client code from the concrete classes you need to instantiate, or if you don't know ahead of time all the concrete classes you are going to need. So use me, just subclass me and implement my factory method

# Factory Method and Abstract Factory compared



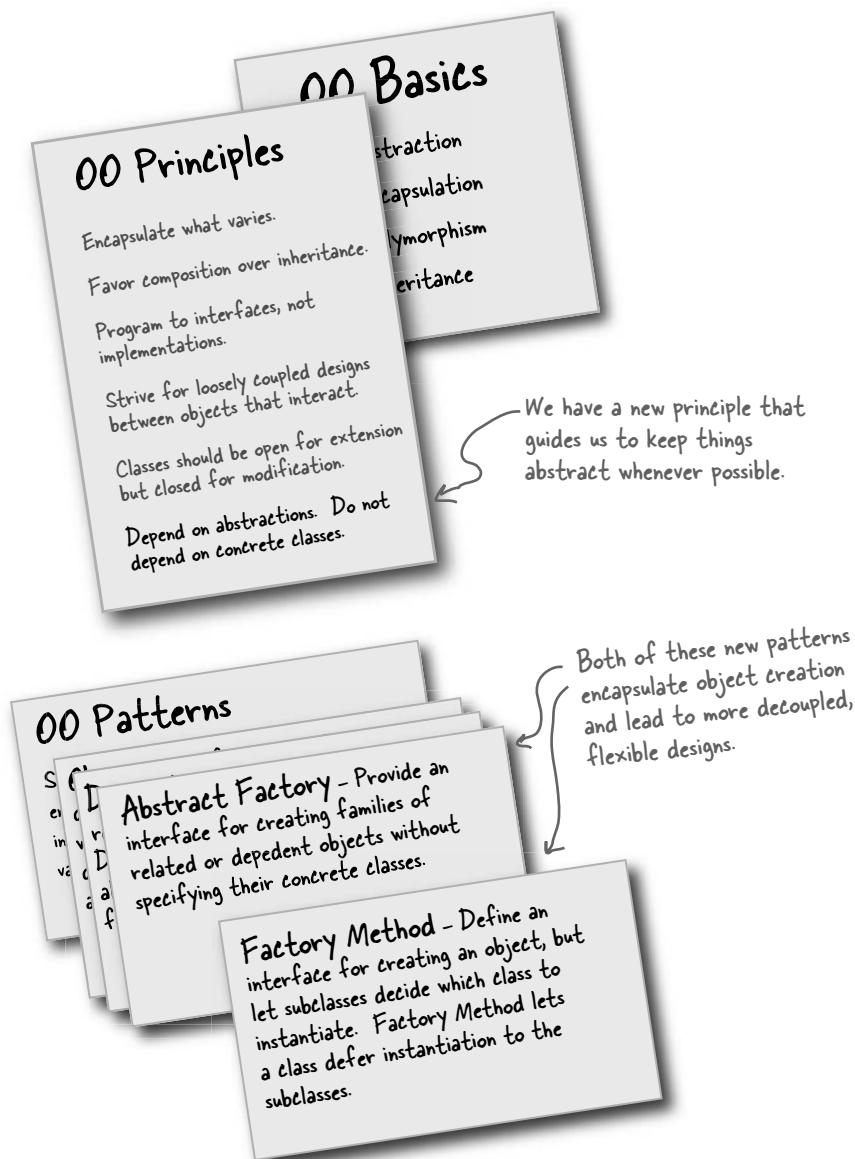


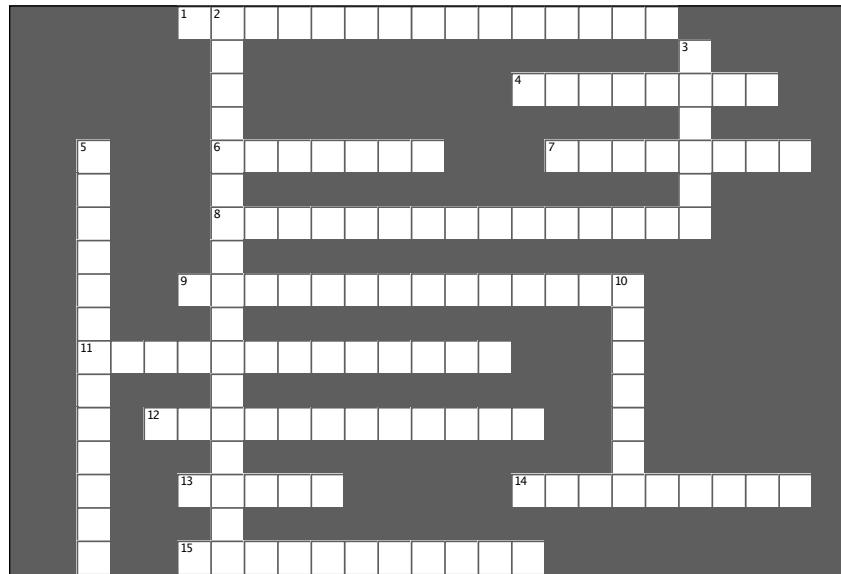
The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.



# Tools for your Design Toolbox

In this chapter, we added two more tools to your tool belt: Factory method and abstract Factory both patterns encapsulate object creation and allow you to decouple your code from concrete types.





#\* \$\$

```
%#2J 1=?CNR 5 AOK@A=?D B@J?DNA EN=
<<<<<<<<<<<
(#2J 1=?CNR 5 AOK@ QDK @?EAN QD2D ?HNN
OK ENGJ@E@A.
* #8 KIA KB7 ESS=9 CMA 日 1=?CNR 5 AOK@?=COM
+##/ HIB AQ ; KMG9 CMA 7 ESS=N PNA OEN GE@KB
?DAANA
, #2J / >ND@?O1=?CNR A=?D EN G@J CB=?CNR EN
= <<<<<<<<<
- #: DAJ RKP PNA JAQ" RKP =MA LNK@H I BC OK
=J <<<<<<<<
%#?M=C7 ESS= ! EN = <<<<<<<<< OK
QKN@N
%#3KAHEAN OEN GE@KBLSS=
% #2J 1=?CNR 5 AOK@ ODA 7 ESS=9 CMA =J @ODA
?KJ?MCA 7 ESS=N =H@LAJ@KJ OEN =>ND@?CNU
% #: DAJ = ?HNN ENGJ@E@N=J K>FA?CNU =
?KJ?MCA ?HNN EN <<<<<<< KJ O@OK>FA?O
% #/ HIB=?CNR L=COM N=H@Q PN OK <<<<<<<
K>FA?O?M=OKJ
```

" %!

```
&#: A PNA@<<<<<<<<< 日 9IE LIA 1=?CNR
=J @/ >ND@?O1=?CNR =J @E DAN@J?A 日 1=?CNR
5 AOK@
' #/ >ND@?O1=?CNR ?M=ON = <<<<<<< KB
LNK@?ON
) #6 KO= 8 0 / 4 B=?CNR L=COM" >POD=J @R
JKJ AODAHANN
%#?M=O@7 ESS= ! EN = <<<<<<<< OK
QKN@N
```



# er ise solutions



## Sharpen your pencil

We've knocked out the `Pizza`; just two more to go and we'll be ready to franchise  
Write the Chicago and California `Pizza` implementations here

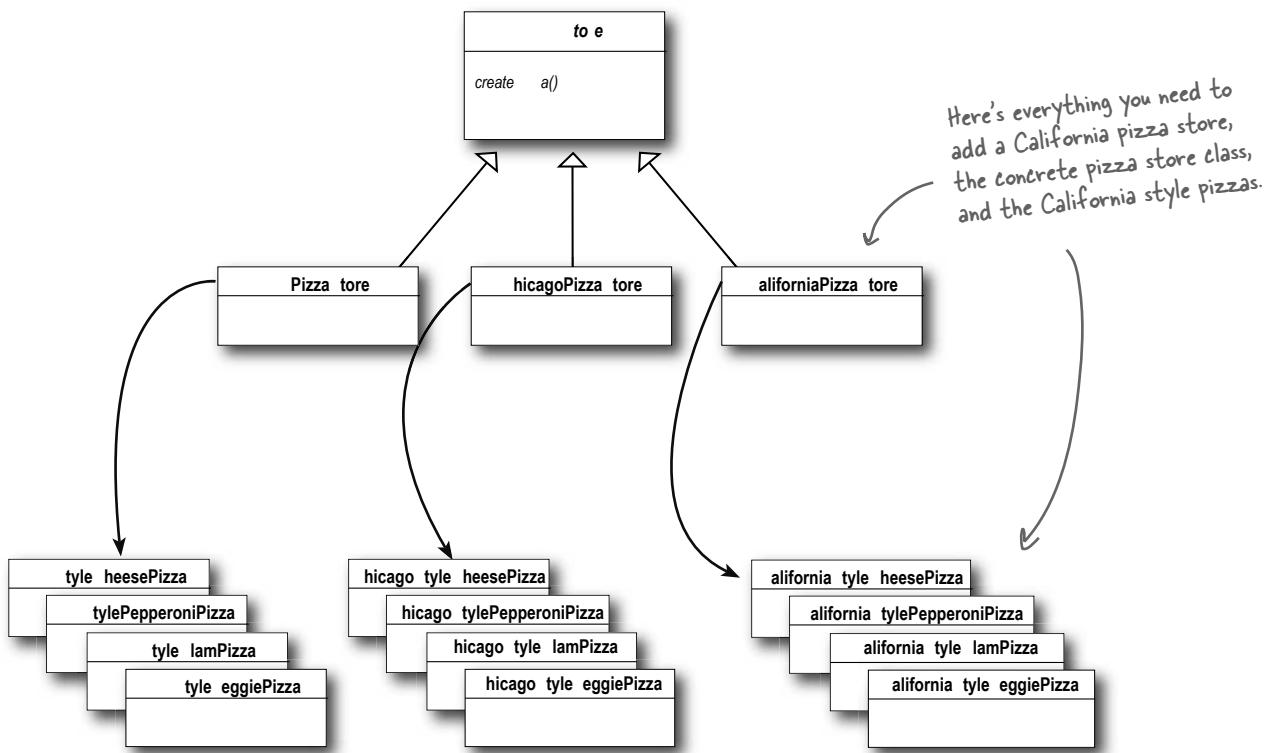
Both of these stores are almost exactly like the New York store... they just create different kinds of pizzas

```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza(); ← For the Chicago pizza
        } else if (item.equals("veggie")) { ← store, we just have to
            return new ChicagoStyleVeggiePizza(); ← make sure we create
        } else if (item.equals("clam")) { ← Chicago style pizzas...
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza(); ← and for the California
        } else if (item.equals("veggie")) { ← pizza store, we create
            return new CaliforniaStyleVeggiePizza(); ← California style pizzas.
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

# Design Principle Solution

We need another kind of pizza for those crazy Californians (crazy in a good way of course). I'll draw another parallel set of classes that you'd need to add a new California region to our Pizza store.



Okay, now write the five silliest things you can think of to put on a pizza. When, you'll be ready to go into business making pizza in California.

Here are our suggestions...

Mashed Potatoes with Roasted Garlic

BBQ Sauce

Artichoke Hearts

M&M's

Peanuts

# A very dependent PizzaStore



Let's pretend you've never heard of an **Pizza** 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:

8

number

|  
number with California too



## Sharpen your pencil

Go ahead and write the ChicagoPizzaIngredientFactory; you can reference the classes below in your implementation

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new ThickCrustDough();
    }

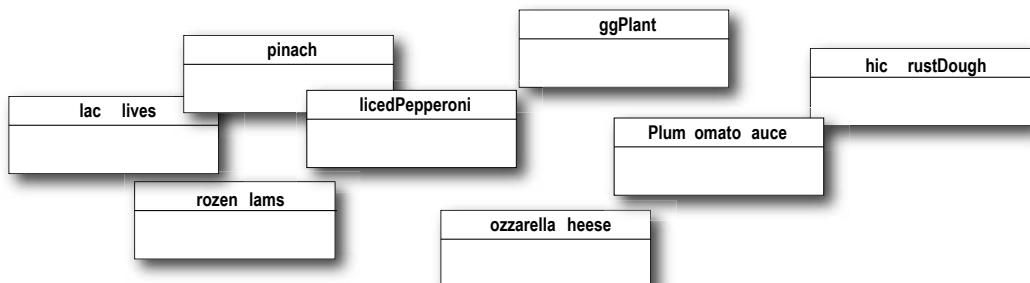
    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
            new Spinach(),
            new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```





# Pu le Solution

	1	2	<	;	1	>	3	@	3	1	>	3	/	@	<	>		3	4
					0														
					8														
					3														
5	?																		
	6	1	>	3	/	@	<	>						7	>	3	5	5	7
																/	;	<	
7	@																		9
:																			
	8	1	<	;	1	>	3	@	3	4	/	1	@	<	>	c			
=			<																
9		9	7	:	=	9	3	:	3	;	@	/	@	7	<	;		10	
3			=																c
11	4	/	1	@	<	>	c	:	3	@	6	<	2						?
	/																		@
1		12	1	6	7	1	/	5	<	?	@	c	9	3					c
@																			9
<		13	=	7	d	d	/								14	2	3	=	3
>			<														;	2	3
C		15	3	;	1	/	=	?	A	9	/	@	3						@

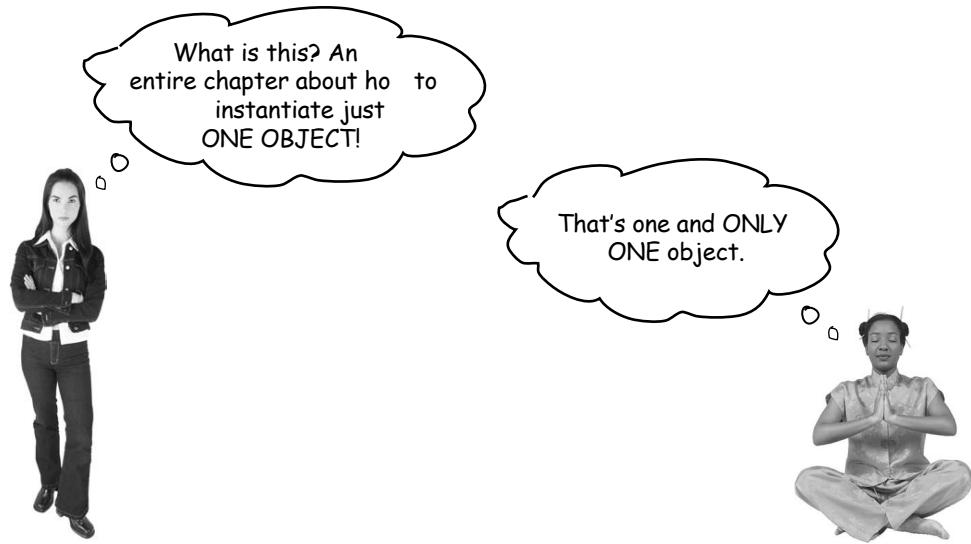
the in eton attern

# \* ne of a kind effects \*



o mi t e  
appy to now that all pattern, the Singleton implementation of class diagram; in fact, the diagram is old and stale! But don't let too comfortable; despite its simplicity from a class design perspective, we are going to encounter it a few times and possibly in its implementation. So, let's begin.

## one and only one



**Developer** What use is that?

**uru** There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for login, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

**Developer** Okay, so maybe there are classes that should only be instantiated once, but do I need a whole chapter for this? Can't I just do this by convention or by global variables? You know, like in Java, I could do it with a static variable.

**uru** In many ways, the Singleton Pattern is aacent for ensuring one and only one object is instantiated for a given class. If you've got a better one, the world would like to hear about it; but remember, like all patterns, the Singleton Pattern is a time-tested method for ensuring only one object gets created. The Singleton Pattern also gives us a global point of access, just like a global variable, but without the downsides.

**Developer** What do downsides?

**uru** Well, here's one example: if you assign an object to a global variable, then that object might be created when your application begins. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.

**Developer** This still doesn't seem like it should be so difficult.

**uru** If you've got a good handle on static class variables and methods as well as access modifiers, it's not. But, in either case, it is interesting to see how a Singleton works, and, as simple as it sounds, Singleton code is hard to get right. Just ask yourself: how do I prevent more than one object from being instantiated? It's not so obvious, is it?

# he ittle ingleton

A small Socratic exercise in the style of *The Little Sisper*

---

How would you create a single object?

```
new MyObject();
```

---

nd, what if another object wanted to create a  
My bject? Could it call new on My bject again?

es, of course.

o as long as we have a class, can we always  
instantiate it one or more times?

es. Well, only if it's a public class.

nd if not?

Well, if it's not a public class, only classes in the same  
package can instantiate it. But they can still instantiate  
it more than once.

---

Hmm, interesting.

id you know you could do this?

o, I'd never thought of it, but I guess it makes  
sense because it is a legal definition.

```
public MyClass {  
    private MyClass() {}  
}
```

What does it mean?

suppose it is a class that can't be instantiated  
because it has a private constructor.

---

Well, is there object that could use  
the private constructor?

Hmm, I think the code in MyClass is the only  
code that could call it. But that doesn't make  
much sense.

Why not ?

Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken and egg problem. I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use new MyClass().

---

kay. It was just a thought.

What does this mean?

MyClass is a class with a static method. We can call the static method like this

```
MyClass.getInstance();
```

```
public MyClass {  
    public static MyClass getInstance() {  
        }  
}
```

Why did you use MyClass, instead of some object name?

Well, getInstance() is a static method; in other words, it is a **CL** method. You need to use the class name to reference a static method.

---

ery interesting. What if we put things together.  
can instantiate a MyClass?

Wow, you sure can.

```
public MyClass {  
    private MyClass() {}  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}
```

o, now can you think of a second way to instantiate an object?

```
MyClass.getInstance();
```

Can you finish the code so that only  
of MyClass is ever created?

instance

es, I think so...

( You'll find the code on the next page.)

*ha ter*

---

# Dissecting the classic Singleton Pattern implementation

```

public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}

```

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.



at it  
If you're just flipping through the book, don't blindly type in this code, you'll see a few issues later in the chapter.

## Code Up Close

`uniqueInstance` holds our ONE instance; remember, it is a static variable.

```

if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;

```

If `uniqueInstance` is null, then we haven't created the instance yet...

...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to `uniqueInstance`. Note that if we never need the instance, it never gets created; this is lazy instantiation.

By the time we hit this code, we have an instance and we return it.

If `uniqueInstance` wasn't null, then it was previously created. We just fall through to the return statement.



oday we are pleased to bring you an interview with a ingleton object. Why don't you begin by telling us a bit about yourself.

Well, I'm totally unique; there is just one of me  
ne?

es, one. I'm based on the ingleton Pattern, which assures that at any one time there is only one instance of me.

sn't that sort of a waste? Someone took the time to develop a full blown class and now all we can get is one object out of it?

ot at all here is power in . Let's say you have an object that contains registry settings. You don't want multiple copies of that object and its values running around that would lead to chaos. By using an object like me you can assure that every object in your application is making use of the same global resource.

ell us more

h, I'm good for all kinds of things. Being single sometimes has its advantages you know. I'm often used to manage pools of resources, like connection or thread pools.

till, only one of your kind? That sounds lonely.

Because there's only one of me, do keep busy, but it would be nice if more developers knew me many developers run into bugs because they have multiple copies of objects floating around they're not even aware of.

o, if we may ask, how do you know there is only one of you? Can't anyone with a new operator create a new you ?

ope I'm truly unique.

Well, do developers swear an oath not to instantiate you more than once?

f course not. The truth be told well, this is getting kind of personal but have no public constructor.

P BL C C C h, sorry, no public constructor?

hat's right. My constructor is declared private.

How does that work? How do you get instantiated?

ou see, to get a hold of a ingleton object, you don't instantiate one, you just ask for an instance. So my class has a static method called `getInstance()`. Call that, and I'll show up at once, ready to work. In fact, I may already be helping other objects when you request me.

Well, Mr. ingleton, there seems to be a lot under your covers to make all this work. Thanks for revealing yourself and we hope to speak with you again soon

# The Chocolate Factory

veryone knows that all modern chocolate factories have computer controlled chocolate boilers. The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.

Here's the controller class for Choc Holic, Inc.'s industrial strength Chocolate Boiler. Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 100 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler.

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

*This code is only started when the boiler is empty!*

*To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.*

*To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.*

*To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.*

you are here ▶



Coc-O-Holic a done a decent o o en rin adt in dont appen, dont yat in ? T en a ain, yo pro a ly pect t at i two CocolateBoiler in tance et loo e, ome ery adt in can appen.

How mi tt in o wron i more t an one in tance o CocolateBoiler i created in an application?



Can yo elp Coc-O-Holic impro e t eir CocolateBoiler cla y t rmin it into a in eton?

```
public class ChocolateBoiler {  
    private boolean empty;  
    private boolean boiled;  
  
    ChocolateBoiler() {  
        empty = true;  
        boiled = false;  
    }  
  
    public void fill() {  
        if (isEmpty()) {  
            empty = false;  
            boiled = false;  
            // fill the boiler with a milk/chocolate mixture  
        }  
    }  
    // rest of ChocolateBoiler code...  
}
```

# Singleton Pattern defined

**Now that you've got the classic illustration of a singleton in your head, it's time to sit back, enjoy a bar of chocolate, and check out the finer points of the Singleton Pattern.**

**Let's start with the concise definition of the pattern:**

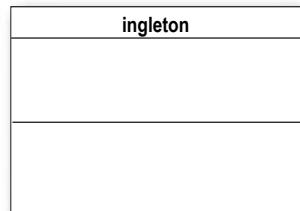
**The Singleton pattern** ensures a class has only one instance, and provides a global point of access to it.

**No big surprises there, but, let's read it down a bit more.**

- What's really going on here? We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
- We're also providing a global access point to the instance whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the singleton is created in a lazy manner, which is especially important for resource intensive objects.

**Okay, let's check out the class diagram.**

The `getInstance()` method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using `Singleton.getInstance()`. That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



The `uniqueInstance` class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

*you are here ▶*

thread are a ro e

Hershey, PA

~~Houston, we have a problem...~~

It looks like the chocolate oiler has let us down despite the fact we proved the code using classic singleton, so somehow the chocolate oiler's `oil` method was able to start oiling the oiler even though a batch of oil and chocolate was already oiling that's gallons of spilled oil and chocolate what happened?

We don't know what happened! The new Singleton code was running fine. The only thing we can think of is that we just added some optimizations to the Chocolate Boiler Controller that makes use of multiple threads.



ha ter

Could the addition of threads have caused this? Isn't it the case that once we've set the unique instance variable to the sole instance of chocolate oiler, all calls to `get_instance` should return the same instance? Right?

# BE the JVM



e a e t o t rea s ea e e uting t is o e our o is to la t e  
 an eter ine et er t ere is a ase in i t o t rea s ig t get a ol  
 o i erent oiler o e ts int:  
 ou reall ust nee to loo at t e  
 se uen e o o erations  
 in t e get nstan e  
 et o an t e alue o  
 uni ue nstan e to see  
 o t e ig t o erla  
 set e o e agnets to el  
 ou stu o t e o e ig t interlea e to reate t o oiler o e ts

```
ChocolateBoiler boiler =
    ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

```
public static ChocolateBoiler
getInstance() {
```

```
    if (uniqueInstance == null) {
```

```
        uniqueInstance =
            new ChocolateBoiler();
```

```
}
```

```
    return uniqueInstance;
```

```
}
```

**a e sure you chec your answer on  
page efore turning the page**

rea  
ne

rea  
o

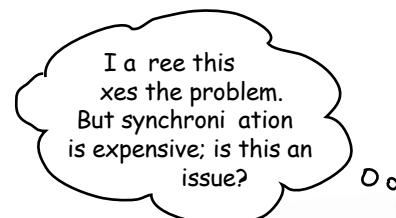
alue o  
uni ue nstan e

## Dealing with multithreading

**ur multithreading woes are almost trivially solved by adding  
get instance a synchronized method**

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.



Good point, and it's actually a little worse than you make out—the only time synchronization is relevant is the first time through this method. In other words, once we've set the unique instance variable to an instance of Singleton, we have no further need to synchronize this method. After the first time through, synchronization is totally unneeded overhead.

# Can we improve multithreading?

For most Java applications, we obviously need to ensure that the singleton works in the presence of multiple threads. But, it looks fairly expensive to synchronize the `getInstance()` method, so what do we do?

Well, we have a few options...

## . Do nothing if the performance of `getInstance` isn't critical to your application

That's right; if calling the `getInstance()` method isn't causing substantial overhead for your application, forget about it. Synchronizing `getInstance()` is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 10x, so if a high-traffic part of your code begins using `getInstance()`, you may have to reconsider.

## 2. Move to an eagerly created instance rather than a lazily created one

If your application always creates and uses an instance of the singleton or the overhead of creation and runtime aspects of the singleton are not onerous, you may want to create your singleton eagerly, like this

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

Using this approach, we rely on the JVM to create the unique instance of the singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static unique variable.

dou e he ed o in

## . Use double-checked locking to reduce the use of synchronization in getInstance

With double checked locking, we first check to see if an instance is created, and if not, this way, we only synchronize the first time through, just what we want.

Let's check out the code

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

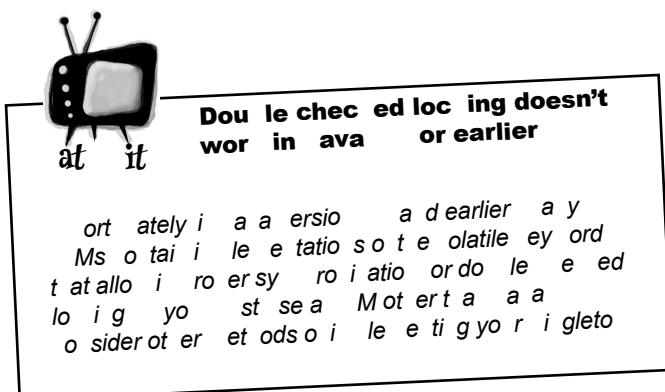
Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

\* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

If performance is an issue in your use of the getInstance() method then this method of implementing the Singleton can drastically reduce the overhead.



ha ter

## Meanwhile, back at the Chocolate Factory...

While we've been off diagnosing the multithreading problems, the chocolate boiler has been cleaned up and is ready to go. But first, we have to fix the multithreading problems. We have a few solutions at hand, each with different tradeoffs, so which solution are we going to employ?



### Sharpen your pencil

For each solution, describe its applicability to the problem in the Chocolate Boiler code:

n ron e t e et n t n e et o

---

---

e e er n t nt t on

---

---

o le e e lo n

---

---

## Congratulations

At this point, the Chocolate Factory is a happy customer and Chocoholic was glad to have some expertise applied to their boiler code. No matter which multithreading solution you applied, the boiler should be in good shape with no more mishaps. Congratulations. You've not only managed to escape lbs of hot chocolate in this chapter, but you've been through all the potential problems of the singleton.

## Dumb Questions

**Q:** or such a simple pattern consisting of only one class, singletons sure seem to have some problems.

**A:** Well, we warned you up front! But don't let the problems discourage you while implementing Singletons *correctly* can be tricky, after reading this chapter you are now well informed on the techniques for creating Singletons and should use them wherever you need to control the number of instances you are creating.

**Q:** Can't I just create a class in which all methods and variables are defined as static? Wouldn't that be the same as a singleton?

**A:** Yes, if our class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy, especially if multiple classes are involved. Often this scenario can result in subtle, hard-to-find bugs involving order of initialization. Unless there is a compelling need to implement our singleton this way, it is far better to stay in the object world.

**Q:** What about class loaders? I heard there is a chance that two class loaders could each end up with their own instance of singleton.

**A:** Yes, that is true as each class loader defines a namespace. If you have two or more classloaders, you can load the same class multiple times once in each classloader. Now, if that class happens to be a Singleton, then since we have more than one version of the class, we also have more than one instance of the Singleton. So, if you are using multiple classloaders and Singletons, be careful. One way around this problem is to specify the classloader yourself.



u or o in eton ein eaten y the ar a e  
o e tor are reat y exa erated

riorto a a a g i t e gar age olle tor allo ed i gleto s  
to e re at rely olle ted i tere as o glo alreere e tot e ot er  
ords yo o ld reate a i gleto a d i t e o lyreere e tot e i gleto  
asi t e i gleto itsel it o ld e olle ted a d destroyed yt e gar age  
olle tor is leads to o si g gs e a se a ter t e i gleto is  
olle ted t e e t all to get sta e rod ed a s i y e i gleto  
a ya li atio s t is a a se o si g e a ior as state is ysterio sly  
reset to i itial al es ort i gs li e et or o e tio s are reset

i e a a t is g as ee ed a d a glo alreere e is o lo ger  
re ired yo are or so e reaso still si g a re a a M t e e  
a are o t is iss e ot er ise yo a slee ell o i g yo r i gleto s  
o t e re at rely olle ted

**Q:** I've always been taught that a class should do one thing and one thing only. or a class to do two things is considered bad OO design. Isn't a singleton violating this?

**A:** You would be referring to the one class, one responsibility principle, and yes, you are correct, the Singleton is not only responsible for managing its one instance and providing global access, it is also responsible for whatever its main role is in our application. So, certainly it can be argued it is taking on two responsibilities. nevertheless, it isn't hard to see that there is utility in a class managing its own instance it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

**Q:** I wanted to subclass my singleton code, but I ran into problems. Is it okay to subclass a singleton?

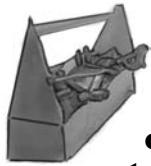
**A:** One problem with subclassing Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change our constructor so that it's public or protected. But then, it's not really a Singleton anymore, because other classes can instantiate it.

If you do change our constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of our derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing registration of sorts is required in the base class.

Before implementing such a scheme, you should ask yourself what you are really gaining from subclassing a Singleton. In most patterns, the Singleton is not necessarily meant to be a solution that can fit into a library. In addition, the Singleton code is trivial to add to an existing class. Just, if you are using a large number of Singletons in your application, you should take a hard look at our design. Singletons are meant to be used sparingly.

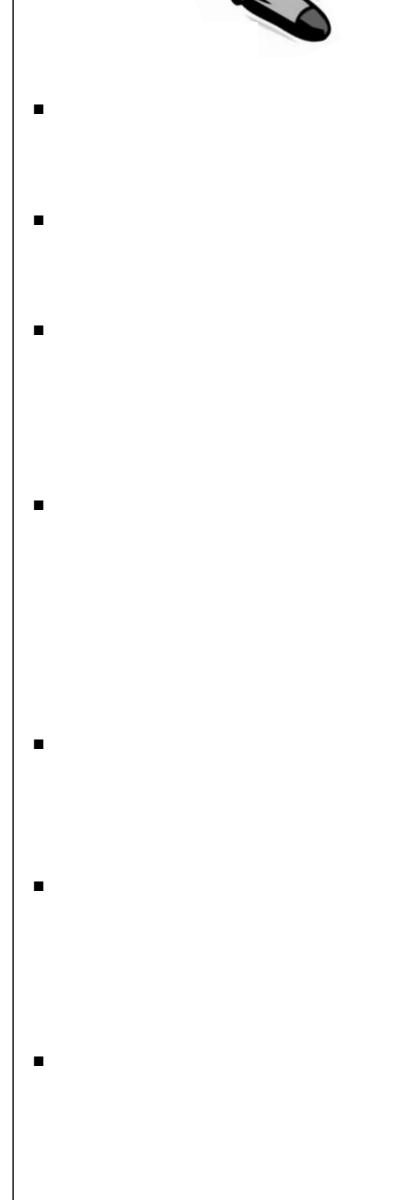
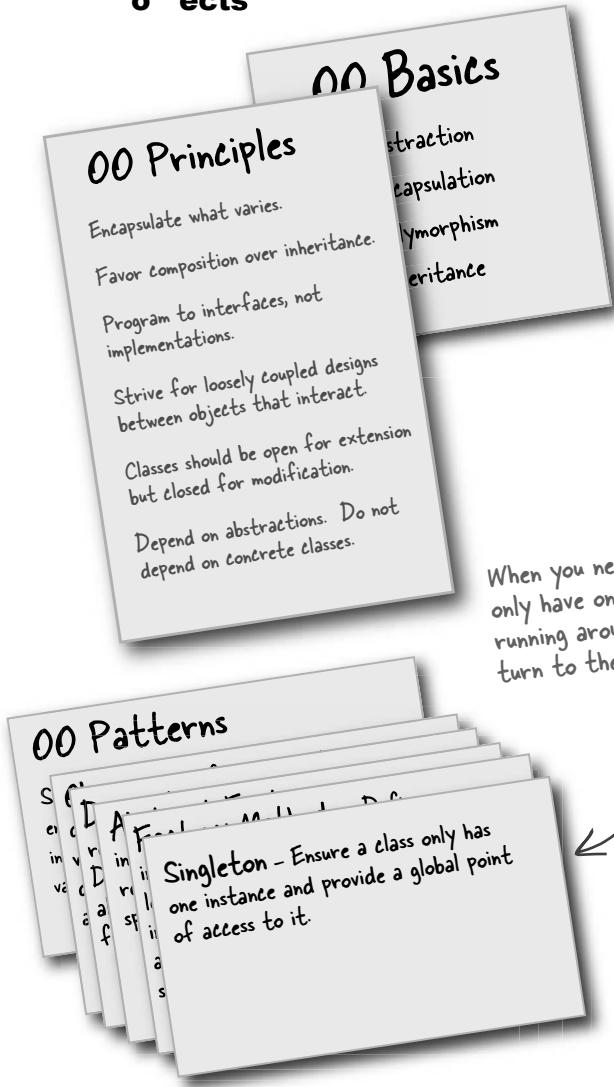
**Q:** I still don't totally understand why global variables are worse than a singleton.

**A:** In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. I've already mentioned one: the issue of late versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. Global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.

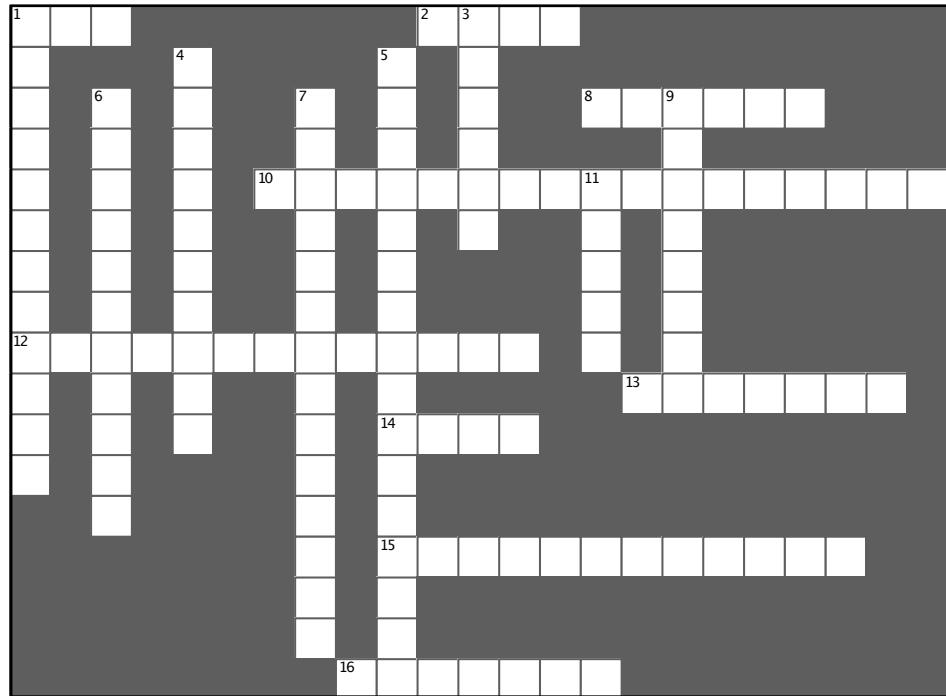


## Tools for your Design Toolbox

You've now added another pattern to your tool box. Singleton gives you another method of creating objects. In this case, unique objects.



As you've seen, despite its apparent simplicity, there are a lot of details involved in the Singleton's implementation. After reading this chapter, though, you are ready to go out and use Singleton in the wild.



### #'\$\$

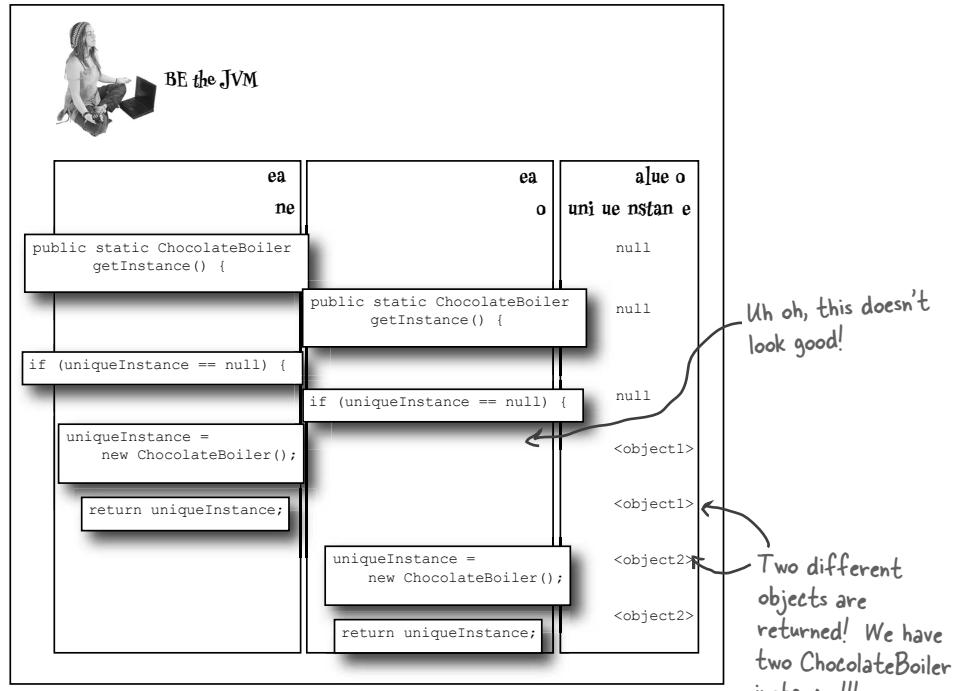
&\$3MP<L I H@ A< EDH?  
 '\$0??@ M! >I >I F<MDH M@=I D@X  
 -\$0H D@I K@M@JF@G@MNDH ><NL@ M@ M  
 I O@T P  
 &%\$8 DBF@MH JK C@ < LD@DBF@BLM@>@>H?  
 !M@K@P! K@L"  
 & \$2R@ G N@M@?DB <JJK >>C DAH MNLDB  
 4<O< &\$  
 &(\$1 C >I F<M@><J DMFI AM@: 8  
 &) \$6 H@<?O<HMB@I Q@XBF =<FO<K@=F@/  
 ; ; ; ; ; ; >K@M@H  
 &\* \$1 I GJ<HR M@<MK ?N@ = I D@L  
 &+\$9I MM@R ?@M@H@>I HLM@MK#P@  
 C<O@M ?@F@M@>I HLM@MK; ; ; ; ; ;

### " %!

&\$5 N@P@; ; ; ; ; ; ; ><H ><NL@JK =F@GL  
 (\$0 8 DBF@MH D < >FLL M@<MG <H@B@ <H  
 BLM@>@I A; ; ; ; ; ;  
 )\$3AR N?I HM@> M PI K@R <=I NM@SR  
 BLM@N@D@H#R N ><H >K@M@R NK@BLM@>@  
 ; ; ; ; ; ;  
 \* \$7 K@KM & \$ #M@ ><H @MR NK@8 DBF@MH L !@P  
 PI K@L"  
 +\$9 C@8 DBF@MH P <L @>= <K@LL@? D@C<? H  
 JN@P ; ; ; ; ; ; ; ; ;  
 , \$9 C@>FLLD D@JF@G@MNDH ?I @HMC<H?F@  
 M@D  
 . \$8 DBF@MH @LN@ I HRI H@I AM@ @@@DM  
 &&\$9 C@8 DBF@MH 7 < M@H C<L I H@

*you are here ▶*

# Exercise solutions



Can you help Cocco-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

```

public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}

```

Answer

# er ise solutions



## Sharpen your pencil

For each solution, describe applicability to the problem of the Chocolate Boiler code:

### n ron et e etn t n e et o

A straightforward technique that is guaranteed to work. We don't seem to have any

performance concerns with the chocolate boiler, so this would be a good choice.

### e e er n t nt ton

We are always going to instantiate the chocolate boiler in our code, so statically initializing the

instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familiar with the standard pattern.

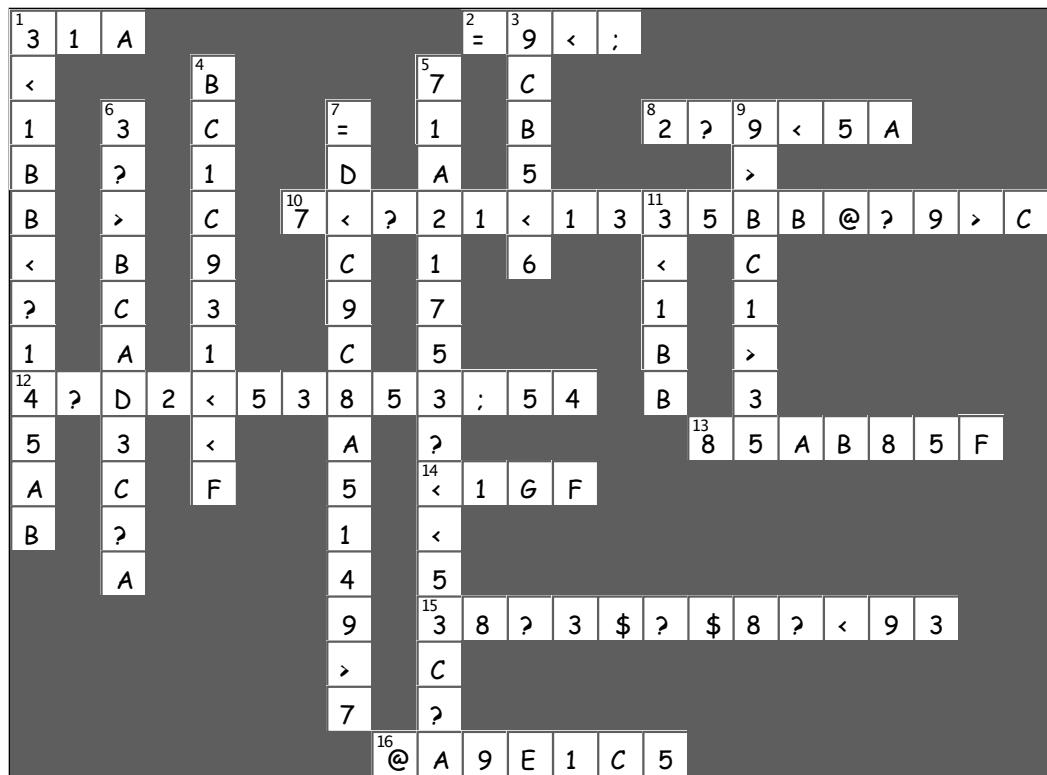
### o le e e lo n

Given we have no performance concerns, double-checked locking seems like overkill. In addition, we'd

have to ensure that we are running at least Java .



# er ise solutions



the omnman attern



## \* *encapsulating invocation* \*



T at ri t, y  
encap latin met od in ocation, we can cry talli e piece o comp tation ot atte  
o ect in o in t e comp tation doe nt need to worry a o t ow to do t in , it t e  
o r cry talli ed met od to et it done. We can al o do ome wic edly mart t in wit  
t e e encap lated met od in ocation , li e a et em away or lo in or re et em to  
implement ndo in o r code.

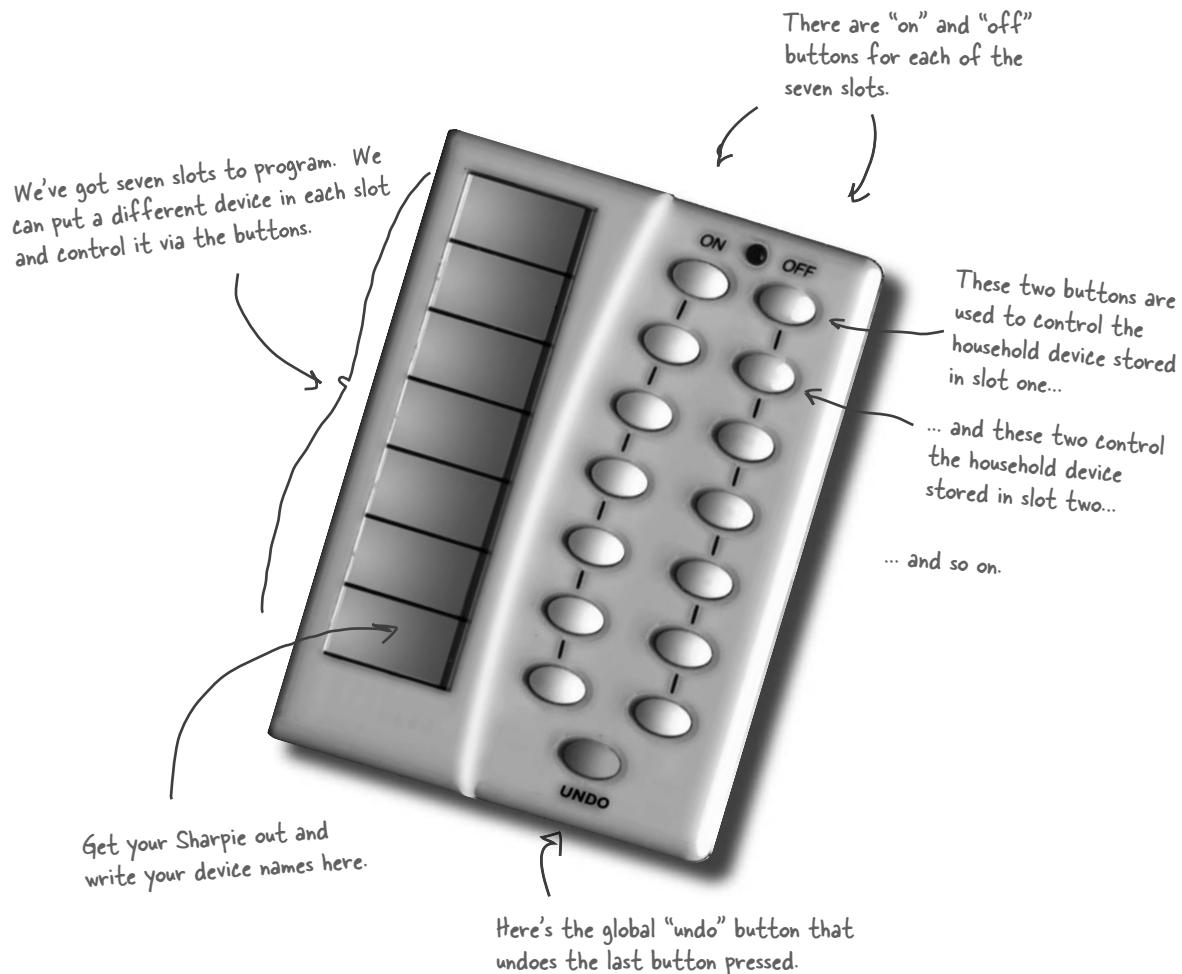


Home Automation or Bust, Inc.  
1221 Industrial Avenue, Suite 2000  
Future City, IL 62914

Billy Thompson

HOME AUTOMATION  
VENDOR CLASSES

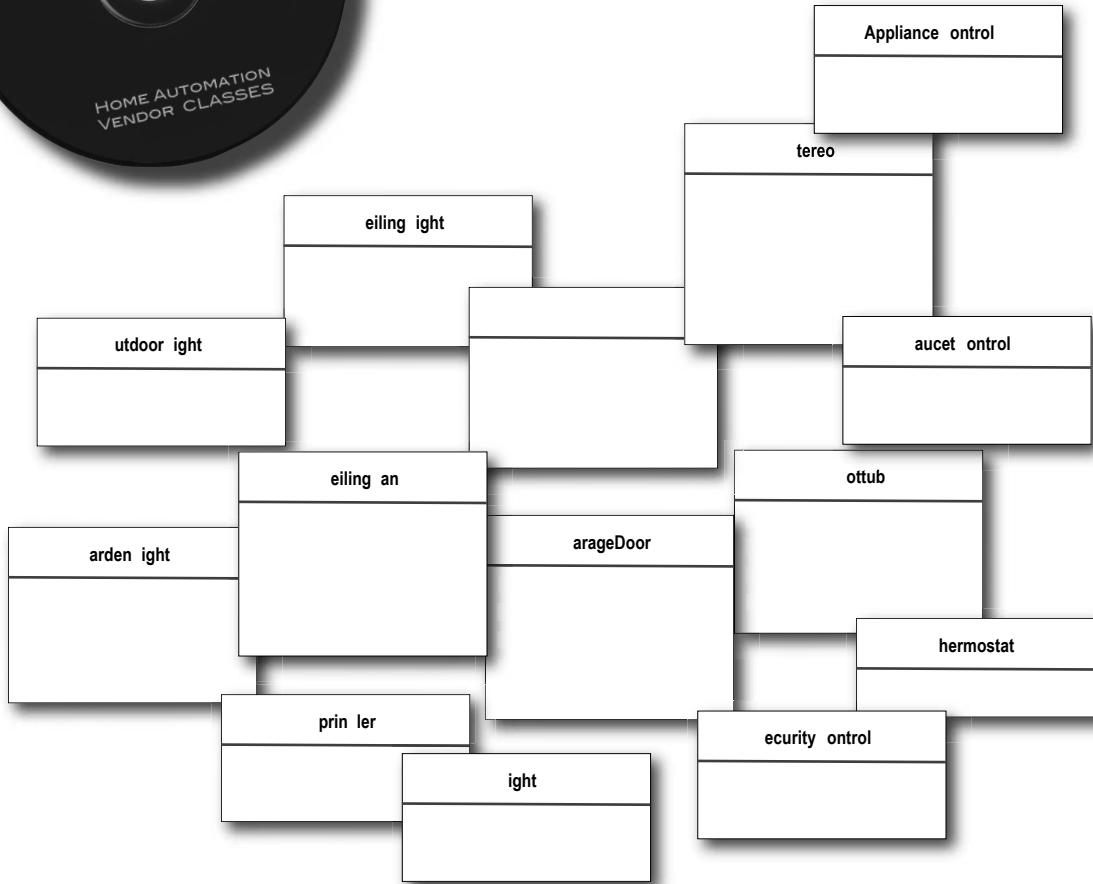
## Free hardware let's check out the Remote Control...





## Taking a look at the vendor classes

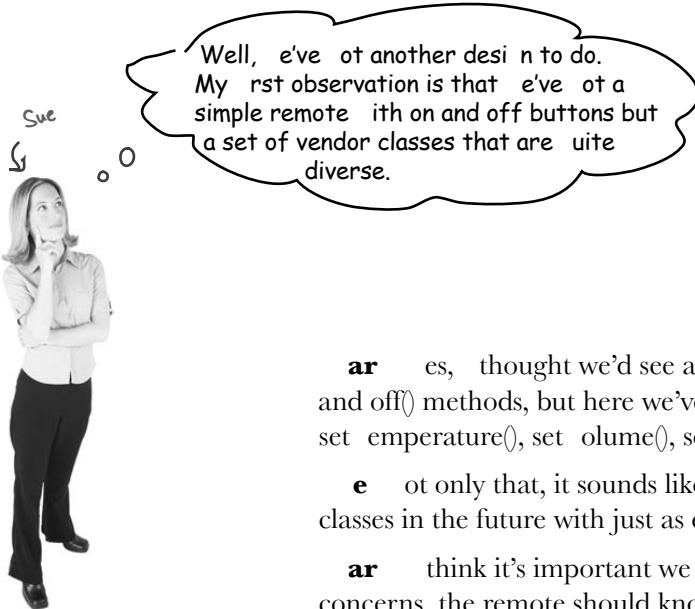
Check out the vendor classes on the C . These should give you some idea of the interfaces of the objects we need to control from the remote.



t looks like we have quite a set of classes here, and not a lot of industry effort to come up with a set of common interfaces. Not only that, it sounds like we can expect more of these classes in the future. Designing a remote control P is going to be interesting. Let's get on to the design.

## Cubicle Conversation

**our tea      ates are already discussing how to design the re    ote control    P**



**ar** es, thought we'd see a bunch of classes with on() and off() methods, but here we've got methods like dim(), setTemperature(), setVolume(), setIrrigation().

**e** ot only that, it sounds like we can expect more vendor classes in the future with just as diverse methods.

**ar** think it's important we view this as a separation of concerns: the remote should know how to interpret button presses and make requests, but it shouldn't know a lot about home automation or how to turn on a hot tub.

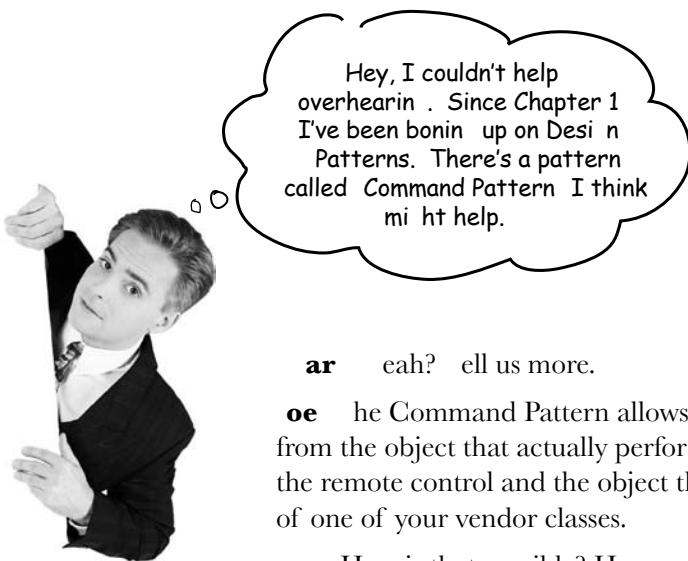
**e** ounds like good design. But if the remote is dumb and just knows how to make generic requests, how do we design the remote so that it can invoke an action that, say, turns on a light or opens a garage door?

**ar** 'm not sure, but we don't want the remote to have to know the specifics of the vendor classes.

**e** What do you mean?

**ar** We don't want the remote to consist of a set of if statements, like if slot Light, then light.on(), else if slot Hottub then hottub.jets on(). We know that is a bad design.

**e** agree. Whenever a new vendor class comes out, we'd have to go in and modify the code, potentially creating bugs and more work for ourselves.



Hey, I couldn't help overhearin'. Since Chapter 1 I've been bonin' up on Design Patterns. There's a pattern called Command Pattern I think mi ht help.

**ar** eah? ell us more.

**oe** he Command Pattern allows you to decouple the requester of an action from the object that actually performs the action. o, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

**e** How is that possible? How can we decouple them? fter all, when press a button, the remote has to turn on a light.

**oe** ou can do that by introducing command objects into your design. command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object). o, if we store a command object for each button, when the button is pressed we ask the command object to do some work. he remote doesn't have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done. o, you see, the remote is decoupled from the light object

**e** his certainly sounds like it's going in the right direction.

**ar** till, 'm having a hard time wrapping my head around the pattern.

**oe** Given that the objects are so decoupled, it's a little difficult to picture how the pattern actually works.

**ar** Let me see if at least have the right idea using this pattern we, could create an P in which these command objects can be loaded into button slots, allowing the remote code to stay very simple. nd, the command objects encapsulate how to do a home automation task along with the object that needs to do it.

**oe** es, think so. also think this pattern can help you with that endo button, but haven't studied that part yet.

**ar** his sounds really encouraging, but think have a bit of work to do to really get the pattern.

**e** Me too.

## Meanwhile, back at the Diner... or, A brief introduction to the Command Pattern

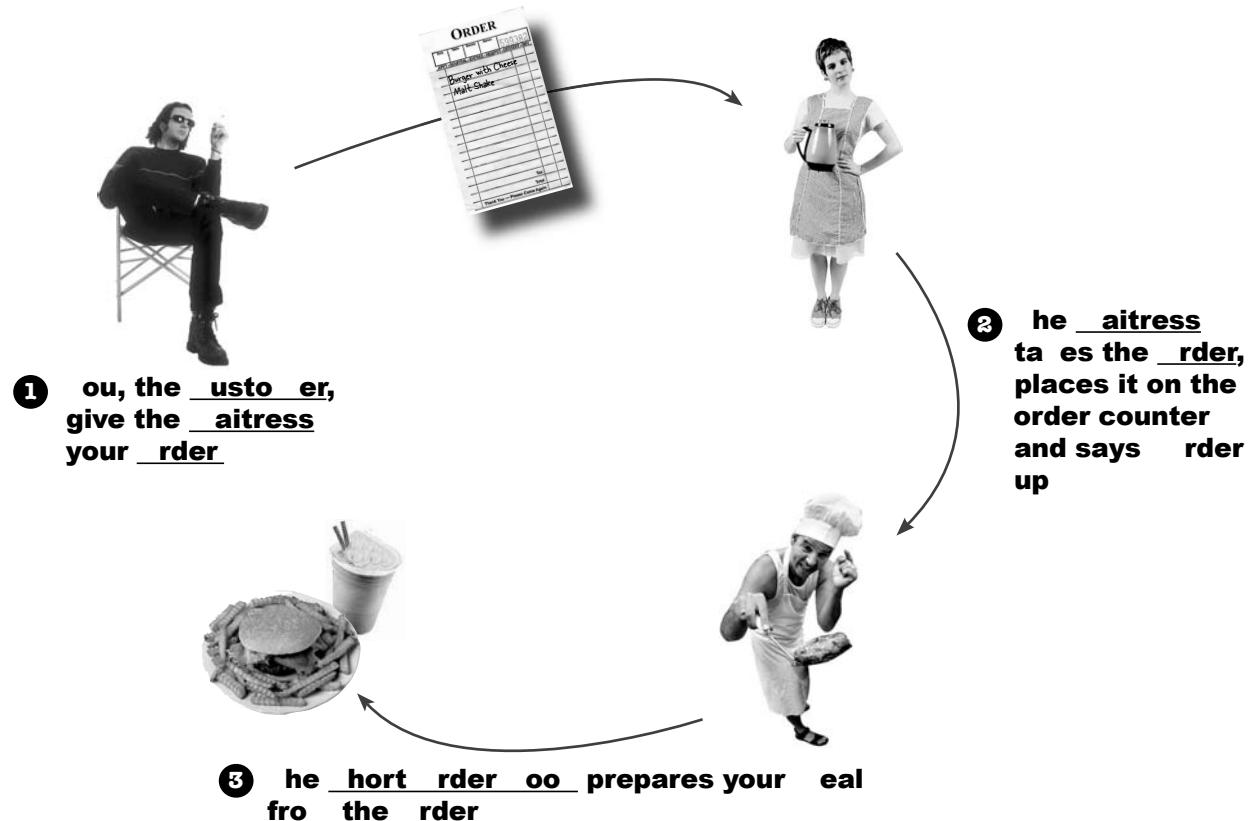
So said, it is a little hard to understand the Command Pattern by just hearing its description. But don't fear, we have some friends ready to help remember our friendly diner from Chapter ? It's been a while since we visited Alice, Flo, and the short order cook, but we've got good reason for returning (well, beyond the food and great conversation) the diner is going to help us understand the Command Pattern.

Let's take a short detour back to the diner and study the interactions between the customers, the waitress, the orders and the short order cook. Through these interactions, you're going to understand the objects involved in the Command Pattern and also get a feel for how the decoupling works. After that, we're going to knock out that remote control P.

Checking in at the Objectville Diner...

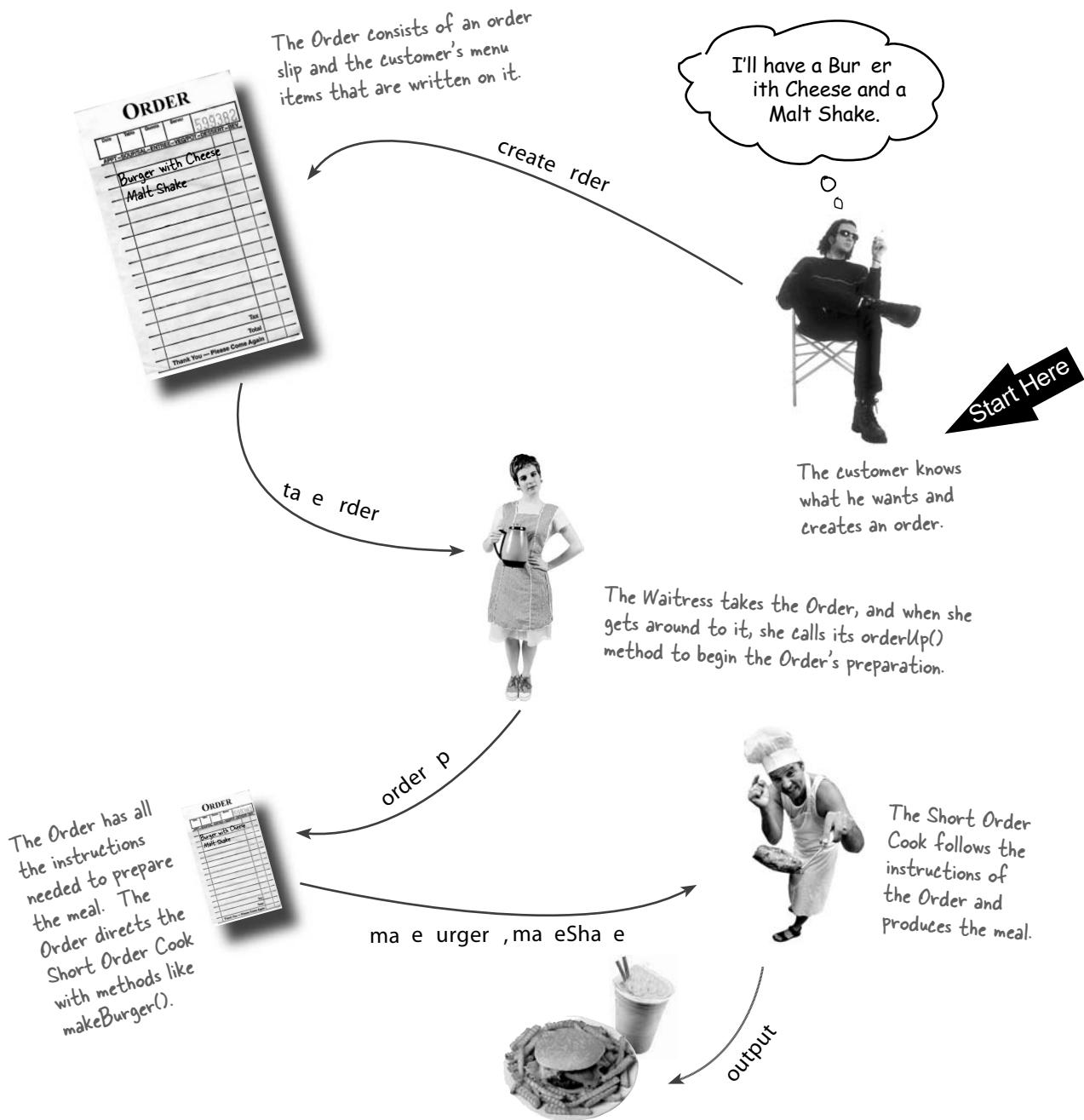


### **ay, we all now how the Diner operates**



## Let's study the interaction in a little more detail...

and given this Diner is in Dectville, let's think about the object and method calls involved, too



# The Objectville Diner roles and responsibilities

## **n order lip encapsulates a request to prepare a meal**

**h o the rder lip as a object a object that acts as a request to prepare a meal.** Like any object, it can be passed around from the Waitress to the order counter, or to the next Waitress taking over her shift. It has an interface that consists of only one method, `order(p)`, that encapsulates the actions needed to prepare the meal. It also has a reference to the object that needs to prepare it (in our case, the Cook). It's encapsulated in that the Waitress doesn't have to know what's in the order or even who prepares the meal; she only needs to pass the slip through the order window and call `rder up`

## **he Waitress's job is to take order slips and invoke the order p method on the**

**he Waitress has to take a order from the customer to help customers tell she makes the back to the order counter the job of the order p method to have the meal prepared.** As we've already discussed, in Objectville, the Waitress really isn't worried about what's on the order or who is going to prepare it; she just knows order slips have an `order(p)` method she can call to get the job done.

Now, throughout the day, the Waitress's `takeOrder()` method gets parameterized with different order slips from different customers, but that doesn't phase her; she knows all order slips support the `order(p)` method and she can call `order(p)` any time she needs a meal prepared.

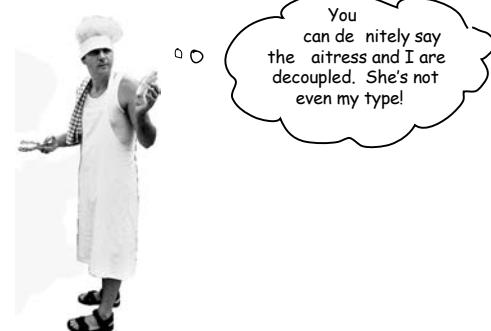
## **he short order cook has the knowledge required to prepare the meal**

**he short order cook is the object that really knows how to prepare meals.** Once the Waitress has invoked the `order(p)` method, the short order Cook takes over and implements all the methods that are needed to create meals.

Notice the Waitress and the Cook are totally decoupled: the Waitress has order slips that encapsulate the details of the meal; she just calls a method on each order to get it prepared. Likewise, the Cook gets his instructions from the order lip; he never needs to directly communicate with the Waitress.



Okay, in real life a waitress would probably care what is on the Order Slip and who cooks it, but this is Objectville... work with us here!



## the diner is a model or object and pattern



Okay, we have a Diner with a Waitress who is decoupled from the Cook by an Order Slip, so what? Get to the point!

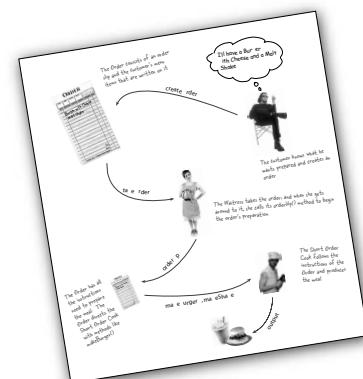
Patience, we're getting there...

Think of the Diner as a model for an design pattern that allows us to separate an object making a request from the objects that receive and execute those requests. For instance, in our remote control P, we need to separate the code that gets invoked when we press a button from the objects of the vendor specific classes that carry out those requests. What if each slot of the remote held an object like the Diner's order slip object? Then, when a button is pressed, we could just call the equivalent of the order p() method on this object and have the lights turn on without the remote knowing the details of how to make those things happen or what objects are making them happen.

Now, let's switch gears a bit and map all this Diner talk to the Command Pattern...



Before we move on, spend some time thinking about the diagram two parts accompanying the Diner role and responsibilities until you get a handle on the Objectville Diner object and its relationships. Once you've done that, get ready to nail the Command Pattern!

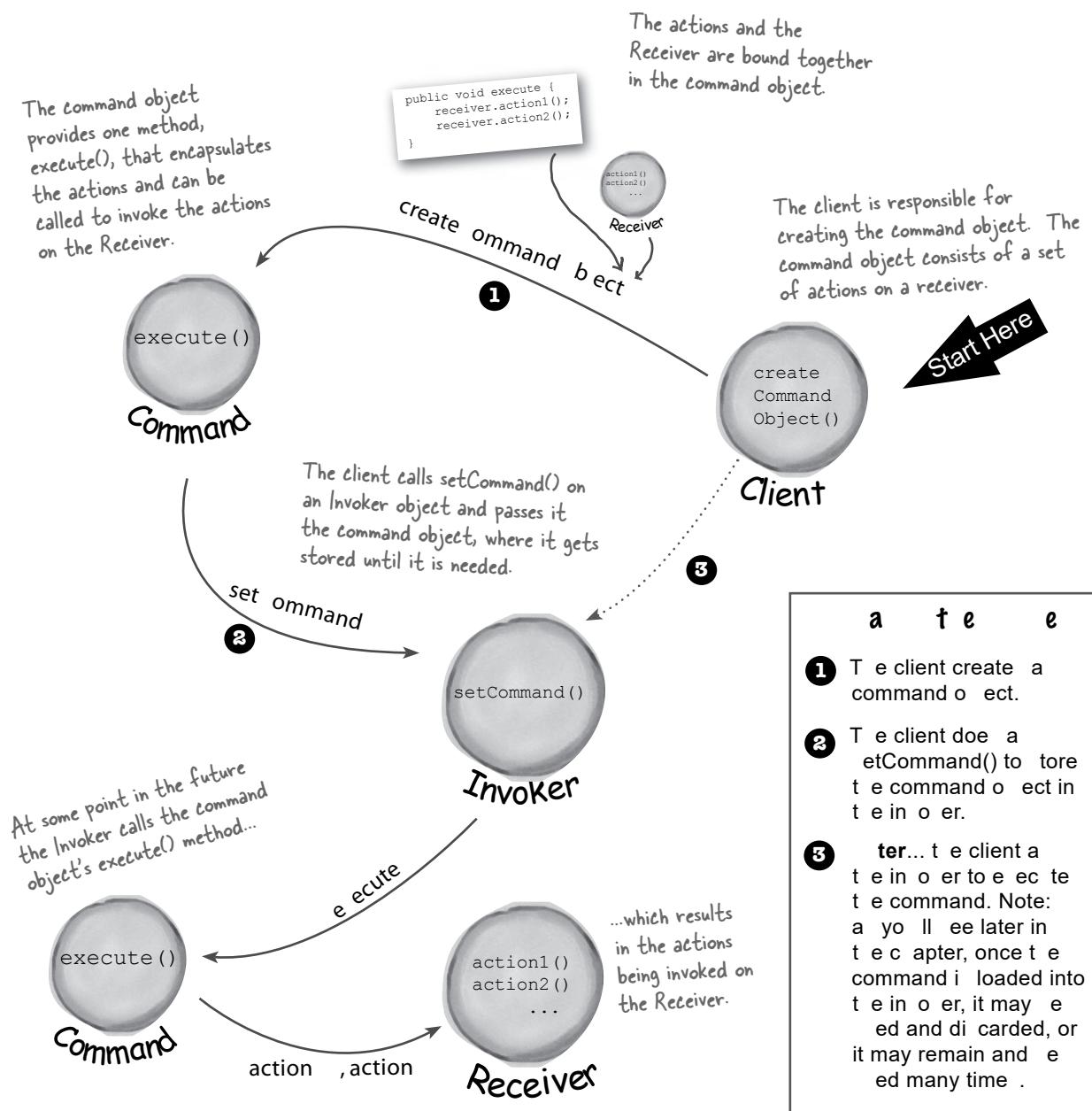


later

# the effect a atte

key, we've spent enough time in the objectville personalities and their responsibilities to quite well. the inner diagram to reflect the Command Pattern. players are the same; only the names have changed.

inner that we know all the now we're going to rework ou'll see that all the



## a t e e

- 1 The client creates a command object.
- 2 The client does a setCommand() to store the command object in the invoker.
- 3 Later... the client activates the invoker to execute the command. Note: a workflow later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.

you are here ▶

*ho doe*    *hat*

## \* WHO DOES ? WHAT? \*

Match the diner objects and methods with the corresponding names from the Command Pattern.

### **Diner**

### **o and Pattern**

actress

omman

hort r er oo

e e ute

or er

ient

r er

n o er

ustomer

e ei er

ta e r er

set omman

*ha ter*

# Our first command object

sn't it about time we build our first command object? Let's go ahead and write some code for the remote control. While we haven't figured out how to design the remote control yet, building a few things from the bottom up may help us...



## ple enting the o and interface

First things first all command objects implement the same interface, which consists of one method. In the inner we called this method order p(); however, we typically just use the name execute().

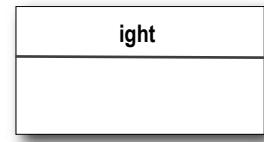
Here's the Command interface

```
public interface Command {  
    public void execute();  
}
```

Simple. All we need is one method called execute().

## ple enting a o and to turn a light on

ow, let's say you want to implement a command for turning a light on. Referring to our set of vendor classes, the Light class has two methods on() and off(). Here's how you can implement this as a command



This is a command, so we need to implement the Command interface.

```
public class LightOnCommand implements Command {  
    Light light;  
  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }  
  
    public void execute() {  
        light.on();  
    }  
}
```

The constructor is passed the specific light that this command is going to control – say the living room light – and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

ow that you've got a LightOnCommand class, let's see if we can put it to use...

u in the o and o e t

## s t ec a bject

kay, let's make things simple say we've got a remote control with only one button and corresponding slot to hold a device to control

```
public class SimpleRemoteControl {  
    Command slot;  
  
    public SimpleRemoteControl() {}  
  
    public void setCommand(Command command) {  
        slot = command;  
    }  
  
    public void buttonWasPressed() {  
        slot.execute();  
    }  
}
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

## C eat a s letestt set e e tec t l

Here's just a bit of code to test out the simple remote control. Let's take a look and we'll point out how the pieces match the Command Pattern diagram

```
public class RemoteControlTest {  
    public static void main(String[] args) {  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Light light = new Light();  
        LightOnCommand lightOn = new LightOnCommand(light);  
  
        remote.setCommand(lightOn);  
        remote.buttonWasPressed();  
    }  
}
```

This is our Client in Command Pattern-speak.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

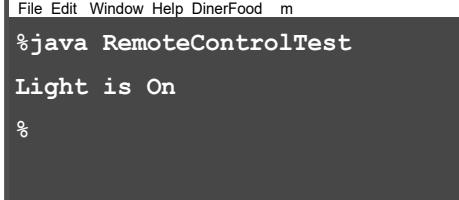
Now we create a Light object, this will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

And then we simulate the button being pressed.

Here, pass the command to the Invoker.

Here's the output of running this test code!



```
File Edit Window Help DinerFood m  
%java RemoteControlTest  
Light is On  
%
```

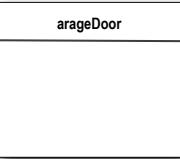
ha ter



## Sharpen your pencil

a test me or o to mpleme t the  
ara e oor pe ommand class.rst s ppl the code or  
the class below. o ll eed the ara e oor class d a ram.

```
public class GarageDoorOpenCommand  
    implements Command {
```



}

Your code here

ow that o e ot o r class what s the o tp to the o llow  
code t: the ara e oor p method pr ts o t ara e oor s  
pe whe t s complete.

```
public class RemoteControlTest {  
    public static void main(String[] args) {  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Light light = new Light();  
        GarageDoor garageDoor = new GarageDoor();  
        LightOnCommand lightOn = new LightOnCommand(light);  
        GarageDoorOpenCommand garageOpen =  
            new GarageDoorOpenCommand(garageDoor);  
  
        remote.setCommand(lightOn);  
        remote.buttonWasPressed();  
        remote.setCommand(garageOpen);  
        remote.buttonWasPressed();  
    }  
}
```



Your output here.

*you are here ▶*

## The Command Pattern defined

You've done your time in the Objectville diner, you've partly implemented the remote control API, and in the process you've got a fairly good picture of how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.

Let's start with its official definition

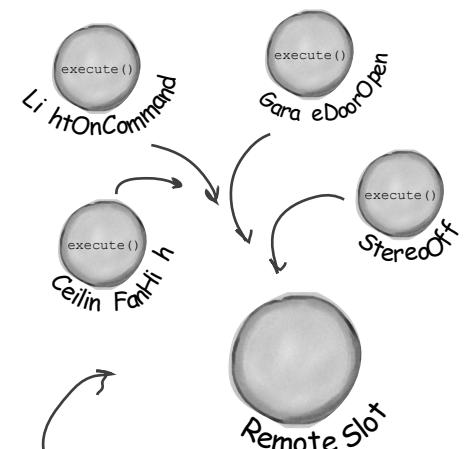
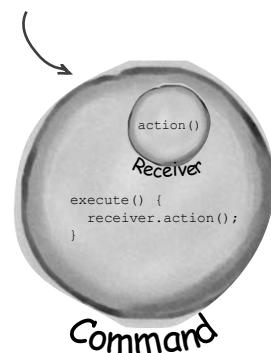
**The Command pattern** encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

Let's step through this. We know that a command object *represents a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute(). When called, execute() causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced.

We've also seen a couple examples of *parameterizing* with a command. Back at the diner, the Waitress was parameterized with multiple orders throughout the day. In the simple remote control, we first loaded the button slot with a light on command and then later replaced it with a garage door open command. Like the Waitress, your remote slot didn't care what command object it had, as long as it implemented the Command interface.

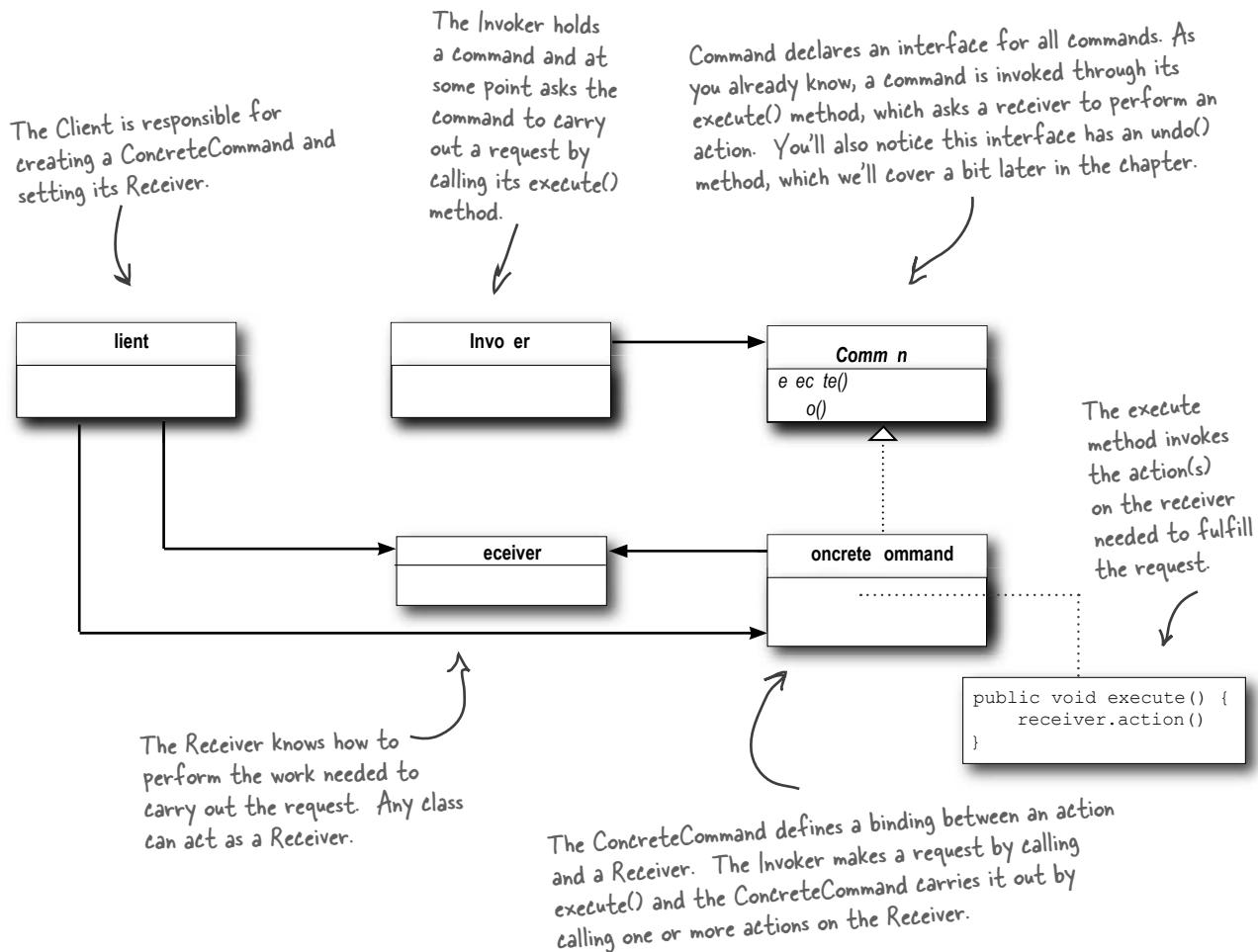
What we haven't encountered yet is using commands to implement *sequences* of *operations*. Don't worry, those are pretty straightforward extensions of the basic Command Pattern and we will get to them soon. We can also easily support what's known as the Meta Command Pattern once we have the basics in place. The Meta Command Pattern allows you to create macros of commands so that you can execute multiple commands at once.

An encapsulated request.



An invoker – for instance one slot of the remote – can be parameterized with different requests.

# The Command Pattern defined: the class diagram



How does the design of the Command Pattern support the decomposition of operations into requests?

you are here ▶

*here do e e in*



Okay, I think I've got a good feel for the Command Pattern now. Great tip Joe, I think we are going to look like superstars after nishing off the Remote Control API.

**ar** Me too. So where do we begin?

**e** Like we did in the simple remote, we need to provide a way to assign commands to slots. In our case we have seven slots, each with an on and off button. So we might assign commands to the remote something like this

```
onCommands[0] = onCommand;  
offCommands[0] = offCommand;
```

**ar** That makes sense, except for the Light objects. How does the remote know the living room from the kitchen light?

**e** Uh, that's just it, it doesn't. The remote doesn't know anything but how to call execute() on the corresponding command object when a button is pressed.

**ar** Yeah, sorta got that, but in the implementation, how do we make sure the right objects are turning on and off the right devices?

**e** When we create the commands to be loaded into the remote, we create one LightCommand that is bound to the living room light object and another that is bound to the kitchen light object. Remember, the receiver of the request gets bound to the command it's encapsulated in. So, by the time the button is pressed, no one cares which light is which, the right thing just happens when the execute() method is called.

**ar** I think I've got it. Let's implement the remote and I think this will get clearer.

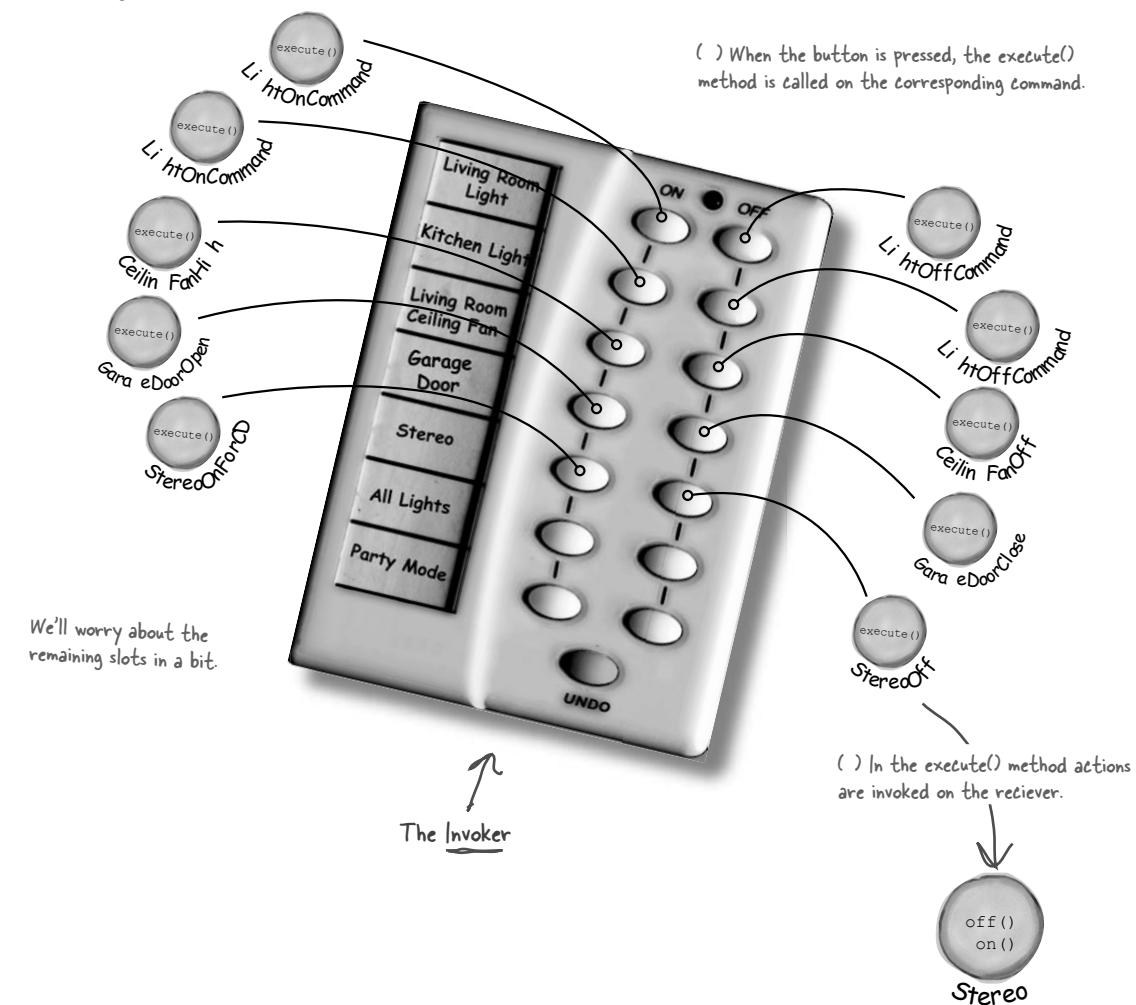
**e** Sounds good. Let's give it a shot...

*ha ter*

# Assigning Commands to slots

We have a plan. We're going to assign each slot to a command in the remote control. This makes the remote control our *invoker*. When a button is pressed the `execute()` method is going to be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans, stereos).

(1) Each slot gets a command.



# Implementing the Remote Control

```
public class RemoteControl {  
    Command[] onCommands;  
    Command[] offCommands;  
  
    public RemoteControl() {  
        onCommands = new Command[7];  
        offCommands = new Command[7];  
  
        Command noCommand = new NoCommand();  
        for (int i = 0; i < 7; i++) {  
            onCommands[i] = noCommand;  
            offCommands[i] = noCommand;  
        }  
    }  
  
    public void setCommand(int slot, Command onCommand, Command offCommand) {  
        onCommands[slot] = onCommand;  
        offCommands[slot] = offCommand;  
    }  
  
    public void onButtonWasPushed(int slot) {  
        onCommands[slot].execute();  
    }  
  
    public void offButtonWasPushed(int slot) {  
        offCommands[slot].execute();  
    }  
  
    public String toString() {  
        StringBuffer stringBuff = new StringBuffer();  
        stringBuff.append("\n----- Remote Control -----\\n");  
        for (int i = 0; i < onCommands.length; i++) {  
            stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()  
                + " " + offCommands[i].getClass().getName() + "\\n");  
        }  
        return stringBuff.toString();  
    }  
}
```

This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the on and off arrays for later use.

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

We've overwritten toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

## Implementing the Commands

Well, we've already gotten our feet wet implementing the LightOnCommand for the remote emoteControl. We can plug that same code in here and everything works beautifully. The off commands are no different; in fact the LightOffCommand looks like this

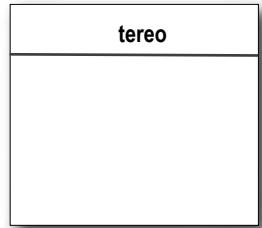
```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Let's try something a little more challenging; how about writing on and off commands for the stereo? Okay, off is easy, we just bind the stereo to the off() method in the StereoOffCommand. On is a little more complicated; let's say we want to write a stereo onWithCD Command...



```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 1, right?

Not too bad. Take a look at the rest of the vendor classes; by now, you can definitely knock out the rest of the Command classes we need for those.

*you are here ▶*

## Putting the Remote Control through its paces

ur job with the remote is pretty much done; all we need to do is run some tests and get some documentation together to describe the P. Home utomation or Bust, inc. sure is going to be impressed, don't you think? We've managed to come up with a design that is going to allow them to produce a remote that is easy to maintain and they're going to have no trouble convincing the vendors to write some simple command classes in the future since they are so easy to write.

Let's get to testing this code

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan = new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
            new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
            new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn =
            new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff =
            new LightOffCommand(kitchenLight);

        CeilingFanOnCommand ceilingFanOn =
            new CeilingFanOnCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
            new CeilingFanOffCommand(ceilingFan);

        GarageDoorUpCommand garageDoorUp =
            new GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown =
            new GarageDoorDownCommand(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
            new StereoOnWithCDCommand(stereo);
        StereoOffCommand stereoOff =
            new StereoOffCommand(stereo);
    }
}
```

The diagram illustrates the creation of objects and their corresponding commands. It uses curly braces to group related code and handwritten notes to explain each group:

- Create all the devices in their proper locations.**: Groups the declarations of `Light`, `CeilingFan`, `GarageDoor`, and `Stereo`.
- Create all the Light Command objects.**: Groups the declarations of `LightOnCommand` and `LightOffCommand` for both the `livingRoomLight` and `kitchenLight`.
- Create the On and Off for the ceiling fan.**: Groups the declarations of `CeilingFanOnCommand` and `CeilingFanOffCommand`.
- Create the Up and Down commands for the Garage.**: Groups the declarations of `GarageDoorUpCommand` and `GarageDoorDownCommand`.
- Create the stereo On and Off commands.**: Groups the declarations of `StereoOnWithCDCommand` and `StereoOffCommand`.

Now, let's check out the execution of our remote control test...

```
File Edit Window Help Command GetT in Done

% java RemoteLoader
----- Remote Control -----
[slot 0] headfirst.command.remote.LightOnCommand
[slot 1] headfirst.command.remote.LightOnCommand
[slot 2] headfirst.command.remote.CeilingFanOnCommand
[slot 3] headfirst.command.remote.StereoOnWithCDCommand
[slot 4] headfirst.command.remote.NoCommand
[slot 5] headfirst.command.remote.NoCommand
[slot 6] headfirst.command.remote.NoCommand

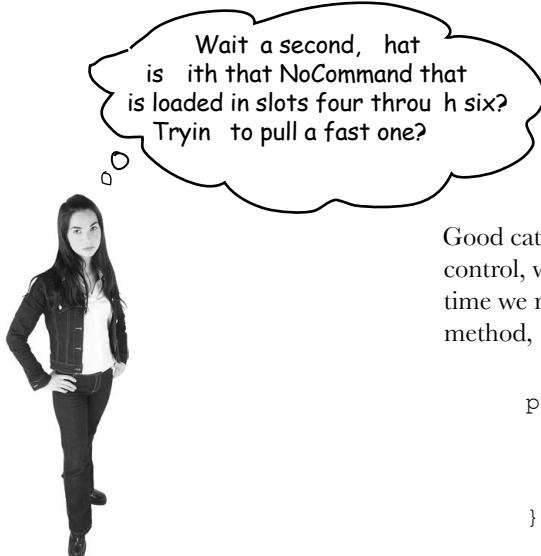
Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off

% headfirst.command.remote.LightOffCommand
headfirst.command.remote.LightOffCommand
headfirst.command.remote.CeilingFanOffCommand
headfirst.command.remote.StereoOffCommand
headfirst.command.remote.NoCommand
headfirst.command.remote.NoCommand
headfirst.command.remote.NoCommand
```

↑  
On slots

←  
Off Slots

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living Room light is on."



Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the `onButtonWasPushed()` method, we would need code like this

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

So, how do we get around that? Implement a command that does nothing

```
public class NoCommand implements Command {
    public void execute() { }
}
```

Then, in our `remoteControl` constructor, we assign every slot a `NoCommand` object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

Now in the output of our test run, you are seeing slots that haven't been assigned to a command, other than the default `NoCommand` object which we assigned when we created the `remoteControl`.



## Pattern on a Le

The `NoCommand` object is an example of a *null object*. *null object* is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling `null` from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a `NoCommand` object that acts as a surrogate and does nothing when its `execute` method is called.

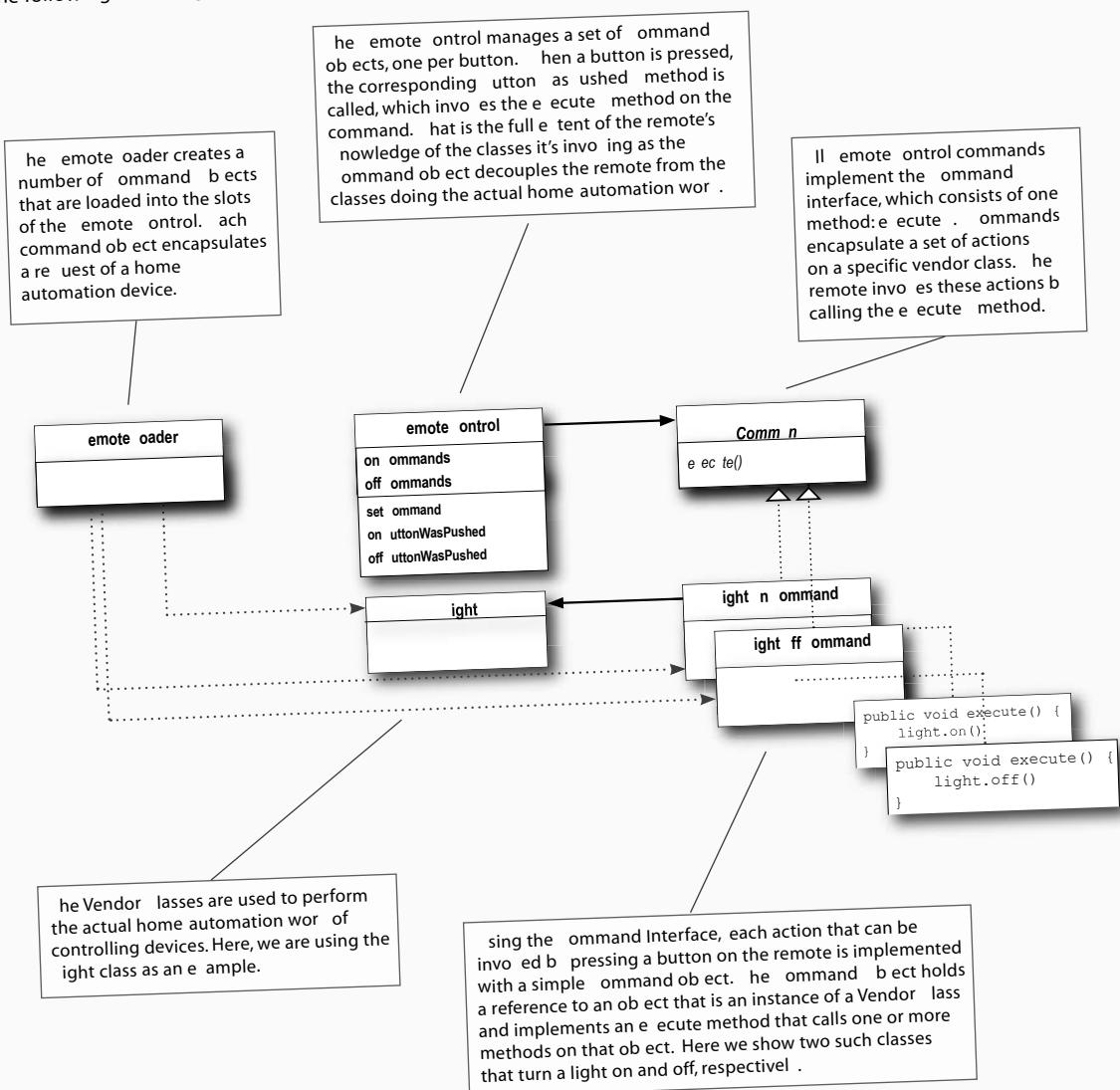
You'll find uses for `null` objects in conjunction with many Design patterns and sometimes you'll even see `null` object listed as a Design pattern.

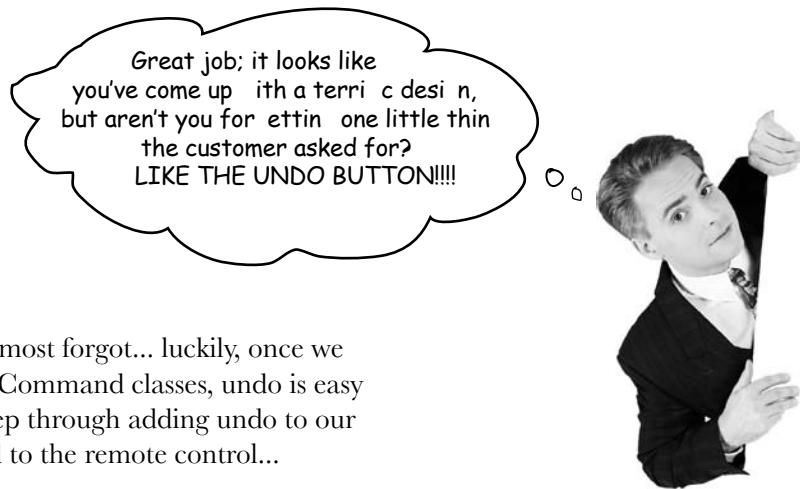
# Time to write that documentation...

## remote Control PI esign for Home utomation or ust, Inc.

We are pleased to present you with the following design and application programming interface for our Home automation remote control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the command pattern to logically decouple the remote control class from the Vendor classes. We believe this will reduce the cost of producing the remote as well as drastically reduce our ongoing maintenance costs.

The following class diagram provides an overview of our design:





Whoops We almost forgot... luckily, once we have our basic Command classes, undo is easy to add. Let's step through adding undo to our commands and to the remote control...

## What are we doing?

Okay, we need to add functionality to support the undo button on the remote. It works like this say the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button then the last action will be reversed in this case the light will turn off. Before we get into more complex examples, let's get the light working with the undo button

- When commands support undo, they have an `undo()` method that mirrors the `execute()` method. Whatever `execute()` last did, `undo()` reverses. So, before we can add undo to our commands, we need to add an `undo()` method to the Command interface

```
public interface Command {  
    public void execute();  
    public void undo();  
}
```

Here's the new `undo()` method.

That was simple enough.

Now, let's dive into the Light command and implement the `undo()` method.

- 2 Let's start with the LightOnCommand if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() method.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

*execute() turns the light on, so undo() simply turns the light back off.*

Piece of cake now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

*And here, undo() turns the light back on!*

Could this be any easier? Okay, we aren't done yet; we need to work a little support into the Remote Control to handle tracking the last button pressed and the undo button press.

- 3 add support for the undo button we only have to make a few small changes to the RemoteControl class. Here's how we're going to do it - we'll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.

```

public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // toString code here...
    }
}

```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

# Time to A that Undo button

kay, let's rework the test harness a bit to test the undo button

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room"); ← Create a Light, and our new undo()
        ← enabled Light On and Off Commands.

        LightOnCommand livingRoomLightOn =
            new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
            new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

← Add the light Commands to the remote in slot .  
 Turn the light on, then off and then undo.  
 Then, turn the light off, back on and undo.

nd here's the test results...

```
File Edit Window Help UndoCommand Deployment
% java RemoteLoader
Light is on ← Turn the light on, then off.
Light is off
----- Remote Control -----
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOffCommand ← Here's the Light commands.
← Now undo holds the LightOffCommand, the last command invoked.

Light is on ← Undo was pressed... the LightOffCommand undo() turns the light back on.
Light is off ← Then we turn the light off then back on.

----- Remote Control -----
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand          headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOnCommand ← Now undo holds the LightOnCommand, the last command invoked.

Light is off ← Undo was pressed, the light is back off.
```

~~e need to ee o e tate or undo~~

## Using state to implement Undo

Okay, implementing undo on the Light was instructive but a little too easy. Typically, we need to manage a bit of state to implement undo. Let's try something a little more interesting, like the CeilingFan from the vendor classes. The ceiling fan allows a number of speeds to be set along with an off method.

Here's the source code for the CeilingFan

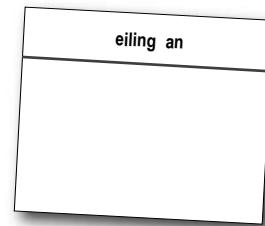
```
public class CeilingFan {  
    public static final int HIGH = 3;  
    public static final int MEDIUM = 2;  
    public static final int LOW = 1;  
    public static final int OFF = 0;  
    String location;  
    int speed;  
  
    public CeilingFan(String location) {  
        this.location = location;  
        speed = OFF;  
    }  
  
    public void high() {  
        speed = HIGH;  
        // code to set fan to high  
    }  
  
    public void medium() {  
        speed = MEDIUM;  
        // code to set fan to medium  
    }  
  
    public void low() {  
        speed = LOW;  
        // code to set fan to low  
    }  
  
    public void off() {  
        speed = OFF;  
        // code to turn fan off  
    }  
  
    public int getSpeed() {  
        return speed;  
    }  
}
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

Hmm, so to properly implement undo, I'd have to take the previous speed of the ceiling fan into account...

These methods set the speed of the ceiling fan.

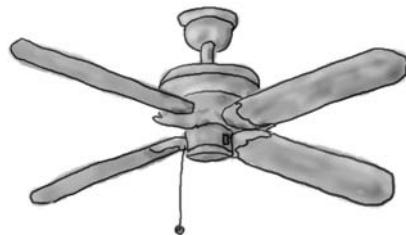
We can get the current speed of the ceiling fan using getSpeed().



ha ter

## Adding Undo to the ceiling fan commands

Now let's tackle adding undo to the various CeilingFan commands. To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting. Here's the code for the CeilingFanHighCommand



```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.



We've got three more ceiling fan command to write: low, medium, and off. Can you see how they are implemented?

te t the ei in an

## Get ready to test the ceiling fan

ime to load up our remote control with the ceiling fan commands. We're going to load slot zero's on button with the medium setting for the fan and slot one with the high setting. Both corresponding off buttons will hold the ceiling fan off command.

Here's our test script

```
public class RemoteLoader {  
  
    public static void main(String[] args) {  
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();  
  
        CeilingFan ceilingFan = new CeilingFan("Living Room");  
  
        CeilingFanMediumCommand ceilingFanMedium =  
            new CeilingFanMediumCommand(ceilingFan);  
        CeilingFanHighCommand ceilingFanHigh =  
            new CeilingFanHighCommand(ceilingFan);  
        CeilingFanOffCommand ceilingFanOff =  
            new CeilingFanOffCommand(ceilingFan);  
  
        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);  
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);  
  
        remoteControl.onButtonWasPushed(0); ← First, turn the fan on medium.  
        remoteControl.offButtonWasPushed(0); ← Then turn it off.  
        System.out.println(remoteControl);  
        remoteControl.undoButtonWasPushed(); ← Undo! It should go back to medium...  
  
        remoteControl.onButtonWasPushed(1); ← Turn it on to high this time.  
        System.out.println(remoteControl);  
        remoteControl.undoButtonWasPushed(); ← And, one more undo; it should go back to medium.  
    }  
}
```

Here we instantiate three commands: high, medium, and off.

Here we put medium in slot zero, and high in slot one. We also load up the off commands.

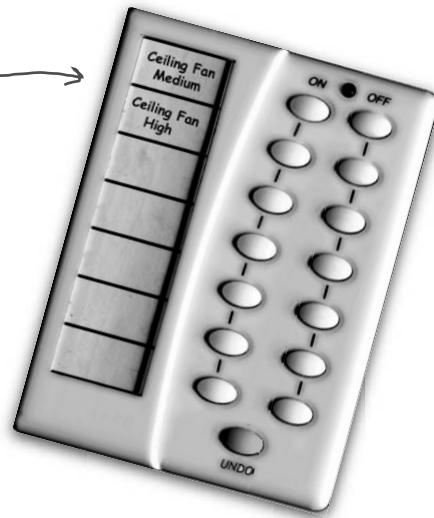
First, turn the fan on medium.

Then turn it off.

Undo! It should go back to medium...

Turn it on to high this time.

And, one more undo; it should go back to medium.



ha ter

# Testing the ceiling fan...

kay, let's fire up the remote, load it with commands, and push some buttons

```

File Edit Window Help Undo T i !

% java RemoteLoader

Living Room ceiling fan is on medium ← Turn the ceiling fan on
Living Room ceiling fan is off ← medium, then turn it off.

----- Remote Control -----
[slot 0] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanOffCommand ← Here are the commands
                                                in the remote control...
...and undo has the last
command executed, the
CeilingFanOfCommand.

Living Room ceiling fan is on medium ← Undo the last command, and it goes back to medium.
Living Room ceiling fan is on high ← Now, turn it on high.

----- Remote Control -----
[slot 0] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanHighCommand ← Now, high is the last
                                                command executed.

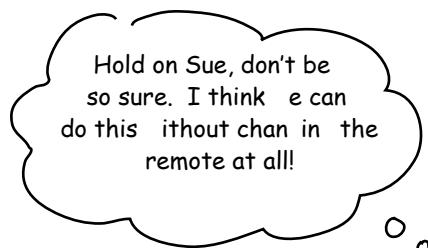
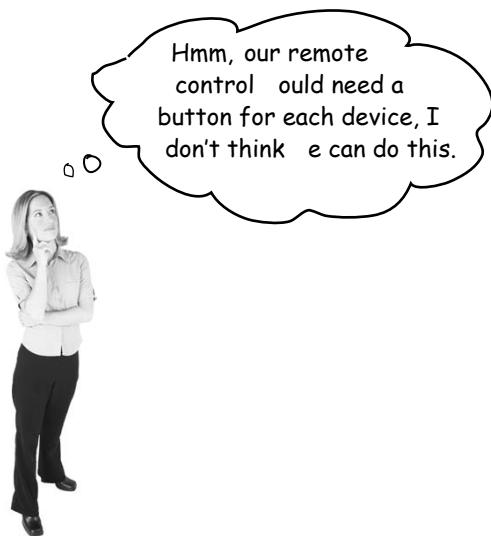
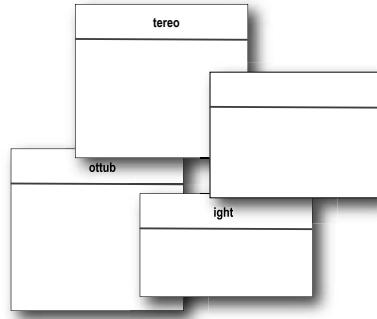
Living Room ceiling fan is on medium ← One more undo, and the ceiling fan
%                                         goes back to medium speed.

```

a ro o and

## Every remote needs a Party Mode

**hat's the point of having a re ote if you can't push one utton and have the lights di ed, the stereo and turned on and set to a D D and the hot tu red up?**



Mary's idea is to make a new kind of Command that can execute other Commands... and more than one of them! Pretty good idea, huh?

```
public class MacroCommand implements Command {  
    Command[] commands;
```

```
    public MacroCommand(Command[] commands) {  
        this.commands = commands;  
    }
```

Take an array of Commands and store them in the MacroCommand.

```
    public void execute() {  
        for (int i = 0; i < commands.length; i++) {  
            commands[i].execute();  
        }  
    }  
}
```

When the macro gets executed by the remote, execute those commands one at a time.

ha ter

# Using a macro command

Let's step through how we use a macro command

- First we create the set of commands we want to go into the macro

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();
```

Create all the devices, a light, tv, stereo, and hot tub.

```
LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Now create all the On commands to control them.



ew ll also eed comma ds or the o b tto s wr te the code to create those here:

- et we create two arrays, one for the on commands and one for the off commands, and load them with the corresponding commands

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

Create an array for On and an array for Off commands...

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...and create two corresponding macros to hold them.

- hen we assign MacroCommand to a button like we always do

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Assign the macro command to a button as we would any command.

*you are here ▶*

- Finally, we just need to push some buttons and see if this works.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

Here's the output.

```
File Edit Window Help o Can tBeatABa a
% java RemoteLoader
----- Remote Control -----
[slot 0] headfirst.command.party.MacroCommand
[slot 1] headfirst.command.party.NoCommand
[slot 2] headfirst.command.party.NoCommand
[slot 3] headfirst.command.party.NoCommand
[slot 4] headfirst.command.party.NoCommand
[slot 5] headfirst.command.party.NoCommand
[slot 6] headfirst.command.party.NoCommand
[undo] headfirst.command.party.NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees

%
```

Here are the two macro commands.

All the Commands in the macro are executed when we invoke the on macro...

and when we invoke the off macro. Looks like it works.



he only thing our MacroCommand is missing its undo functionality. When the undo button is pressed after a macro command, all the commands that were invoked in the macro must undo their previous actions. Here's the code for MacroCommand; go ahead and implement the undo() method

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        // Implementation goes here
    }
}
```

**Q:** Do I always need a receiver?  
Why can't the command objects implement the details of the execute method?

**A:** In general, we strive for dumb command objects that just invoke an action on a receiver; however, there are many examples of smart command objects that implement most, if not all, of the logic needed to carry out a request. Certainly you can do this just keep in mind you'll no longer have the same level of decoupling between the invoker and receiver, nor will you be able to parameterize your commands with receivers.

## There are no Dumb Questions

**Q:** How can I implement a history of undo operations? In other words, I want to be able to press the undo button multiple times.

**A:** Great question! It's pretty easy actually instead of keeping just a reference to the last command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo method.

**Q:** Could I have just implemented Party code as a Command by creating a PartyCommand and putting the calls to execute the other Commands in the PartyCommands execute method?

**A:** You could, however, you'd essentially be hardcoding the party mode into the art command. If you go to the trouble with the Macro command, you can decide dynamically which commands you want to go into the art command, so you have more flexibility using Macro commands. In general, the Macro command is a more elegant solution and requires less new code.

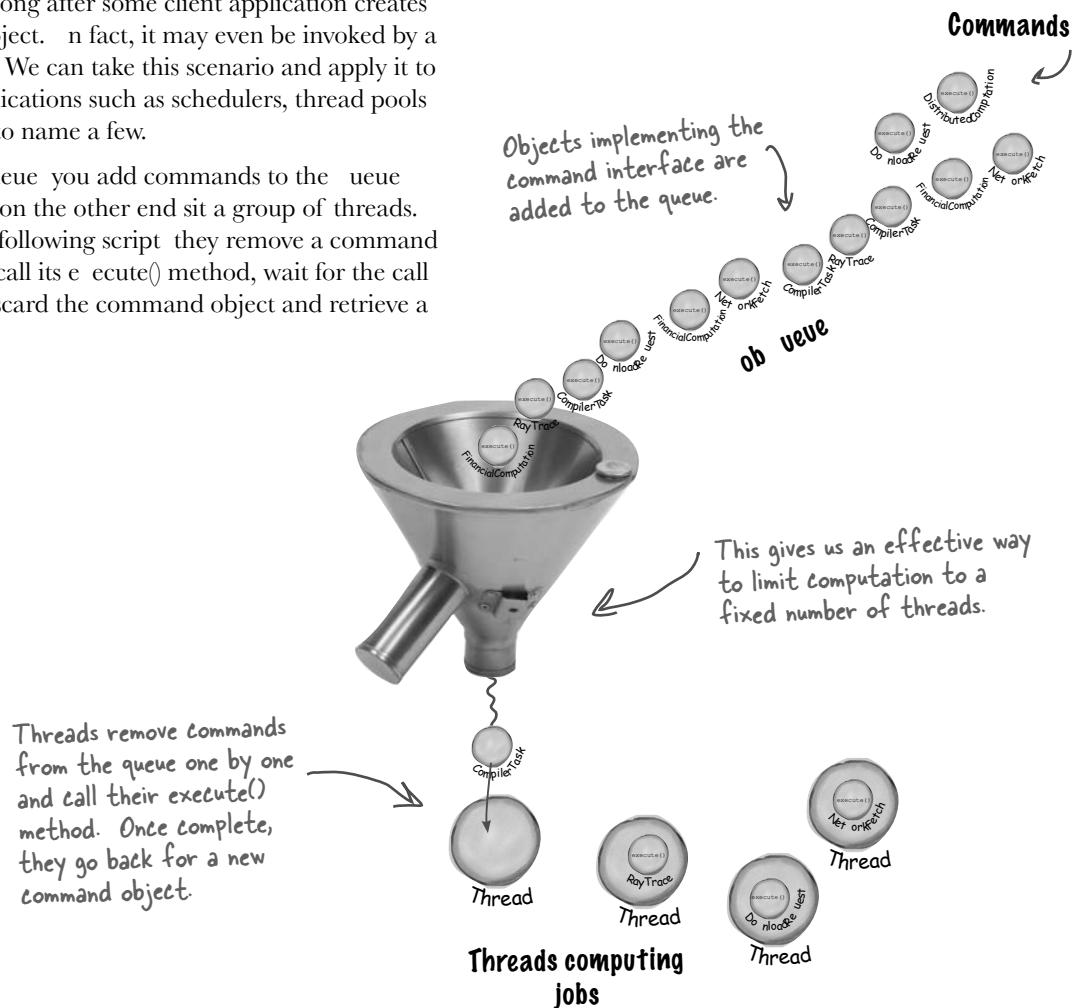
*you are here ▶*

## More uses of the Command Pattern: using re uests

Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first class object. Now, the computation itself may be invoked long after some client application creates the command object. In fact, it may even be invoked by a different thread. We can take this scenario and apply it to many useful applications such as schedulers, thread pools and job queues, to name a few.

Imagine a job queue you add commands to the queue on one end, and on the other end sit a group of threads.

Threads run the following script they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.



Note that the job queue classes are totally decoupled from the objects that are doing the computation. One minute a thread may be computing a financial computation, and the next it may be retrieving something from the network. The job queue objects don't care; they just retrieve commands and call execute(). Likewise, as long as you put objects into the queue that implement the Command Pattern, your execute() method will be invoked when a thread is available.

ha ter



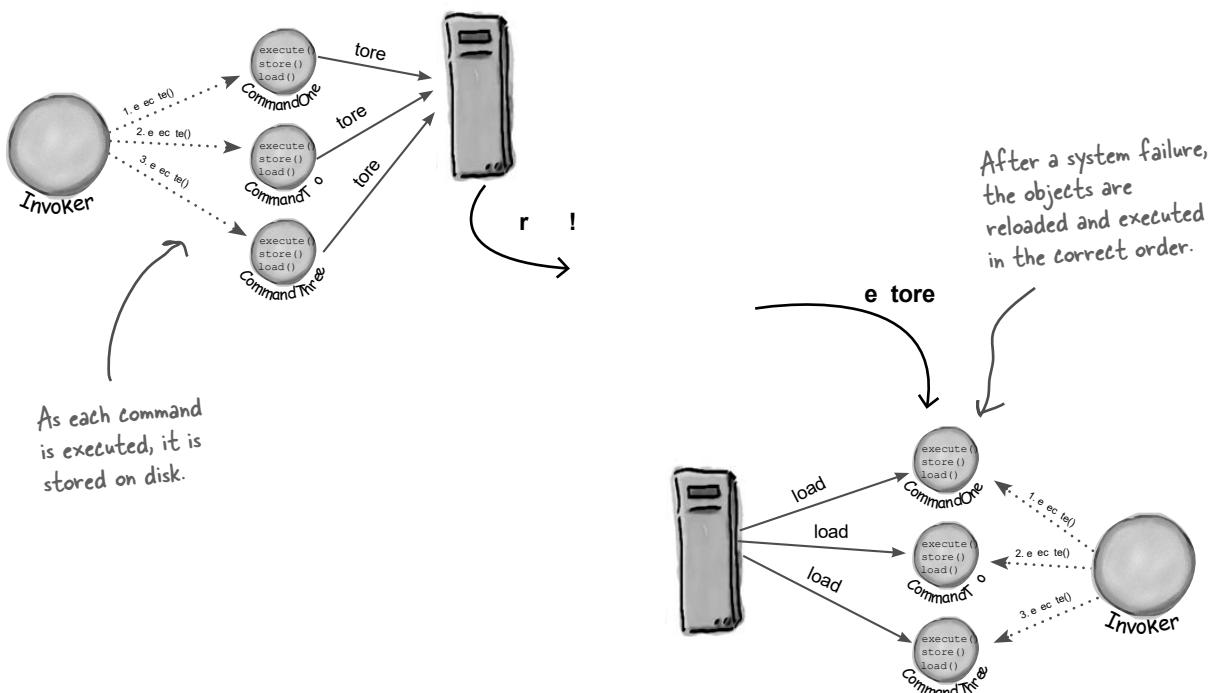
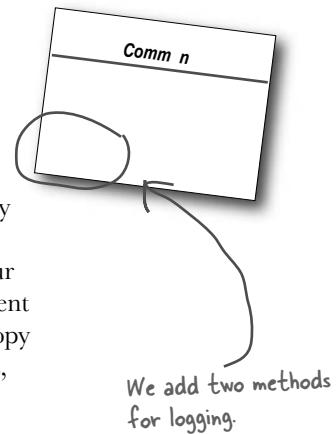
How mi t a we er er ma e  
eo c a ee? W at ot er  
application can yo t in o ?

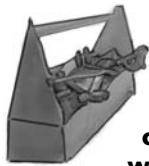
## More uses of the Command Pattern: logging requests

The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions. The Command Pattern can support these semantics with the addition of two methods `store()` and `load()`. In Java we could use object serialization to implement these methods, but the normal caveats for using serialization for persistence apply.

How does this work? As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their `execute()` methods in batch and in order.

Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last check point, and if there is a system failure, apply those operations to our checkpoint. Take, for example, a spreadsheet application we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs. In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.





# Tools for your Design Toolbox

our tool is starting to get heavy in this chapter we've added a pattern that allows us to encapsulate methods into objects store them, pass them around, and invoke them when you need them



**OO Principles**

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions. Do not depend on concrete classes.

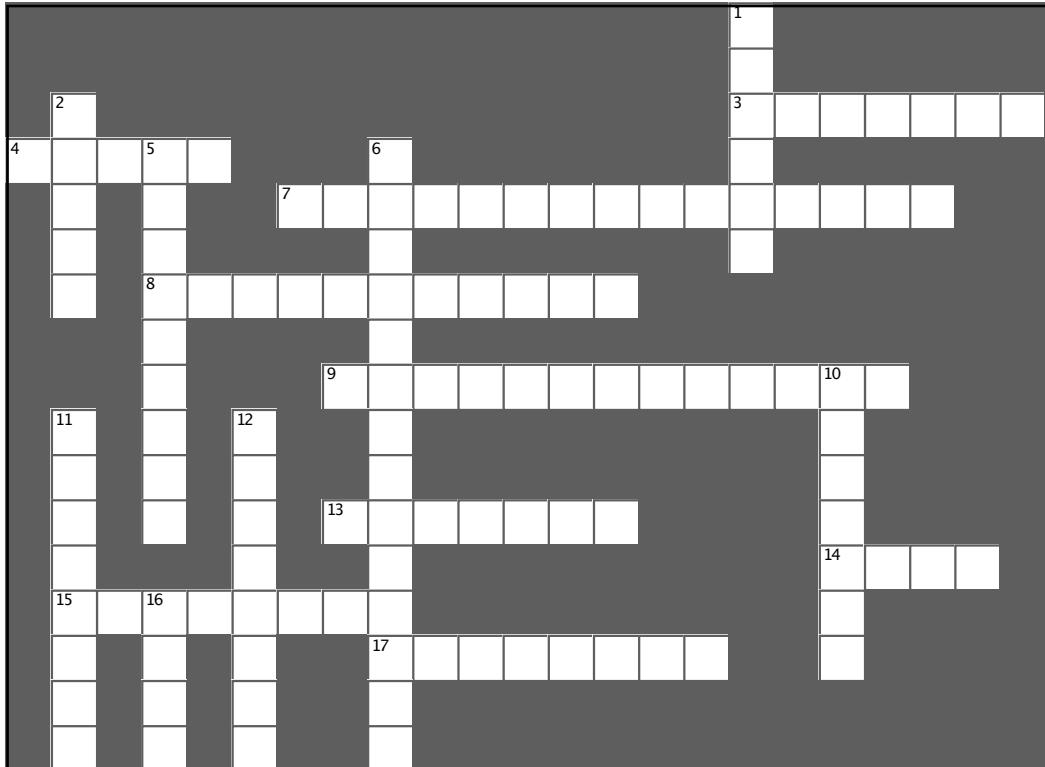
**OO Basics**

- straction
- apsulation
- ymorphism
- eritance

**OO Patterns**

- Strategy
- Decorator
- Adapter
- Factory Method
- Singleton
- Composite
- Chain of Responsibility
- Mediator
- Visitor

**Command** - Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



### Across

3. The Waitress was one
4. A command \_\_\_\_ a set of actions and a receiver
7. Dr. Seuss diner food
8. Our favorite city
9. Act as the receivers in the remote control
13. Object that knows the actions and the receiver
14. Another thing Command can do
15. Object that knows how to get things done
17. A command encapsulates this

### Down

1. Role of customer in the command pattern
2. Our first command object controlled this
5. Invoker and receiver are \_\_\_\_\_
6. Company that got us word of mouth business
10. All commands provide this
11. The cook and this person were definitely decoupled
12. Carries out a request
16. Waitress didn't do this



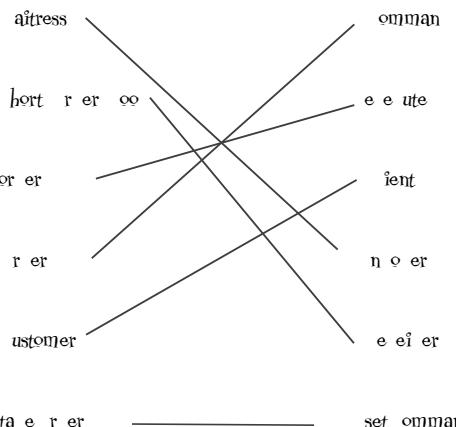
## er ise solutions



Match the diner objects and methods with the corresponding names from the Command Pattern

### Diner

### o and Pattern



```
public class GarageDoorOpenCommand implements Command {  
    GarageDoor garageDoor;  
    public GarageDoorOpenCommand(GarageDoor garageDoor) {  
        this.garageDoor = garageDoor;  
    }  
  
    public void execute() {  
        garageDoor.up();  
    }  
}
```

```
File Edit Window Help GreenE  &Ham  
%java RemoteControlTest  
Light is on  
Garage Door is Open  
%
```



# er ise solutions



Write the undo() method for MacroCommand

```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

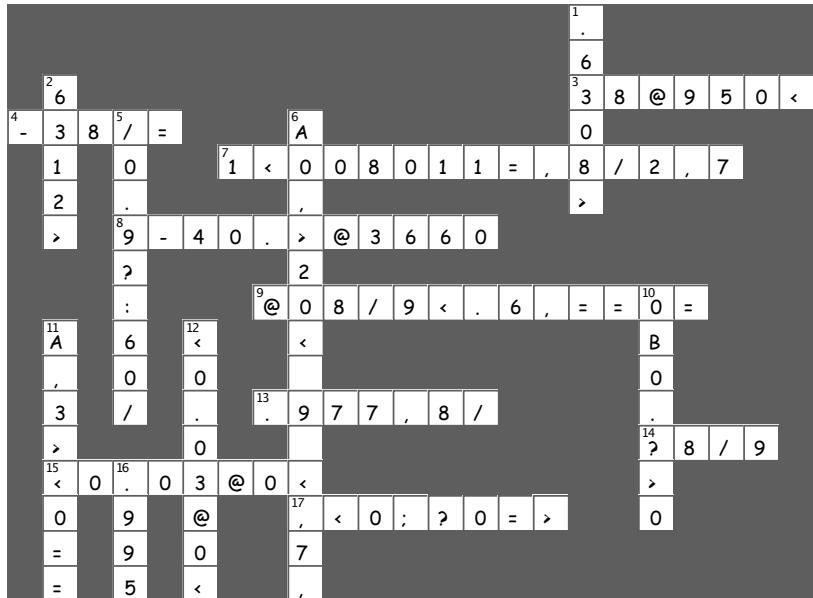
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].undo();
        }
    }
}
```



**e w ll also eed comma nds or the o b tto .  
r te the code to create those here:**

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```





the A a ter an a a e atterns

## eing daptive



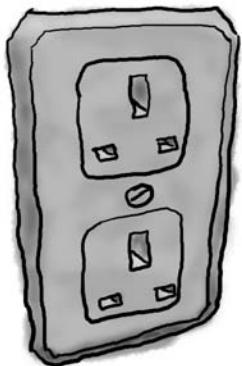
So and impo i le? Not w en we a e  
De i n Pattern . Remem er t e Decorator Pattern? We r e o e t to i et em new  
re pon i litie . Now we re oin to wrap ome o ect wit a di erent p roje: to ma e t eir  
inter ace loo li e omet in t ey're not. W y wo ld we do t at? So we can adapt a de i n  
e pectin one inter ace to a cla t at implement a di erent inter ace. T at not all; w ile  
we re at it, we re oin to loo at anot er pattern t at wrap o ect to impli yt eir inter ace.

## Adapters all around us

o ll eno tro le n er t n n t n ter  
e et e re l orl ll o te o t or ne le  
e o e er nee e to e el to n ro e n  
o ntr en o e ro l nee e n o er ter



European Wall Outlet

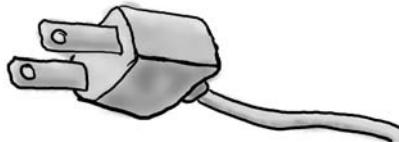


The European wall outlet exposes one interface for getting power.

AC Power Adapter



Standard AC Plug



The US laptop expects another interface.



The adapter converts one interface into another.

You know what the adapter does: it sits in between the plug of your laptop and the European AC outlet; its job is to adapt the European outlet so that you can plug your laptop into it and receive power. Or look at it this way: the adapter changes the interface of the outlet into one that your laptop expects.

Some AC adapters are simple: they only change the shape of the outlet so that it matches your plug, and they pass the AC current straight through. But other adapters are more complex internally and may need to step the power up or down to match your devices' needs.

Okay, that's the real world, what about object-oriented adapters? Well, our adapters play the same role as their real-world counterparts: they take an interface and adapt it to one that a client is expecting.

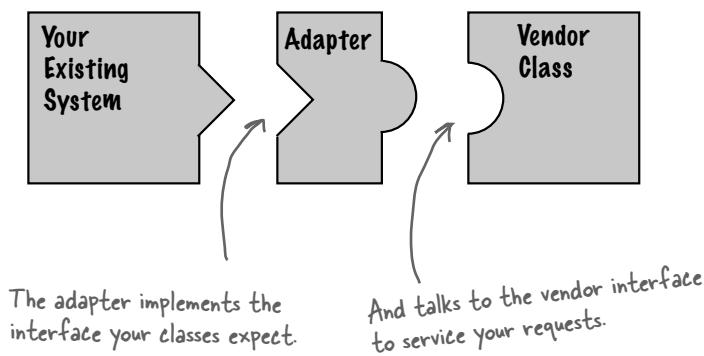
How many other real world adapters can you think of?

# Object oriented adapters

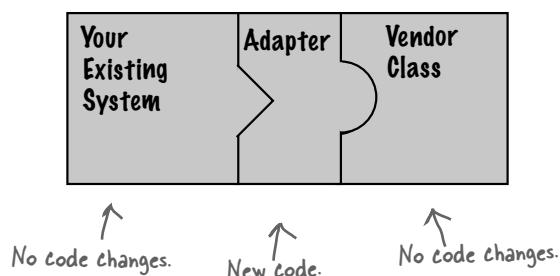
ay you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor



kay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



he adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

If it walks like a duck and quacks like a duck,  
then it must be a duck turkey wrapped  
with a duck adapter...

It's time to see an adapter in action earlier our  
ducks from chapter ? Let's review a slightly simplified  
version of the Duck interfaces and classes



```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Here's a subclass of Duck, the MallardDuck

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
    public void fly() {  
        System.out.println("I'm flying");  
    }  
}
```

Simple implementations: the duck just prints out what it is doing.

Now it's time to meet the newest fowl on the block

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}

```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

**Now, let's say you're short on Duck objects and you'd like to use some turkey objects in their place. Obviously we can't use the turkeys outright because they have a different interface.**

So, let's write an adapter.



## Code Up Close

```

public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}

```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

te t the ada ter

# Test drive the adapter

Now we just need some code to test drive our adapter

```
public class DuckTestDrive {  
    public static void main(String[] args) {  
        MallardDuck duck = new MallardDuck(); ← Let's create a Duck...  
  
        WildTurkey turkey = new WildTurkey(); ← and a Turkey.  
        Duck turkeyAdapter = new TurkeyAdapter(turkey); ← And then wrap the turkey  
                                                       in a TurkeyAdapter, which  
                                                       makes it look like a Duck.  
  
        System.out.println("The Turkey says...");  
        turkey.gobble(); ← Then, let's test the Turkey:  
                           make it gobble, make it fly.  
        turkey.fly();  
  
        System.out.println("\nThe Duck says...");  
        testDuck(duck);  
  
        System.out.println("\nThe TurkeyAdapter says...");  
        testDuck(turkeyAdapter); ← Now let's test the duck  
                               by calling the testDuck()  
                               method, which expects a  
                               Duck object.  
    }  
  
    static void testDuck(Duck duck) {  
        duck.quack(); ← Here's our testDuck() method; it  
        duck.fly();   gets a duck and calls its quack()  
    }             and fly() methods.  
}
```

Test run

```
File Edit Window Help Don'tFor etToD c  
%java RemoteControlTest  
The Turkey says...  
Gobble gobble  
I'm flying a short distance  
  
The Duck says...  
Quack  
I'm flying  
  
The TurkeyAdapter says...  
Gobble gobble  
I'm flying a short distance  
I'm flying a short distance
```

← The Turkey gobbles and flies a short distance.

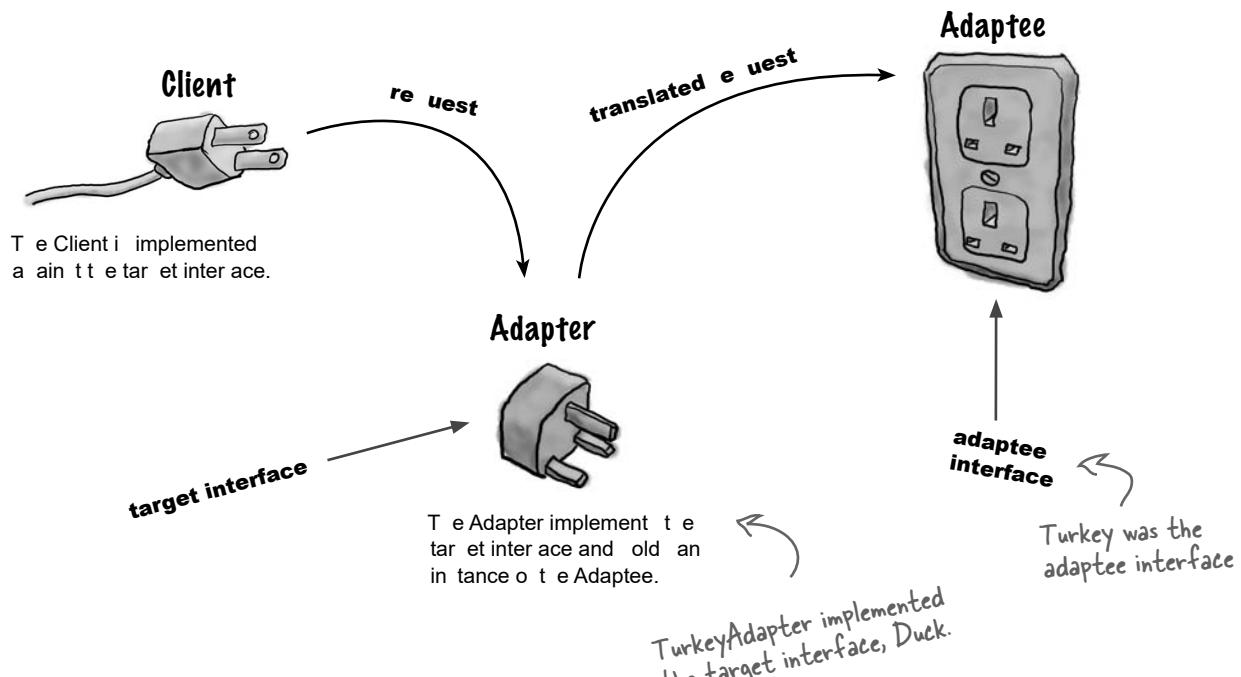
← The Duck quacks and flies just like you'd expect.

← And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

ha ter

# The Adapter Pattern explained

Now that we have an idea of what an adapter is, let's step back and look at all the pieces again.



## Here's how the Client uses the Adapter

- 1 The client makes a request to the adapter by calling a method on it using the target interface**
- 2 The adapter translates the request into one or more calls on the adaptee using the adaptee interface**
- 3 The client receives the results of the call and never knows there is an adapter doing the translation**

Note that the Client and Adaptee are decoupled – neither knows about the other.

you are here ▶



## Sharpen your pencil

Let say we also need an Adapter that converts a DC to a Trey. Let call it DC Adapter. Write that class:

How did you handle the legacy method (after all we now have a dependency on the target system)? Could there be another way?

*there are no  
Dumb Questions*

**Q:** How much adapting does an adapter need to do? It seems like if I need to implement a large target interface I could have a lot of work on my hands

**A:**

**Q:** Does an adapter always wrap one and only one class?

**A:**

**Q:** What if I have old and new parts of my system, the old parts expect the old vendor interface but we've already written the new parts to use the new vendor interface? It is going to get confusing using an adapter here and the unwrapped interface there. Wouldn't it be better off just writing my older code and forgetting the adapter?

**A:**

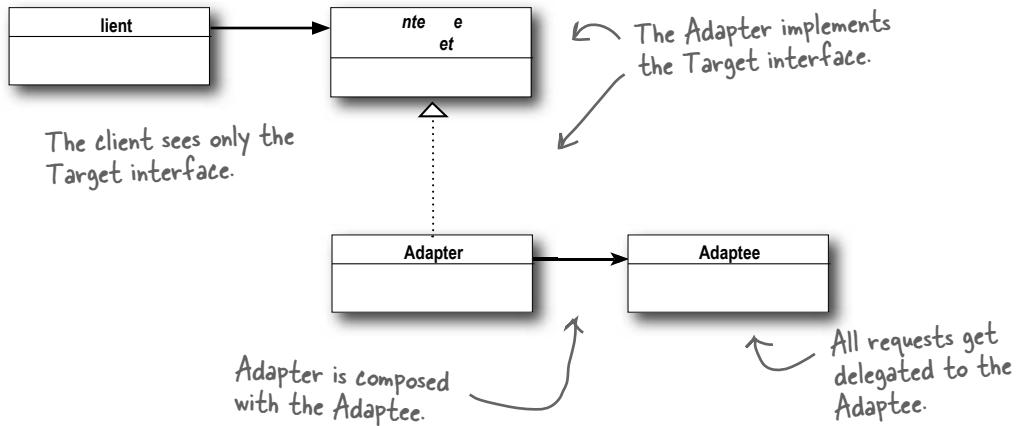
# Adapter Pattern defined

nough ducks, turkeys and C power adapters; let's get real and look at the official definition of the Adapter Pattern

**The Adapter pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well



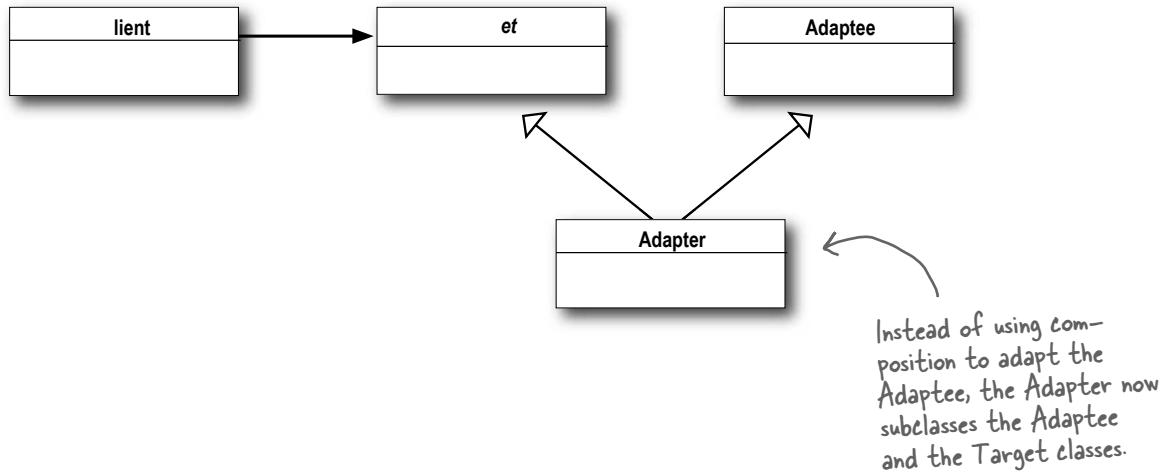
The Adapter Pattern is full of good design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Also check out how the pattern binds the client to an interface, not an implementation; we could use several adapters, each converting a different backend set of classes. Furthermore, we could add new implementations after the fact, as long as they adhere to the target interface.

## Object and class adapters

ow despite having defined the pattern, we haven't told you the whole story yet. There are actually two kinds of adapters: object adapters and class adapters. This chapter has covered object adapters and the class diagram on the previous page is a diagram of an object adapter.

So what's a class adapter and why haven't we told you about it? Because you need multiple inheritance to implement it, which isn't possible in Java. But, that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language. Let's look at the class diagram for multiple inheritance.



Look familiar? That's right: the only difference is that with class adapter we subclass the target and the adaptee, while with object adapter we use composition to pass requests to an adaptee.

 RA POWER

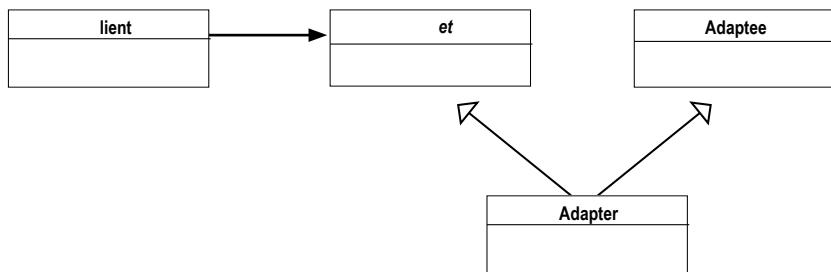
Object adapter and class adapter are two different means of adapting an interface to an adaptee (composition or inheritance). How do they differ in implementation?



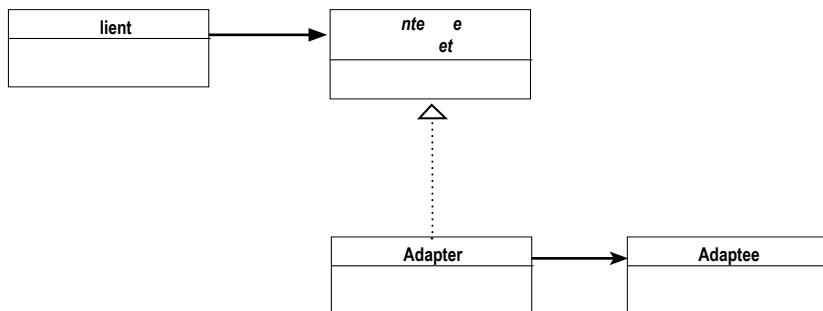
## u agnets

our job is to take the duck and turkey magnets and drag them over the part of the diagram that describes the role played by that bird, in our earlier example. (try not to flip back through the pages.) Then add your own annotations to describe how it works.

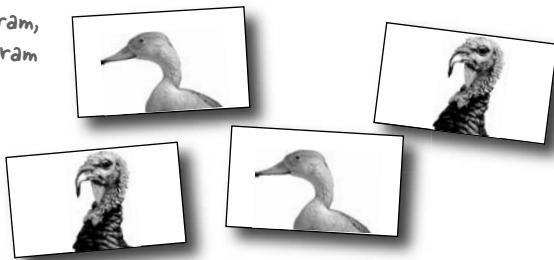
### class adapter



### struct adapter



Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.



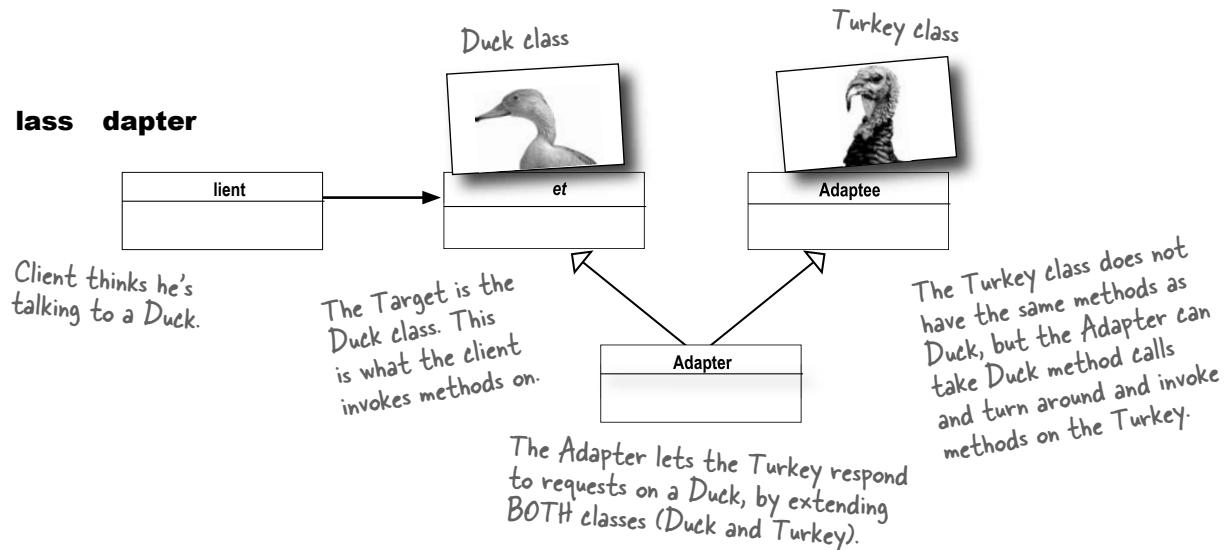
*you are here ▶*



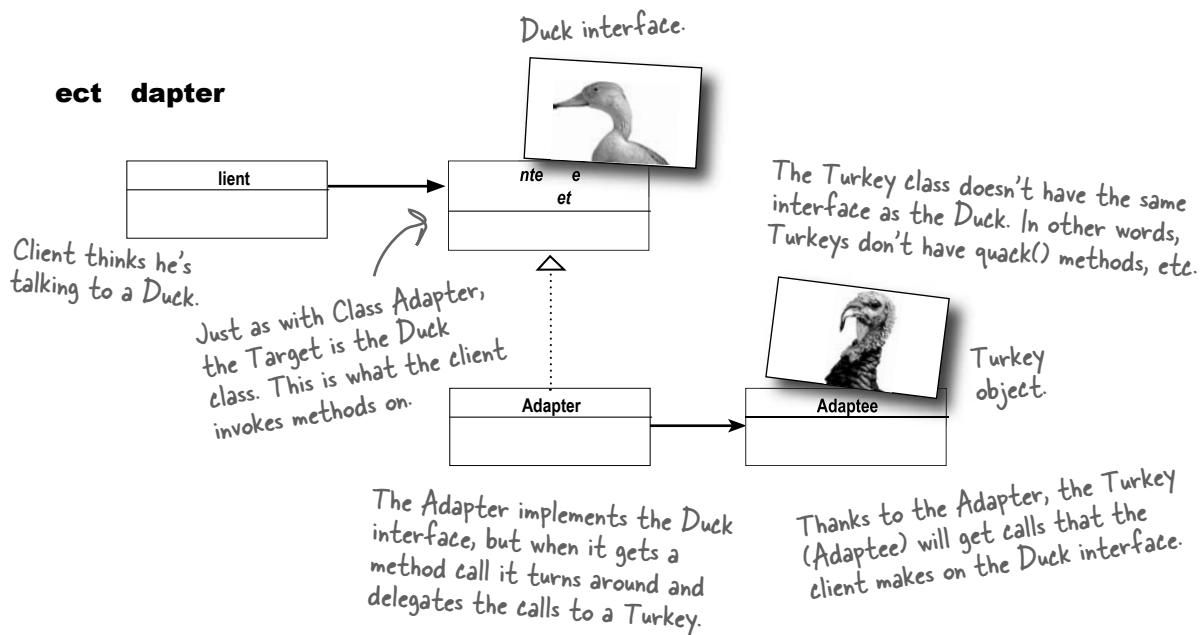
## u ns er agnets

Note: the class adapter uses multiple inheritance, so you can't do it in Java...

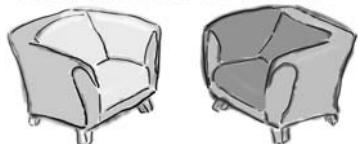
### lass dapter



### ect dapter



## Fireside Chats



Tonight's talk: **The Object Adapter and Class Adapter meet face to face.**

### Object Adapter

Because I use composition I've got a leg up. I can not only adapt an adaptee class, but any of its subclasses.

In my part of the world, we like to use composition over inheritance; you may be saving a few lines of code, but all I'm doing is writing a little code to delegate to the adaptee. We like to keep things flexible.

You're worried about one little object? You might be able to quickly override a method, but any behavior I add to my adapter code works with my adaptee class — all its subclasses.

Hey, come on, cut me a break, I just need to compose with the subclass to make that work.

You wanna see messy? Look in the mirror

### Class Adapter

That's true, I do have trouble with that because I am committed to one specific adaptee class, but I have a huge advantage because I don't have to reimplement my entire adaptee. I can also override the behavior of my adaptee if I need to because I'm just subclassing.

Flexible maybe, efficient? No. Using a class adapter there is just one of me, not an adapter and an adaptee.

Yeah, but what if a subclass of adaptee adds some new behavior? Then what?

Sounds messy...

*you are here ▶*

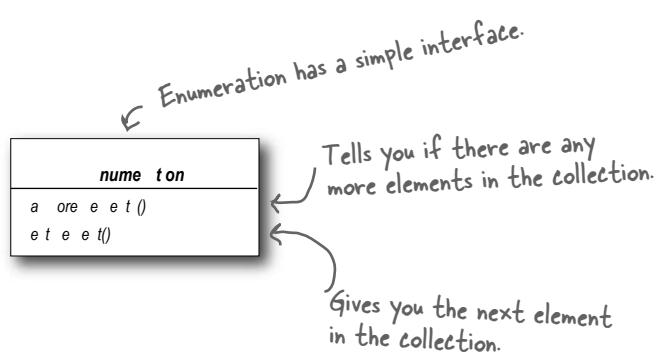
# Real world adapters

Let's take a look at the use of a simple adapter in the real world (something more serious than trucks at least)...

## Old world Enumerators

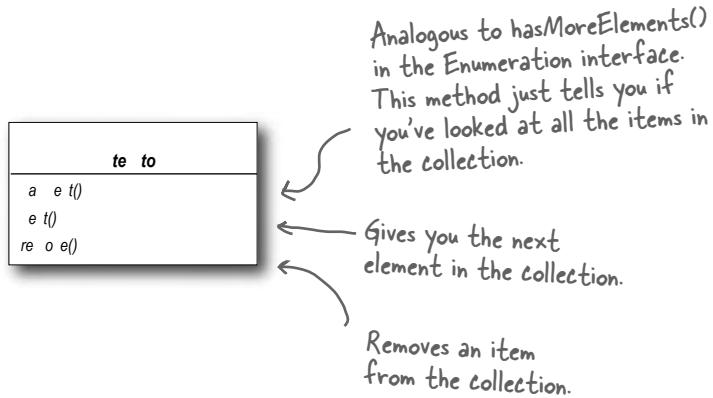
If you've been around Java for a while you probably remember that the early collections types (Vector, Stack, Hashtable, and a few others) implement a method `elements()`, which returns an Enumeration.

The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



## New world Iterators

When Sun released their more recent Collections classes they began using an Iterator interface that, like Enumeration, allows you to iterate through a set of items in a collection, but also adds the ability to remove items.

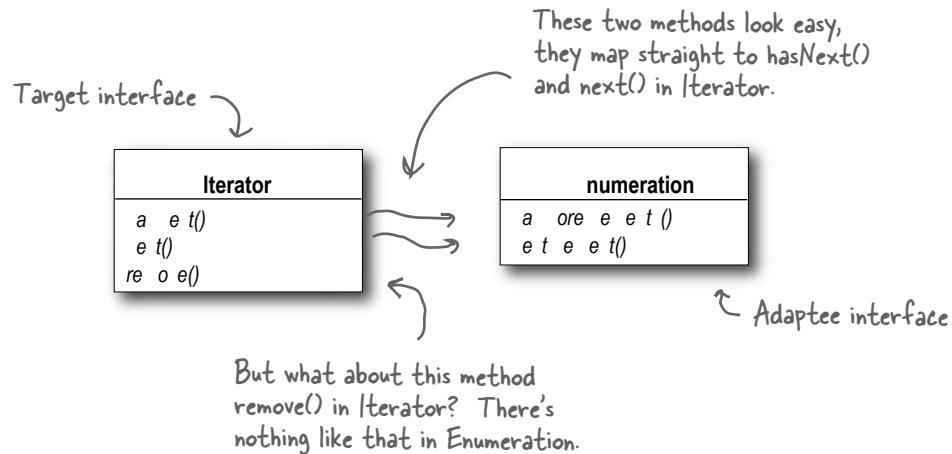


## And today...

We are often faced with legacy code that uses the Enumerator interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

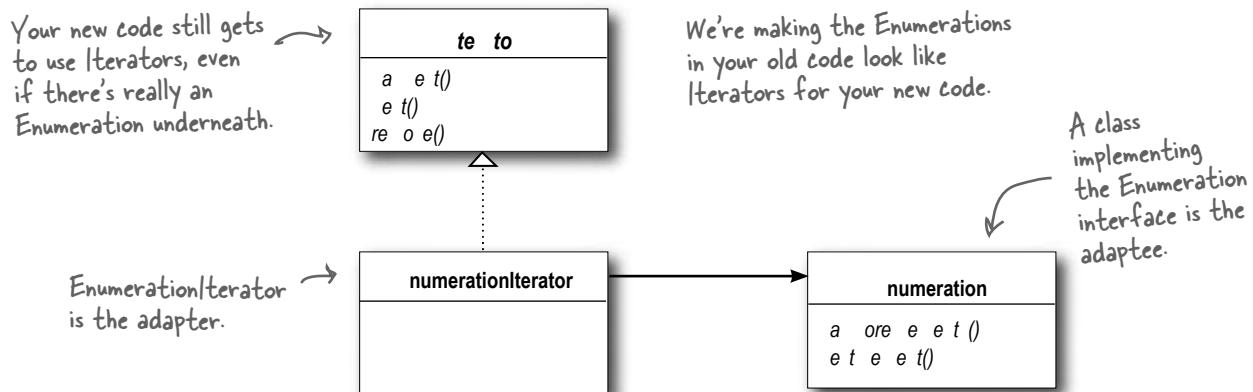
# Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



## Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the `Iterator` interface and is composed with an adaptee. The `next()` and `hasNext()` methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about `remove()`? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram.



## Dealing with the remove method

Well, we know enumeration just doesn't support remove. It's a read only interface.

There's no way to implement a fully functioning remove() method on the adapter. The best we can do is throw a runtime exception. Luckily, the designers of the iterator interface foresaw this need and defined the remove() method so that it supports an unsupported operation exception.

This is a case where the adapter isn't perfect; clients will have to watch out for potential exceptions, but as long as the client is careful and the adapter is well documented this is a perfectly reasonable solution.

## Writing the EnumerationIterator adapter

Here's simple but effective code for all those legacy classes still producing enumerations

```
public class EnumerationIterator implements Iterator {
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface... it has to look like an Iterator.

The Enumeration we're adapting. We're using composition so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

... and the Iterator's next() method is delegated to the Enumeration's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.



While Java has gone in the direction of the iterator, there is nevertheless a lot of legacy **client code** that depends on the Enumeration interface, so an adapter that converts an iterator to an Enumeration is also quite useful.

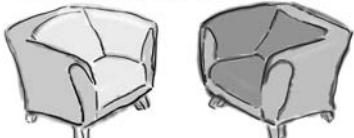
Write an adapter that adapts an iterator to an Enumeration. You can test your code by adapting an ArrayList. The ArrayList class supports the iterator interface but doesn't support enumerations (well, not yet anyway).



Some AC adapters do more than just an Enumeration interface; they add overcurrent protection, indicator lights, and other bells and whistles.

If you were going to implement the interface, what pattern would you use?

## Fireside Chats



Tonight's talk: **The Decorator pattern and the Adapter pattern discuss the differences.**

### Decorator

'm important. My job is all about *res si i ty* you know that when a ecorator is involved there's going to be some new responsibilities or behaviors added to your design.

hat may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.

Cute. I don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the re quest.

### Adapter

ou guys want all the glory while us adapters are down in the trenches doing the dirty work converting interfaces. Our jobs may not be glamorous, but our clients sure do appreciate us making their lives simpler.

ry being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying an uncoupled client is a happy client.

Hey, if adapters are doing their job, our clients never even know we're there. It can be a thank less job.

## **Decorator**

Well us decorators do that as well, only we allow *new code* to be added to classes without altering existing code. I still say that adapters are just fancy decorators. I mean, just like us, you wrap an object.

No, our job in life is to extend the behaviors or responsibilities of the objects we wrap, we aren't a *sim e ss t r g*.

Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *mis* apart in our *int*e*t*.

## **Adapter**

But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *yo* code, they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.

No, no, no, not at all. We *do*s convert the interface of what we wrap, you *ever* do. I'd say a decorator is like an adapter; it is just that you don't change the interface

Hey, who are you calling a simple pass through? Come on down and we'll see how long *yo* last converting a few interfaces

No yeah, I'm with you there.

*ho doe*    *hat*

## And now for something different...

### **here's another pattern in this chapter**

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.

We're going to look at a pattern now that alters an interface, but for a different reason to simplify the interface. It's aptly named the Facade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well lit facade.



Match each pattern with its intent

<b>Pattern</b>	<b>Intent</b>
Decorator	Implements one interface to another
Adapter	Doesn't alter the interface but assumes responsibility
Facade	Creates an interface similar

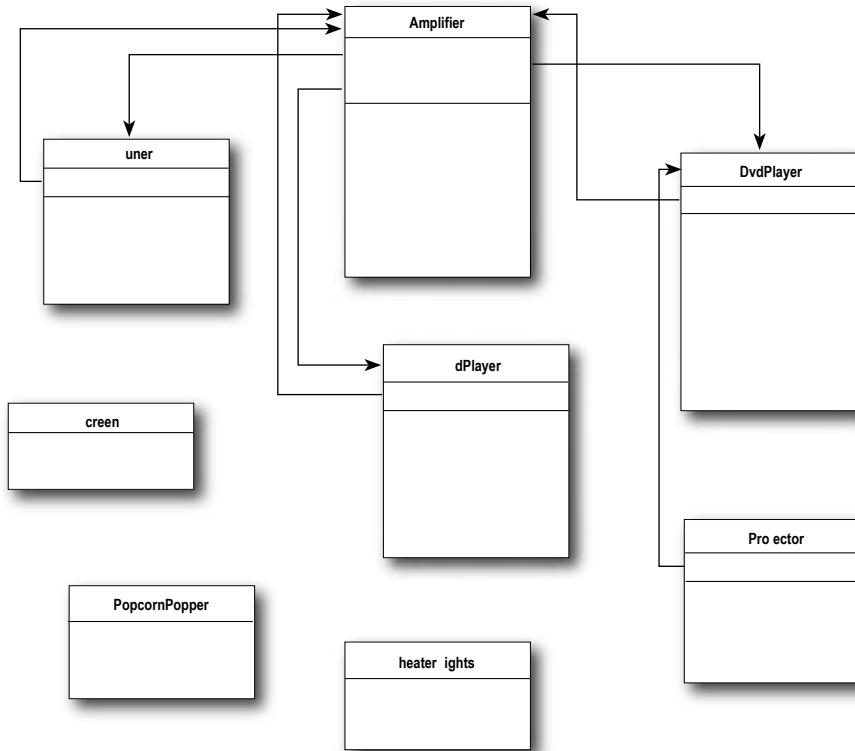
*ha ter*

# Home Sweet Home Theater

Before we dive into the details of the Facade Pattern, let's take a look at a growing national obsession building your own home theater.

You've done your research and you've assembled a killer system complete with a player, a projection video system, an automated screen, surround sound and even a popcorn popper.

Check out all the components you've put together



That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

You've spent weeks running wire, mounting the projector, making all the connections and fine tuning. Now it's time to put it all in motion and enjoy a movie...

*you are here ▶*

## Watching a movie the hard way

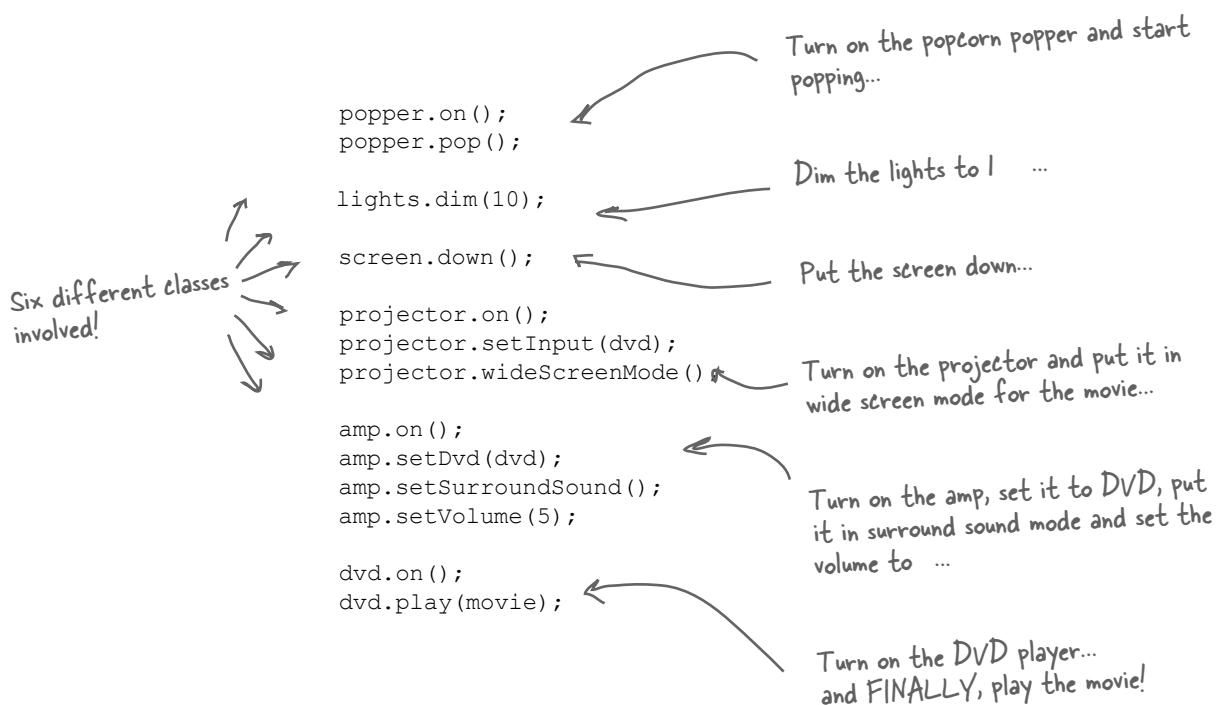
Pic out a D D, rela , and get ready for ovie agic h,  
there's ust one thing to watch the ovie, you need to  
perfor a few tas s

- ➊ Turn on the popcorn popper
- ➋ Start the popper popping
- ➌ Dim the lights
- ➍ Put the screen down
- ➎ Turn the projector on
- ➏ Set the projector input to DVD
- ➐ Put the projector on wide-screen mode
- ➑ Turn the sound ampli er on
- ➒ Set the ampli er to DVD input
- ➓ Set the ampli er to surround sound
- ➔ Set the ampli er volume to medium
- ➕ Turn the DVD Player on
- ➖ Start the DVD Player playing

I'm already exhausted  
and all I've done is turn  
everythin on!



Let's check out those same tasks of the classes and the method calls needed to perform the



But there's more

- When the movie is over, how do you turn everything off?  
Wouldn't you have to do all of this over again, in reverse?
- Wouldn't it be as complex to listen to a CD or the radio?
- If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure.

So what to do? The complexity of using your home theater is becoming apparent

Let's see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

*you are here ▶*

## ights, Camera, Facade

Facade is just what you need – with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.

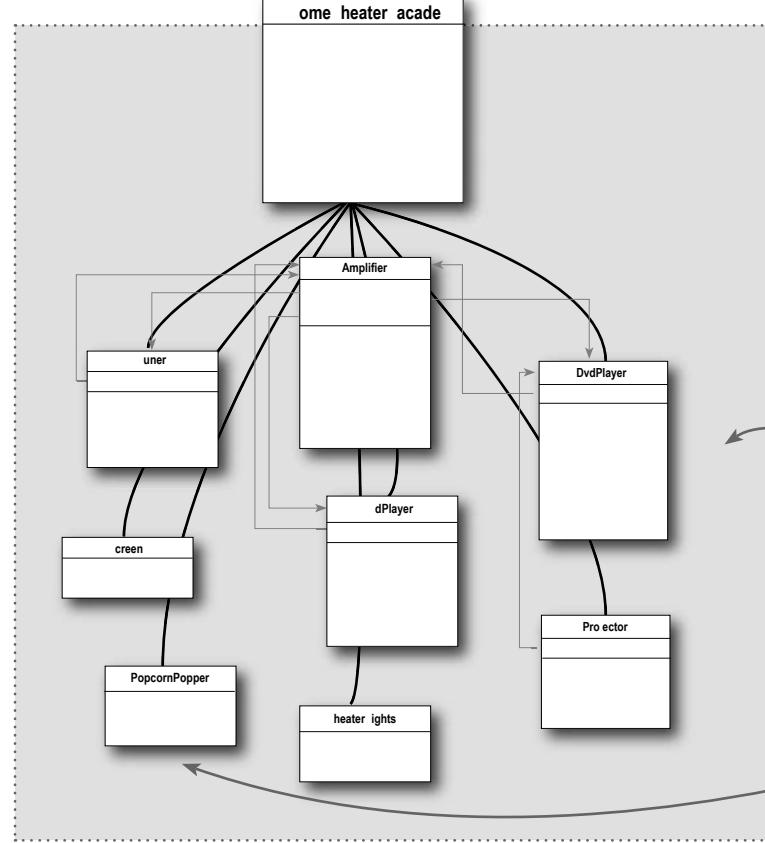
Let's take a look at how the Facade operates

- 1 It's time to create a facade for the home theater's system. To do this we create a new class Home theater facade, which exposes a few simple methods such as watchMovie .

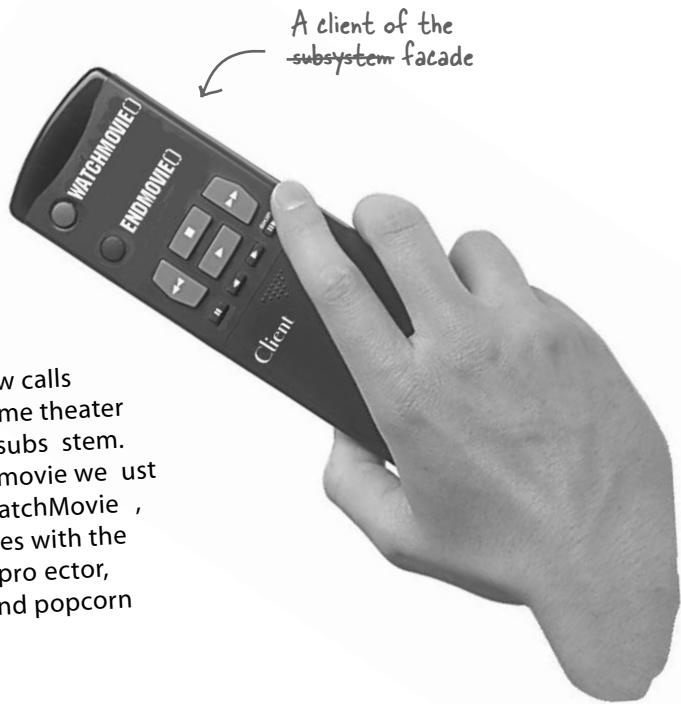
The Facade

- 2 The facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie method.

The subsystem the Facade is simplifying.



- ➊ Your client code now calls methods on the home theater facade, not on the subsystem. So now to watch a movie we just call one method, `watchMovie()`, and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.



- ➋ The facade still leaves the subsystem accessible to be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

there are no  
**Dumb Questions**

**Q:** If the facade encapsulates the subsystem classes how does a client that needs lower level functionality gain access to them?

**A:**

**Q:** What is the benefit of the facade other than the fact that I now have a simpler interface?

**A:**

**A facade not only simplifies an interface, it decouples a client from a subsystem of components.**

**Q:** Does the facade add any functionality or does it just pass through each request to the subsystem?

**A:**

**Q:** One way to tell the difference between the Adapter Pattern and the facade Pattern is that the adapter wraps one class and the facade may represent many classes?

**A:**

**Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.**

**Q:** Does each subsystem have only one facade?

**A:**

intent  
alter

simplified

# Constructing your home theater facade

Let's step through the construction of the HomeTheaterFacade. The first step is to use composition so that the facade has access to all the components of the subsystem.

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
        Tuner tuner,
        DvdPlayer dvd,
        CdPlayer cd,
        Projector projector,
        Screen screen,
        TheaterLights lights,
        PopcornPopper popper) {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // other methods here
}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

## Implementing the simplified interface

Now it's time to bring the components of the subsystem together into a unified interface. Let's implement the watchMovie() and endMovie() methods

```
public void watchMovie(String movie) {  
    System.out.println("Get ready to watch a movie...");  
    popper.on();  
    popper.pop();  
    lights.dim(10);  
    screen.down();  
    projector.on();  
    projector.wideScreenMode();  
    amp.on();  
    amp.setDvd(dvd);  
    amp.setSurroundSound();  
    amp.setVolume(5);  
    dvd.on();  
    dvd.play(movie);  
}  
  
public void endMovie() {  
    System.out.println("Shutting movie theater down...");  
    popper.off();  
    lights.on();  
    screen.up();  
    projector.off();  
    amp.off();  
    dvd.stop();  
    dvd.eject();  
    dvd.off();  
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.



T in a o tt e acade yo e enco ntered in t e Ja a API.  
W ere wo ld yo li e to a e a ew new one ?

# Time to watch a movie the easy way

t's H W M

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here
        HomeTheaterFacade homeTheater =
            new HomeTheaterFacade(amp, tuner, dvd, cd,
                                  projector, screen, lights, popper);
        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```



Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Here's the output.

Calling the Facade's `watchMovie()` does all this work for us...

```
File Edit Window Help Sna e W ydlHa eToBeSna e ?
%java HomeTheaterTestDrive
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

...and here, we're done watching the movie, so calling `endMovie()` turns everything off.

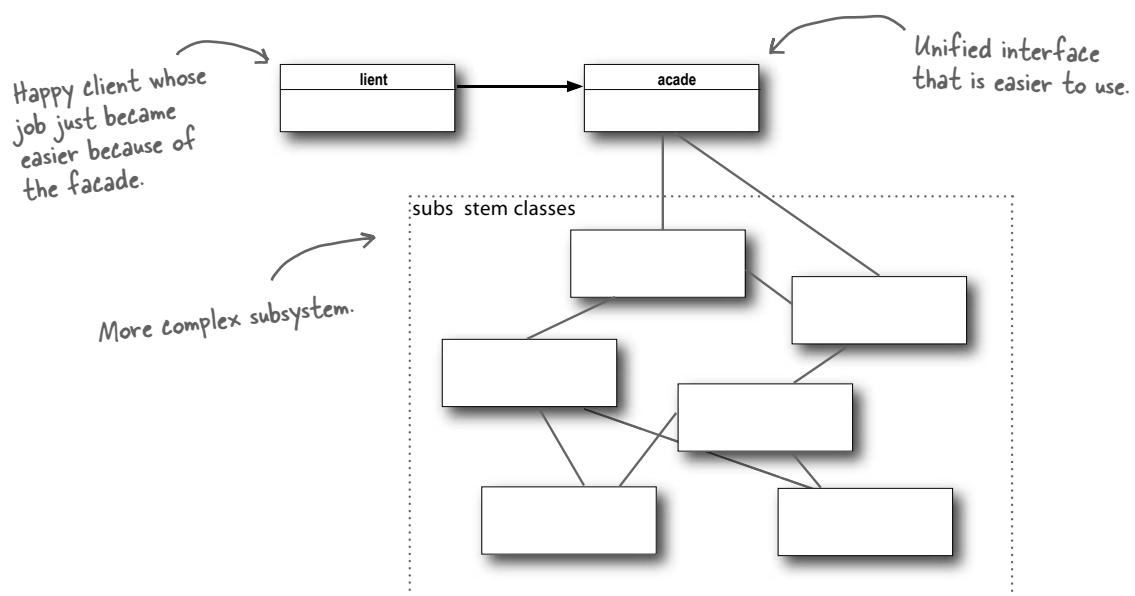
## Facade Pattern defined

To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind bending abstractions to get your head around. But that doesn't make it any less powerful—the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object oriented principle.

Before we introduce that new principle, let's take a look at the official definition of the pattern

**The facade pattern** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.

here isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade is to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram



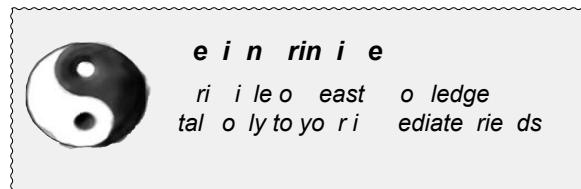
That's it; you've got another pattern under your belt. Watch out, this one can challenge some assumptions

Now, it's time for that new principle.

## The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close friends.

The principle is usually stated as



But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.


How many classes interact with code coupled to?

```
public float getTemp() {
    return station.getThermometer().getTemperature();
}
```

## How NOT to Win Friends and Influence Objects

key, but how do you keep from doing this? The principle provides some guidelines: take any object; now from any method in that object, the principle tells us that we should only invoke methods that belong to

- the object itself
- objects passed in as a parameter to the method
- any object the method creates or instantiates
- any components of the object

Notice that these guidelines tell us not to call methods on objects that were returned from calling other methods!!

Think of a "component" as any object that is referenced by an instance variable. In other words think of this as a HAS-A relationship.

This sounds kind of stringent doesn't it? What's the harm in calling the method of an object we get back from another call? Well, if we were to do that, then we'd be making a request of another object's subpart (and increasing the number of objects we directly know). In such cases, the principle forces us to ask the object to make the request for us; that way we don't have to know about its component objects (and we keep our circle of friends small). For example

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

```
public float getTemp() {
    return station.getTemperature();
}
```

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

## Keeping your method calls in bounds...

Here's a Car class that demonstrates all the ways you can call methods and still adhere to the Principle of Least Knowledge

```

public class Car {
    Engine engine; ← Here's a component of
    // other instance variables this class. We can call
                                its methods.

    public Car() { ← Here we're creating a new
        // initialize engine, etc. object, its methods are legal.
    }

    public void start(Key key) { ← You can call a method
        Doors doors = new Doors(); on an object passed as
                                a parameter.

        boolean authorized = key.turns(); ← You can call a method on a
                                            component of the object.

        if (authorized) { ← You can call a local method
            engine.start(); ← within the object.
            updateDashboardDisplay(); ←
            doors.lock(); ← You can call a method on an
                            object you create or instantiate.

        }
    }

    public void updateDashboardDisplay() { ←
        // update display
    }
}

```

*there are no  
Dumb Questions*

**Q:** here is another principle called the Law of Demeter; how are they related?

**A:**

**Q:** Are there any disadvantages to applying the Principle of Least Knowledge?

io atin the rin i eo ea t no ed e



Do either or the code violate the Principle of Least Knowledge?  
Why or why not?

```
public House {  
    WeatherStation station;  
  
    // other methods and constructor  
  
    public float getTemp() {  
        return station.getThermometer().getTemperature();  
    }  
}  
  
  
public House {  
    WeatherStation station;  
  
    // other methods and constructor  
  
    public float getTemp() {  
        Thermometer thermometer = station.getThermometer();  
        return getTempHelper(thermometer);  
    }  
  
    public float getTempHelper(Thermometer thermometer) {  
        return thermometer.getTemperature();  
    }  
}
```



**HARD HAT AREA. WATCH OUT  
FOR FALLING ASSUMPTIONS**

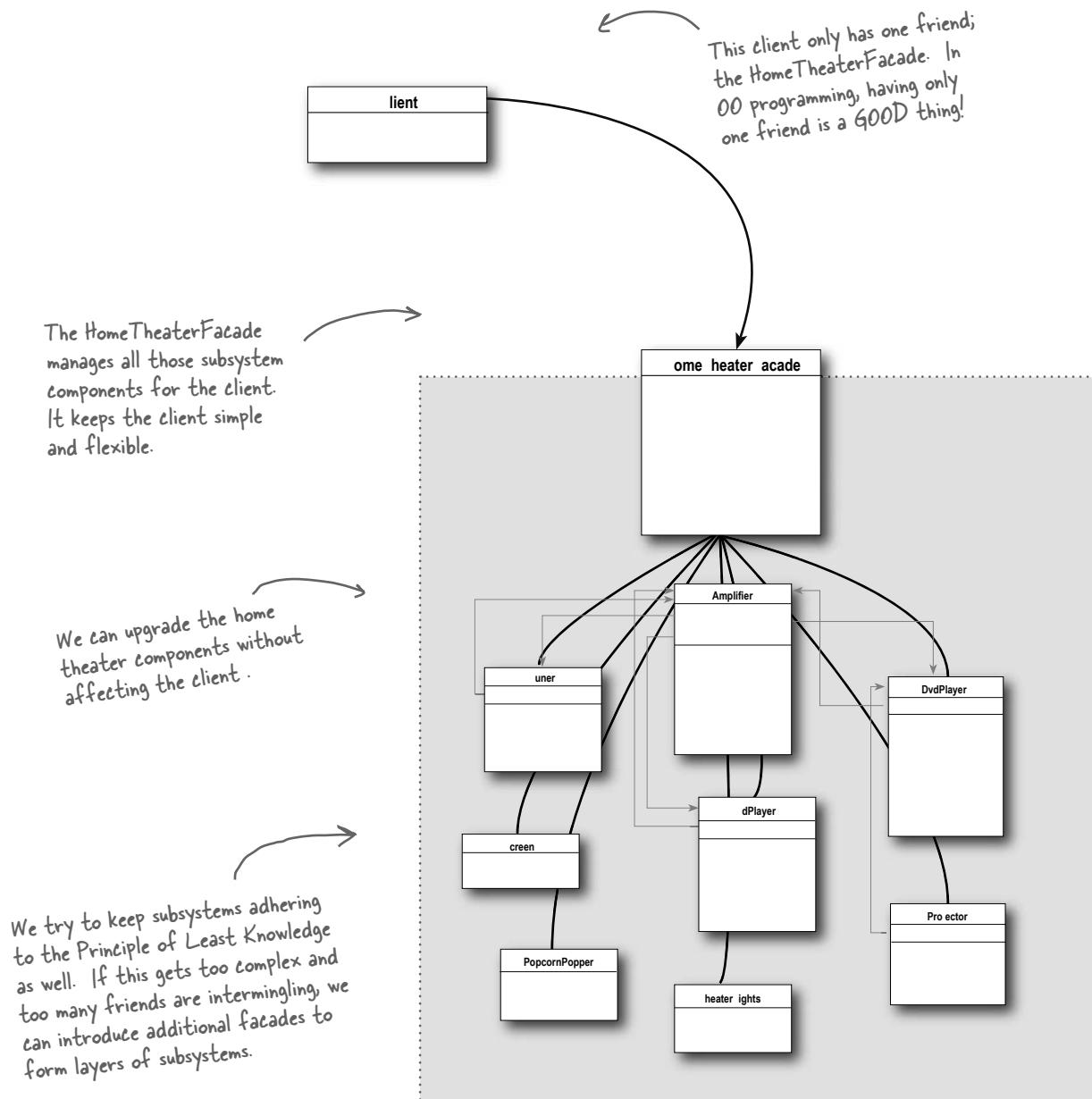


Can you think of a common Java situation that violates the Principle of Least Knowledge?

Should you care?

Answer: How about System.out.println()?

# The Facade and the Principle of Least Knowledge



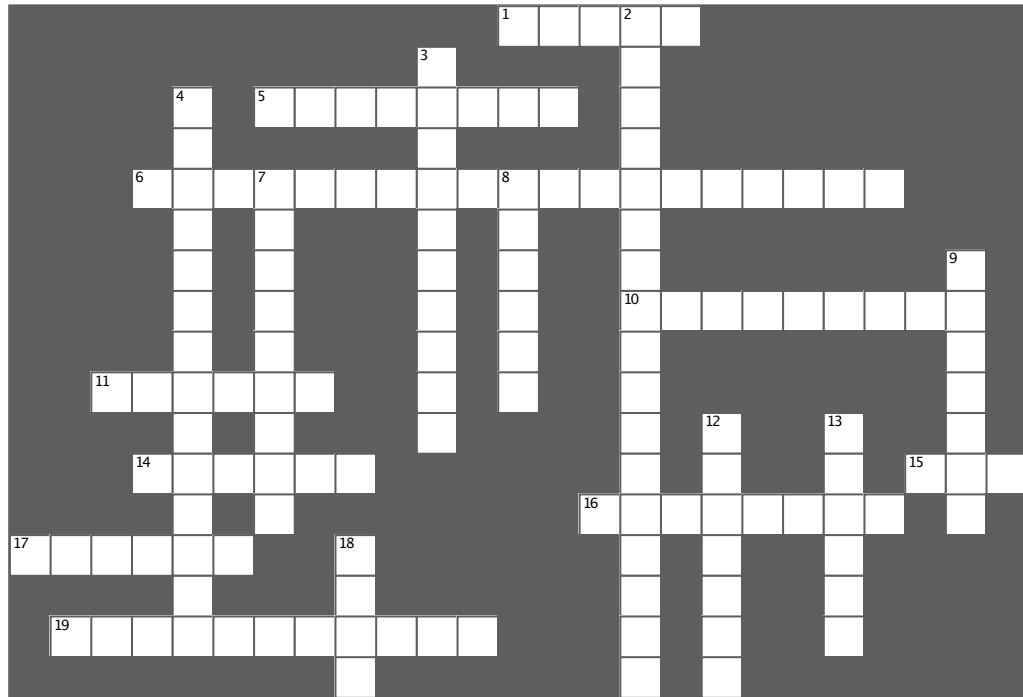


# Tools for your Design Toolbox

**our tool is starting to get heavy in this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use**



The diagram illustrates the relationship between Object-Oriented (OO) concepts. At the top left is a stack of cards labeled "OO Principles". The top card lists principles such as Encapsulate what varies, Favor composition over inheritance, Program to interfaces, not implementations, Strive for loosely coupled designs between objects that interact, Classes should be open for extension but closed for modification, Depend on abstractions, Do not depend on concretions, and Only talk to your friends. To the right of this stack is a single card labeled "OO Basics" which lists Abstraction, Encapsulation, Polymorphism, and Inheritance. A curved arrow points from the "Only talk to your friends" principle towards the "OO Basics" card. Below these is another stack of cards labeled "OO Patterns". The top card in this stack is partially visible, showing the first few letters of several patterns: Strategy, Decorator, Adapter, Facade, and Model View Controller. To the right of this stack is a card describing the Adapter pattern: "Adapter - Converts the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces." Below this is a card describing the Facade pattern: "Facade - Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." A curved arrow points from the "Adapter" card towards the "OO Basics" card.



#'\$\$

```
%#; NQAKN@PKN?=@LFA@N?=J KJ@S@NL KJA
K>FA?P
)#/ J / @LFAN<<<< =J @FANB?A
* #6 KRA SA S=PDA@) SKN@)
%$#3@ 1 QNKLA UKQI EDPJAA@KJA KBPDAOA
RSK SKN@)
%#/ @LFANS E@D RSK NKIAO RSK SKN@)
%# 2=?=@A OP@H<<<< HKS HARAH=??ACO
%# 0 Q?GO@X IP>APPANPD=J ; QNGAUO
%# 0 E@P=JP=CA KBPDA 9 NB?E@A KB5A=OP
4JKSH@A : UPAI #K@P<<<<
%#< <<<<<<< O@ L@HAO=J @FANB?A
%#< <<<<<<< O@ L@HAO=J @FANB?A
%# 7 AS / I ANP=J @A=I RSK SKN@)
```

" %!

```
&#0 A?KN@PKN?=H@ @LFANPD@ ' SKN@)
' #8 JA=@P=JP=CA KB2=?=@
( #9 NB?E@A PD=PS=QJ P=O@=OJ=O@P@QJ @@@
RSK SKN@)
+#/ <<<<< =@O@JAS >AD=RKN
, #6 =OMQAN@C =O= 0 Q?G
- #1 T=I L@A PD=PRKH@AO@DA 9 NB?E@A KB5A=OP
4JKSH@A : UPAI #K@P<<<<
%#7 K I KRA E@KI L@HAO S@DKQP@D@)
%#/ @LFAN?@U PQAO@DA <<<<<<< @FANB?A
%#/ J / @LFAN=J @= 0 A?KN@PKN?=J >A O@P@K
<<<< =J K>FA?P
```

you are here ▶

## exercise solution



# Exercise solutions

### Sharpen your pencil

Let's say we also need an adapter to convert a Duck to a Turkey.  
Let's call it Duck Adapter. Write that class:

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

    public void gobble() {
        duck.quack();
    }

    public void fly() {
        if (rand.nextInt(5) == 0) {
            duck.fly();
        }
    }
}
```

Now we are adapting Turkeys to Ducks, so we implement the Turkey interface.

We stash a reference to the Duck we are adapting.

We also recreate a random object; take a look at the fly() method to see how it is used.

A gobble just becomes a quack.

Since ducks fly a lot longer than turkeys, we decided to only fly the duck on average one of five times.

### Sharpen your pencil

Do either or both classes violate the Principle of Least Knowledge?  
For each, why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}

public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Violates the Principle of Least Knowledge!  
You are calling the method of an object returned from another call.

Doesn't violate Principle of Least Knowledge!  
This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

## Answers



# er ise solutions

You've seen how to implement an adapter that adapts an enumeration to an iterator; now write an adapter that adapts an iterator to an enumeration.

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

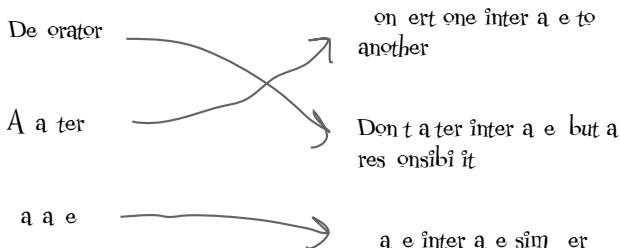
    public Object nextElement() {
        return iterator.next();
    }
}
```

## \* WHO DOES WHAT? \*

Match each pattern with its intent

### **Pattern**

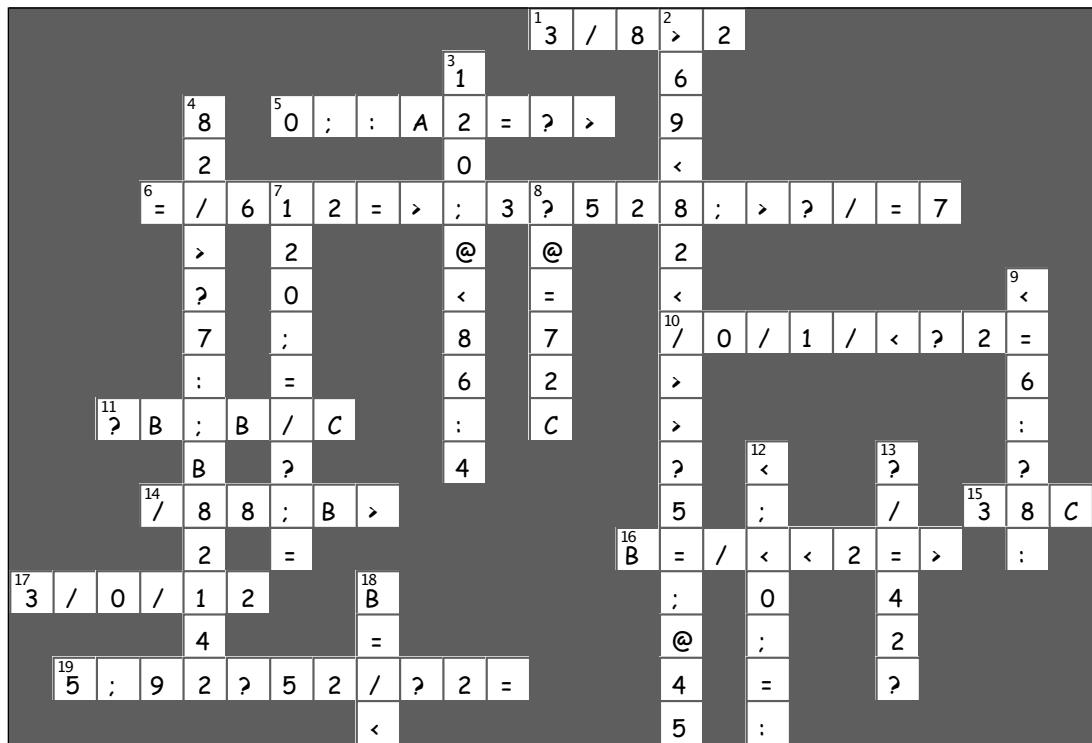
### **Intent**



ro ord u e o ution



# er ise solutions



ha ter

the em ate etho attern

# ncapsulating Igorith s



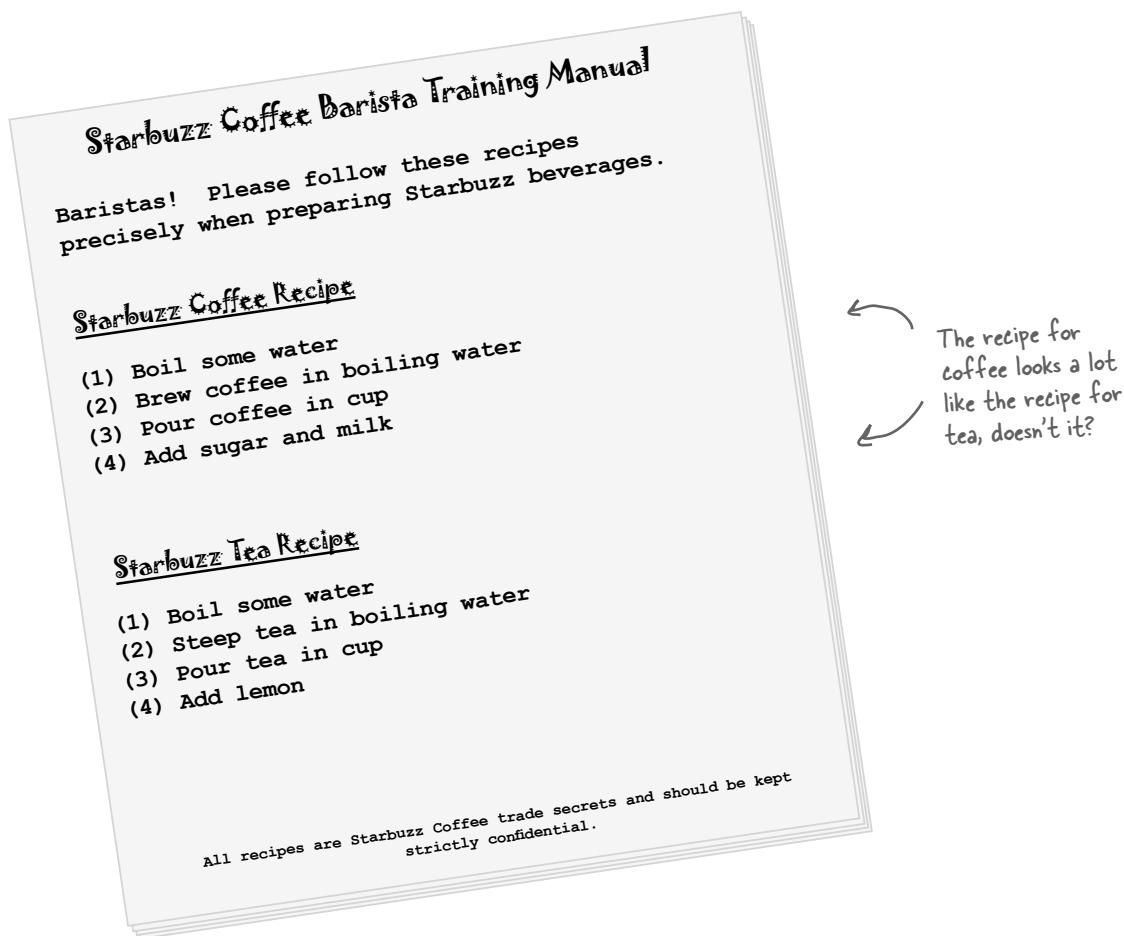
We're oin to et down to encap latin  
piece o al orit m o t at cla e can oo t em el e ri t into a comp tation  
anytime t ey want. We're en oin to learn a o t a de i n principle in pired y  
Hollywood.

*o ee and tea re i e are i i ar*

## It's time for some more caffeine

ome people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine of course

But there's more; tea and coffee are made in very similar ways. Let's check it out



# Whipping up some coffee and tea classes in ava

Let's play coding arista and write  
so e code for creating coffee and tea



## Here's the coffee

```
Here's our Coffee class for making coffee.
public class Coffee {
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup and add sugar and milk.

*you are here ▶*

## and now the ea

```
public class Tea {  
  
    void prepareRecipe() {  
        boilWater();  
        steepTeaBag(); ←  
        pourInCup();  
        addLemon();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water"); ←  
    }  
  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea"); ←  
    }  
  
    public void addLemon() {  
        System.out.println("Adding Lemon"); ←  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup"); ←  
    }  
}
```

This looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.



When I've got code duplication, that's a good sign I need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?



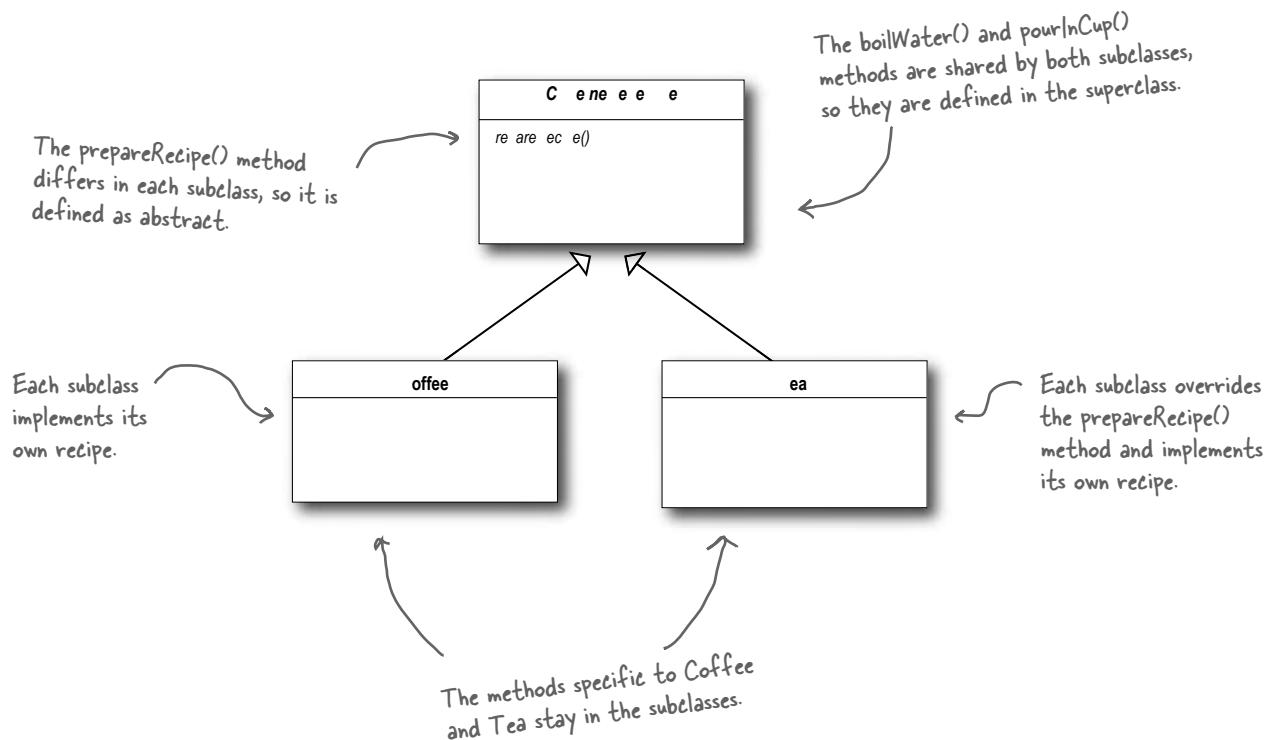


## esign Pu le

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy.

## Sir, may I abstract your Coffee, Tea?

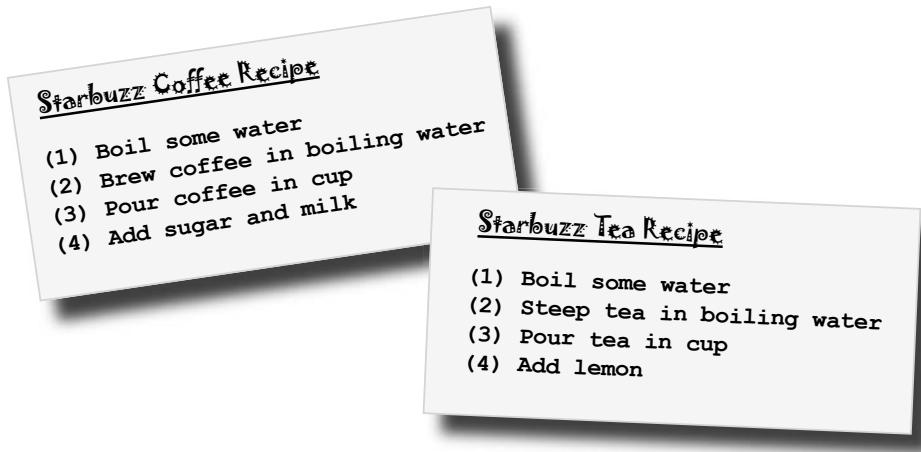
t looks like we've got a pretty straightforward design  
e ercise on our hands with the Coffee and Tea classes.  
our first cut might have looked something like this



Did we do a good job on the redesign? Hmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

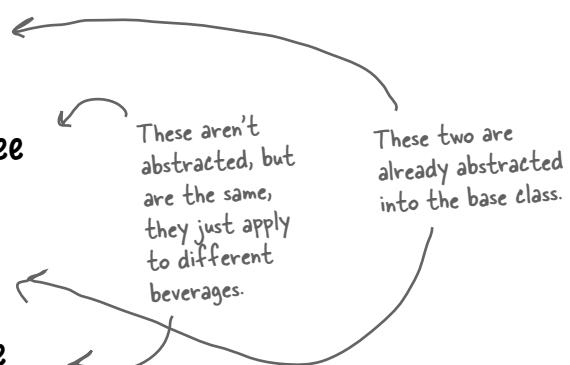
## Taking the design further...

o what else do coffee and tea have in common? Let's start with the recipes



Notice that both recipes follow the same algorithm

- ① Boil some water.
- ② Use the hot water to extract the coffee or tea.
- ③ Pour the resulting beverage into a cup.
- ④ Add the appropriate condiments to the beverage.



o, can we find a way to abstract prepare recipe too? Yes, let's find out

you are here ▶

## Abstracting prepareRecipe

Let's step through abstracting prepare recipe from each subclass that is, the coffee and tea classes

- 1 The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.

```
coffee                                tea
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}

void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

```
graph LR; A[coffee] --> B[steepTeaBag()]; B --> C[addLemon()]; C --> D[addSugarAndMilk()]; D --> E[tea]
```

Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepare\_recipe() method will look like this

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

- 2 Now we have a new prepare\_recipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass

```

public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
}

```

CaffeineBeverage is abstract, just like in the class design.

```

abstract void brew();
abstract void addCondiments();

void boilWater() {
    System.out.println("Boiling water");
}

void pourInCup() {
    System.out.println("Pouring into cup");
}
}

```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps and to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

- ③ Finally we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments

```

public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

```

As in our design, Tea and Coffee now extend CaffeineBeverage.

Tea needs to define brew() and addCondiments() the two abstract methods from Beverage.

```

public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}

```

Same for Coffee; except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

a dia ra or a eine e era e



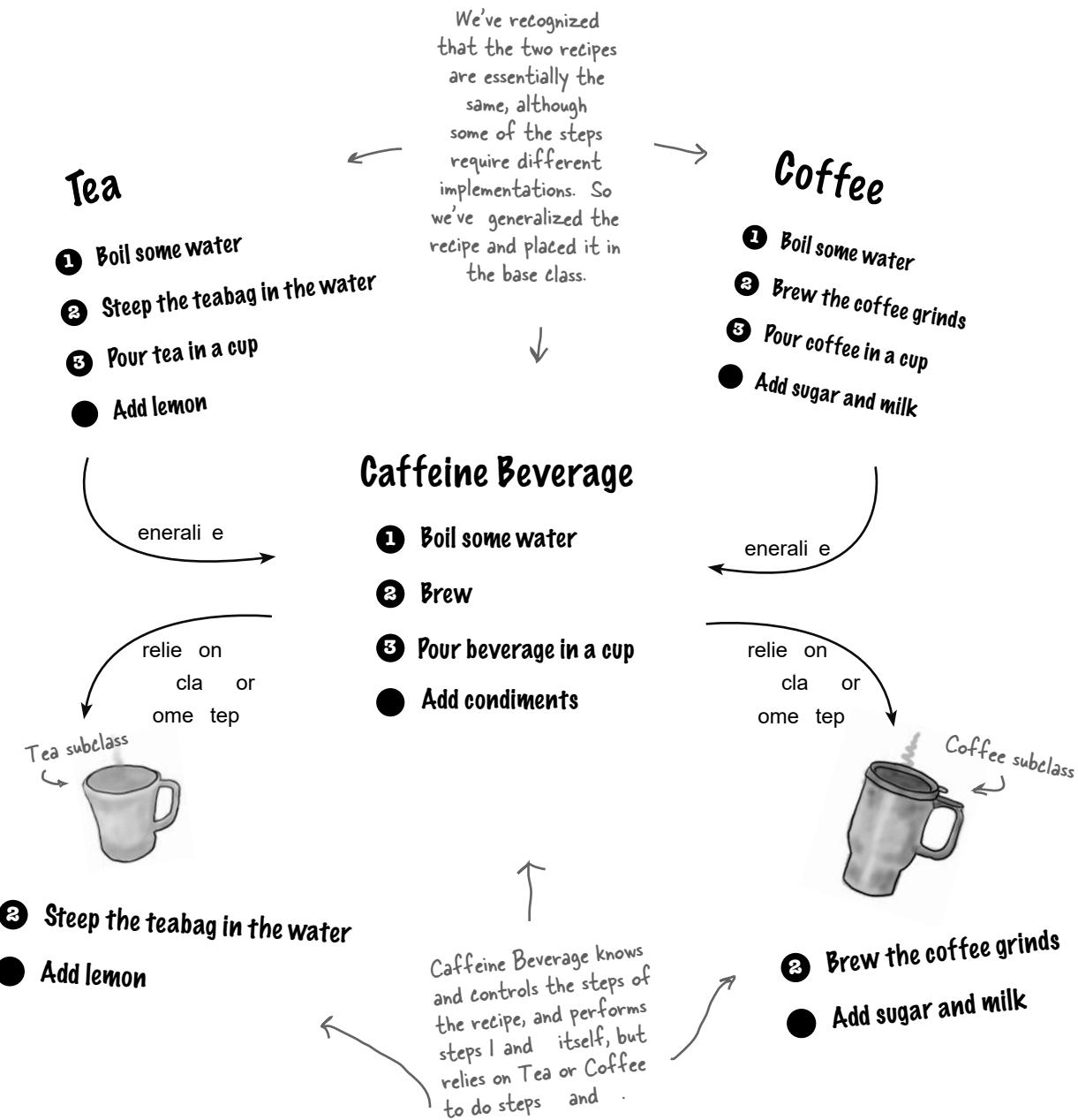
## Sharpen your pencil

---

Draw the new class diagram now that we've modified the implementation of `prepareRecipe()` into the `CafeineBeverage` class:

ha ter

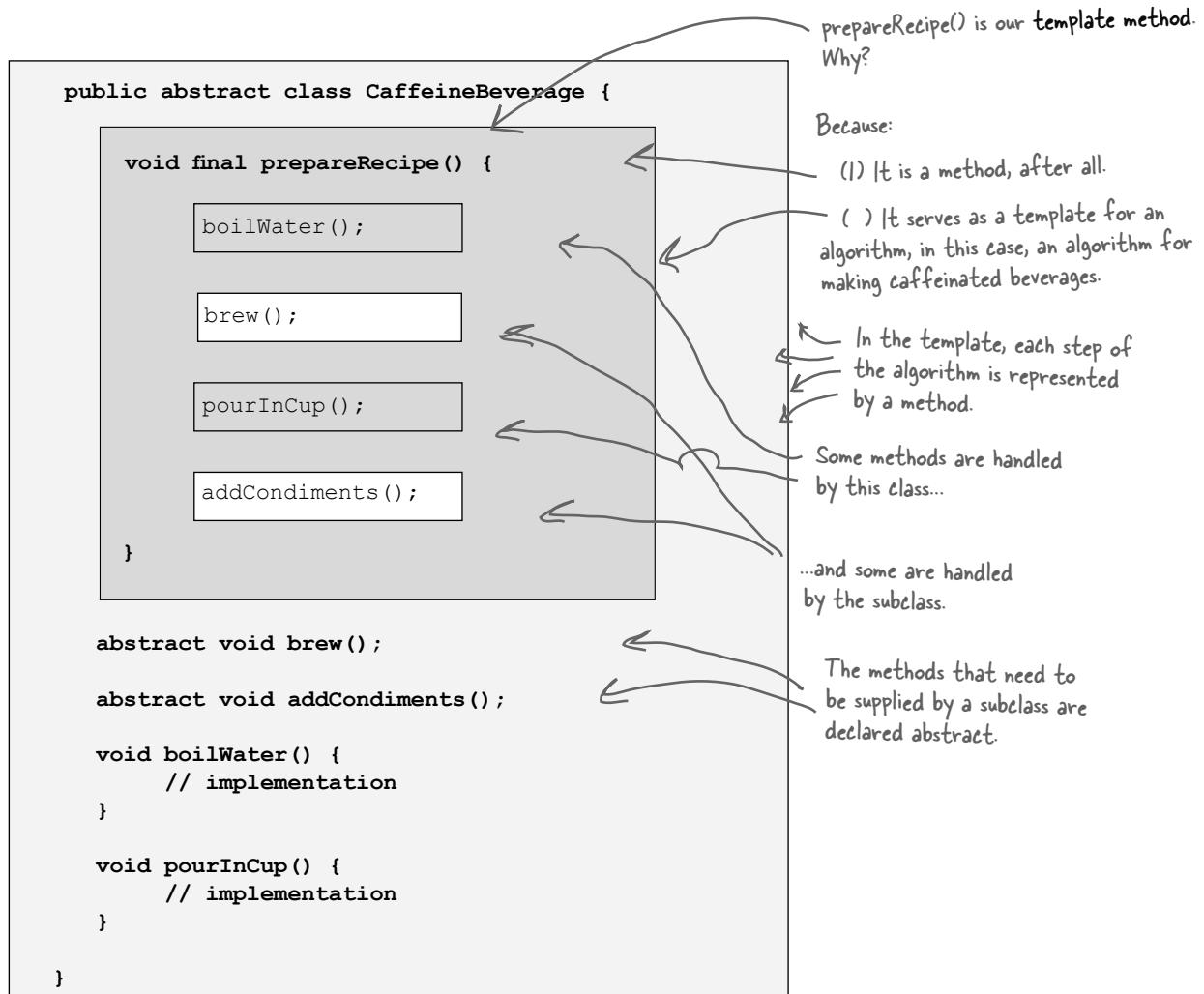
# What have we done?



you are here ▶

## Meet the Template Method

We've basically just implemented the template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual template method



**The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.**

# et's make some tea...

**et's step through a ing a tea and trace through how the te plate ethod wor s ou'll see that the te plate ethod controls the algorith at certain points in the algorith , it lets the su class supply the i ple entation of the steps**



- 1      kay, first we need a tea object...

```
Tea myTea = new Tea();
```

```
boilWater();
brew();
pourInCup();
addCondiments();
```

- 2      hen we call the template method

```
myTea.prepareRecipe();
```

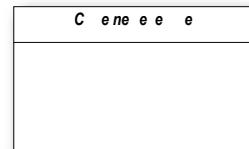
which follows the algorithm for making caffeine beverages...

The prepareRecipe() method controls the algorithm, no one can change this, and it counts on subclasses to provide some or all of the implementation.

- 3      First we boil water

```
boilWater();
```

which happens in CaffeineBeverage.



- e t we need to brew the tea, which only the subclass knows how to do

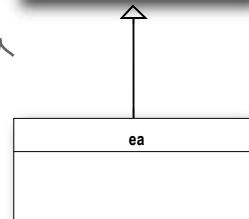
```
brew();
```

●      ow we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage

```
pourInCup();
```

- Finally, we add the condiments, which are specific to each beverage, so the subclass implements this

```
addCondiments();
```



you are here ▶

## What did the Template Method get us?



**Underpowered Tea/Coffee implementation**

Code and Tea are running slow; they control the algorithm.

Code is duplicated across Coffee and Tea.

Code can't be shared because it's open in the class and maintained in multiple classes.

Classes are organized in abstract methods, creating a lot of work to add a new caffeine era.

Knowledge of the algorithm is slow to implement it in different orders many classes.



**New, hip CaffeineBeverage powered by Template Method**

The CaffeineBeverage class runs fast; it's real optimization, and protects it.

The CaffeineBeverage class implements the algorithm, and the class is.

The algorithm is in one place and code can only need to be made there.

The Template Method provides a framework that other caffeine-era classes can be plugged into. New caffeine-era classes only need to implement a complete method.

The CaffeineBeverage class concentrates knowledge about the algorithm and relies on classes to provide complete implementation.

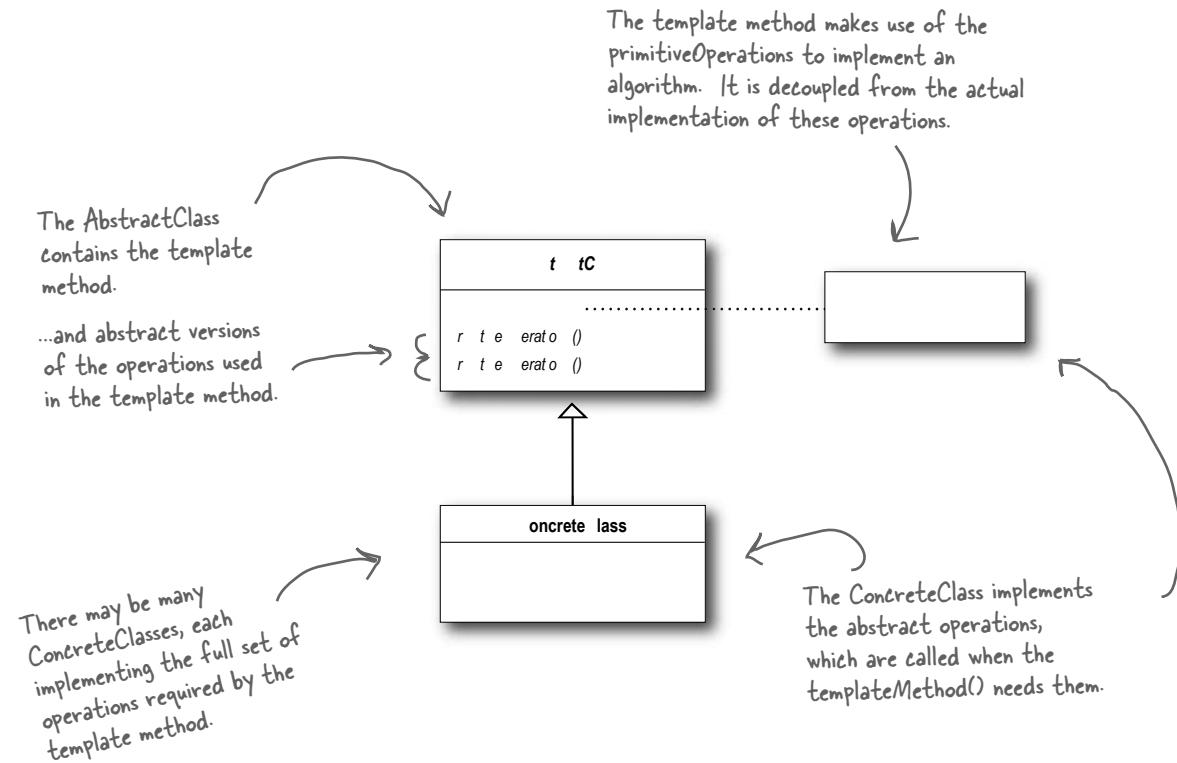
# Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details.

**he template method patter** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

Let's check out the class diagram





## Code Up Close

Let's take a closer look at how the `AbstractClass` is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared `abstract` and meant to be subclassed by classes that provide implementations of the operations.

```
abstract class AbstractClass {
    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();
    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

Here's the template method. It's declared `final` to prevent subclasses from reworking the sequence of steps in the algorithm.

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...



## Code Up Close

Now we're going to look even closer at the types of methods that can go in the abstract class

We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

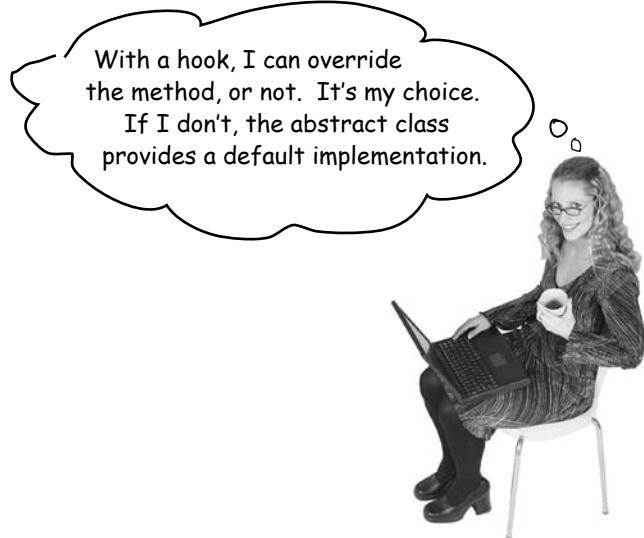
A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

# e late et

hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to hook into the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

here are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later



```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, `customerWantsCondiments()`. If the customer WANTS condiments, only then do we call `addCondiments()`.

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a hook because the subclass can override this method, but doesn't have to.

# Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

**Here's where you override the hook and provide your own functionality.**

**Get the user's input on the condiment decision and return true or false depending on the input.**

**This code asks the user if he'd like milk and sugar and gets his input from the command line.**

## et's run the TestDrive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

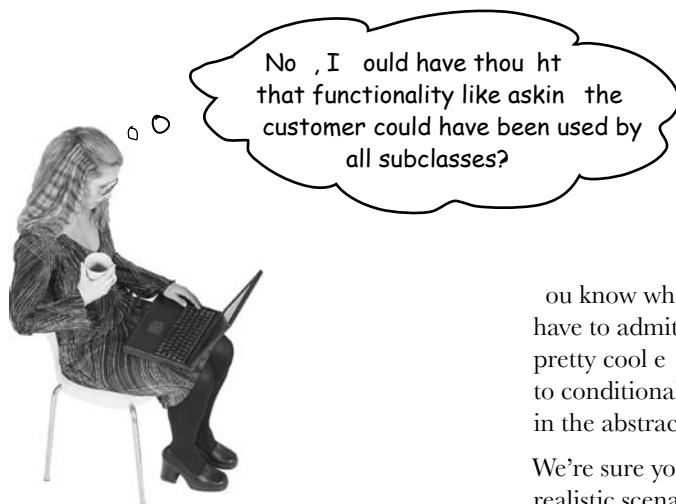
        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

And let's give it a run...

```
File Edit Window Help end-more- one tea
%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y ←
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n ←
%
```



You know what? We agree with you. But you have to admit before you thought of that it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class, right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

---

## *there are no* Dumb Questions

**Q:** When I'm creating a template method how do I know when to use abstract methods and when to use hooks?

**A:**

**Q:** It seems like I should keep my abstract methods small in number otherwise it will be a big job to implement them in the subclass

**A:**

**Q:** What are hooks really supposed to be used for?

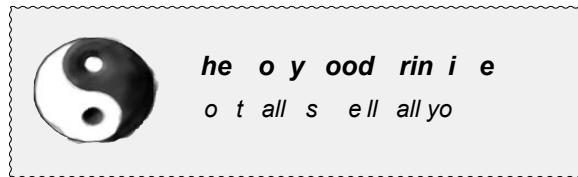
**A:**

**Q:** Does a subclass have to implement all the abstract methods in the Abstract class?

**A:**

## The Hollywood Principle

We've got another design principle for you; it's called the Hollywood Principle



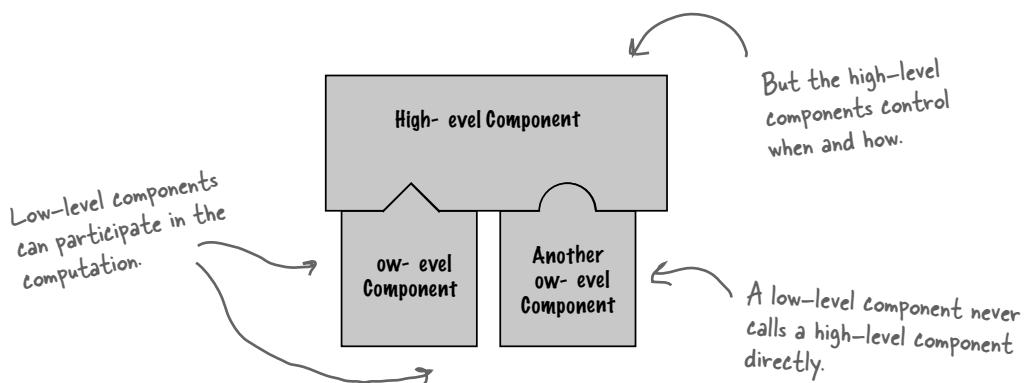
You've heard me say it before, and I'll say it again: don't call me, I'll call you!



easy to remember, right? But what has it got to do with design?

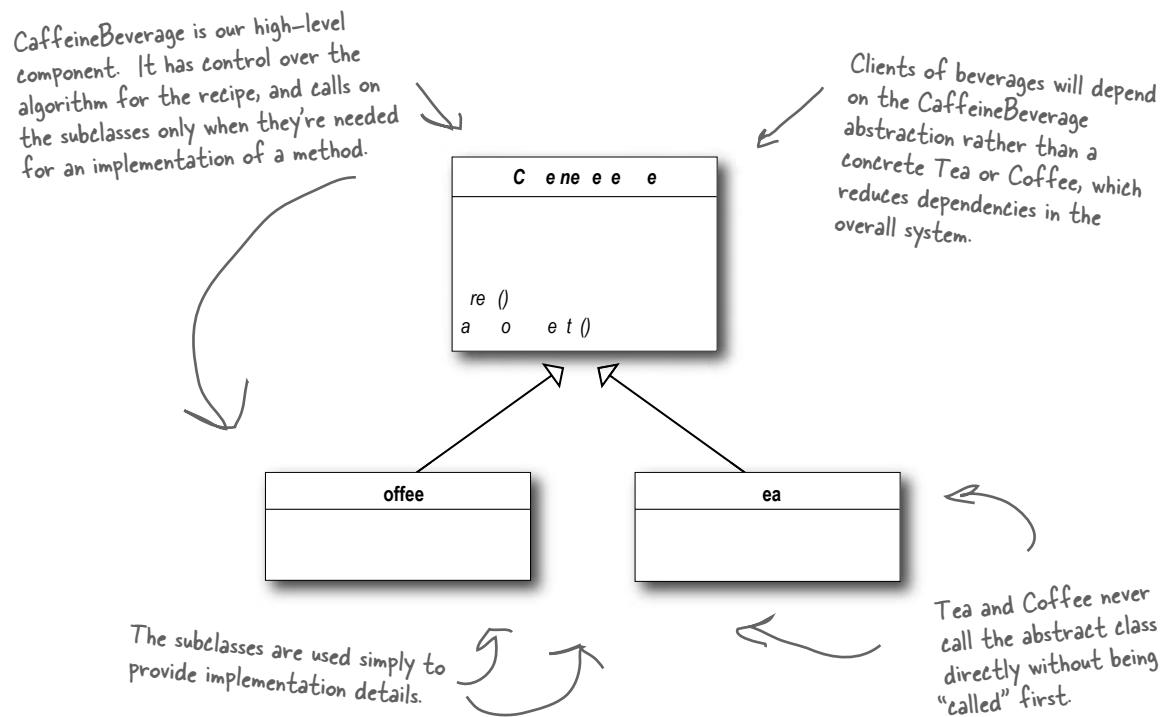
The Hollywood principle gives us a way to prevent dependency rot. Dependency rot happens when you have high level components depending on low level components depending on high level components depending on sideways components depending on low level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

With the Hollywood Principle, we allow low level components to hook themselves into a system, but the high level components determine when they are needed, and how. In other words, the high level components give the low level components a "don't call us, we'll call you" treatment.



# The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How? Let's take another look at our CaffeineBeverage design.



What other pattern made the Hollywood Principle?

The Factory Method, Orderer; any other?

*ho doe hat*

## Dumb Questions

**Q:** How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

**A:**

**Q:** Is a low level component disallowed from calling a method in a higher level component?

**A:**



Match each pattern with its description

### **Pattern**

### **Description**

em ate etho

handles interface behaviors and uses delegation to implement behavior

trate

uses inheritance to implement states in an algorithm

a tor etho

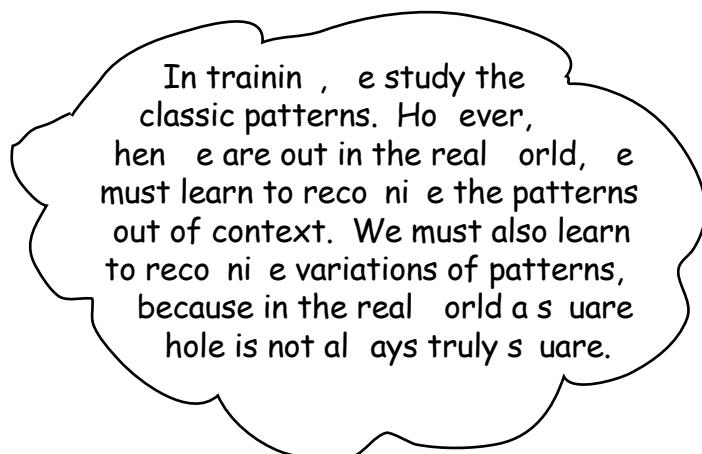
uses inheritance to reuse code

## Template Methods in the Wild

The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild. You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.

This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...

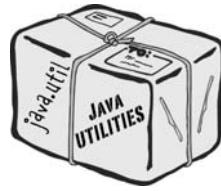


## Sorting with Template Method

What's something we often need to do with arrays?  
Sort them

Recognizing that, the designers of the `java.util.Arrays` class have provided us with a handy template method for sorting. Let's take a look at how this method operates

We actually have two methods here and they act together to provide the sort functionality.



We've pared down this code a little to make it easier to explain. If you'd like to see it all, grab the source from Sun and check it out...

The first method, `sort()`, is just a helper method that creates a copy of the array and passes it along as the destination array to the `mergeSort()` method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The `mergeSort()` method contains the sort algorithm, and relies on an implementation of the `compareTo()` method to complete the algorithm. If you're interested in the nitty gritty of how the sorting happens, you'll want to check out the Sun source code.

```
private static void mergeSort(Object src[], Object dest[],
                             int low, int high, int off)
{
    for (int i=low; i<high; i++) {
        for (int j=i; j>low &&
             ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    return;
}
```

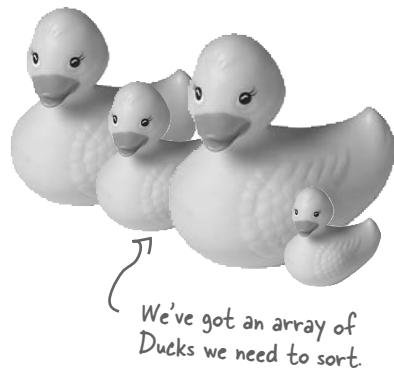
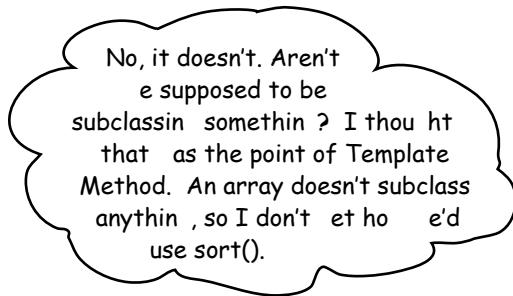
Think of this as the template method.

This is a concrete method, already defined in the `Arrays` class.

`compareTo()` is the method we need to implement to "fill out" the template method.

## We've got some ducks to sort...

Let's say you have an array of ducks that you'd like to sort. How do you do it? Well, the sort template method in `arrays` gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the `compareTo()` method... Make sense?



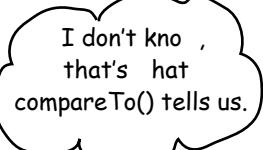
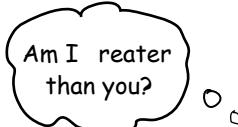
We've got an array of Ducks we need to sort.

Good point. Here's the deal: the designers of `sort()` wanted it to be useful across all arrays, so they had to make `sort()` a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because `sort()` really isn't defined in our superclass, the `sort()` method needs to know that you've implemented the `compareTo()` method, or else you don't have the piece needed to complete the sort algorithm.

To handle this, the designers made use of the `Comparable` interface. All you have to do is implement this interface, which has one method (surprise) `compareTo()`.

## What is `compareTo` ?

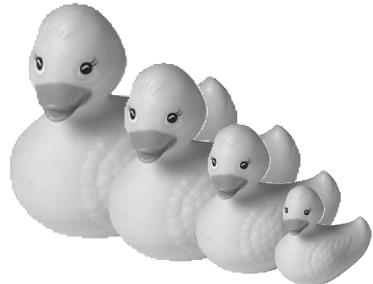
The `compareTo()` method compares two objects and returns whether one is less than, greater than, or equal to the other. `sort()` uses this as the basis of its comparison of objects in the array.



# Comparing Ducks and Ducks

Okay, so you know that if you want to sort ducks, you're going to have to implement this compareTo() method; by doing that you'll give the arrays class what it needs to complete the algorithm and sort your ducks.

Here's the duck implementation



```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Object object) {
        Duck otherDuck = (Duck) object;
        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Remember, we need to implement the Comparable interface since we aren't really subclassing.

Our Ducks have a name and a weight

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

# et's sort some Ducks

**Here's the test drive for sorting Duck s**

```

public class DuckSortTestDrive {
    public static void main(String[] args) {
        Duck[] ducks = {
            new Duck("Daffy", 8),
            new Duck("Dewey", 2),
            new Duck("Howard", 7),
            new Duck("Louie", 2),
            new Duck("Donald", 10),
            new Duck("Huey", 2)
        };
        System.out.println("Before sorting:");
        display(ducks);
        Arrays.sort(ducks);
        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (int i = 0; i < ducks.length; i++) {
            System.out.println(ducks[i]);
        }
    }
}

```

Notice that we call `Arrays' static method sort`, and pass it our Ducks.

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

It's sort time!

Let's print them (again) to see their names and weights.

**et the sorting co ence**

```

File Edit Window Help DonaldNeed ToGoOnADiet
%java DuckSortTestDrive
Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%

```

The unsorted Ducks

The sorted Ducks

# The making of the sorting duck machine

**Let's trace through how the arrays sort template method works. We'll check out how the template method controls the algorithm, and at certain points in the algorithm, how it uses our Duck class to supply the implementation of a step.**

- 1 First, we need an array of ducks

```
Duck[] ducks = {new Duck("Daffy", 8), ...};
```

- 2 When we call the sort() template method in the Array class and pass it our ducks

```
Arrays.sort(ducks);
```

The sort() method (and its helper mergeSort()) controls the sort procedure.

- 3 To sort an array, you need to compare two items one by one until the entire list is in sorted order.

When it comes to comparing two ducks, the sort method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to

```
ducks[0].compareTo(ducks[1]);
```

First Duck    Duck to compare it to

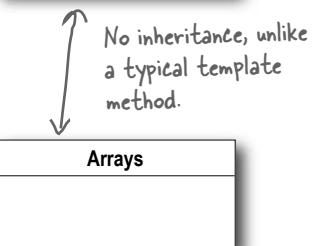
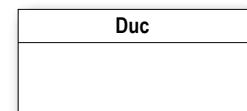
- If the ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays

```
swap();
```

- The sort method continues comparing and swapping ducks until the array is in the correct order

```
for (int i=low; i<high; i++) {  
    ... compareTo() ...  
    ... swap() ...  
}
```

The sort() method controls the algorithm, no class can change this. sort() counts on a Comparable class to provide the implementation of compareTo()



## there are no Dumb Questions

**Q:** Is this really the template method Pattern or are you trying too hard?

e tre

**A:**

**Q:** His implementation of sorting actually seems more like the Strategy Pattern than the template method Pattern. Why do we consider it template method?

**A:**

**Q:** Are there other examples of template methods in the Java API?

**A:**



We now that we could do composition over inheritance, right? Well, the implementer of the sort() template method decided not to do inheritance and instead to implement sort() as a static method that composed with a Comparable interface. How is this better? How is it worse? How would you approach this problem? Do Java arrays make particularly tricy?



This is another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

## Swingin' with Frames

Next on our template Method safari... keep your eye out for swinging Frames! If you haven't encountered Frame, it's the most basic swing container and inherits a paint() method. By default, paint() does nothing because it's a hook. By overriding paint(), you can insert yourself into Frame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the Frame. Here's an embarrassingly simple example of using a Frame to override the paint() hook method

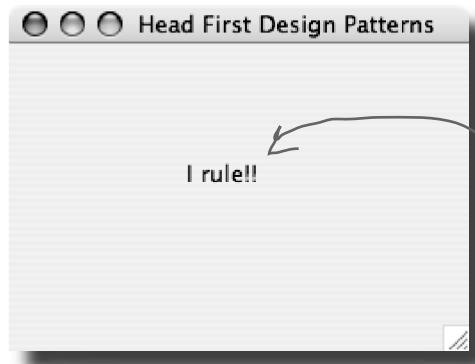


```
public class MyFrame extends JFrame {  
  
    public MyFrame(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        this.setSize(300, 300);  
        this.setVisible(true);  
    }  
  
    public void paint(Graphics graphics) {  
        super.paint(graphics);  
        String msg = "I rule!!";  
        graphics.drawString(msg, 100, 100);  
    }  
  
    public static void main(String[] args) {  
        MyFrame myFrame = new MyFrame("Head First Design Patterns");  
    }  
}
```

We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

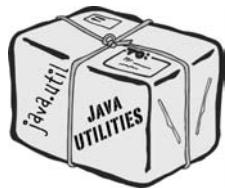
Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls paint(). By default, paint() does nothing... it's a hook. We're overriding paint(), and telling the JFrame to draw a message in the window.



Here's the message that gets painted in the frame because we've hooked into the paint() method.

# Applets



ur final stop on the safari the applet.

ou probably know an applet is a small program that runs in a web page. ny applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet is going away...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint() is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

If the user goes to another page, the stop hook is used, and the applet can do whatever it needs to do to stop its actions.

And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

Well looky here! Our old friend the paint() method! Applet also makes use of this method as a hook.

**Concrete applets make extensive use of hooks to supply their own behaviors. Because these methods are implemented as hooks, the applet isn't required to implement them.**

## Fireside Chats



Tonight's talk: **Template Method and Strategy**  
**compare methods.**

### Template Method

Hey strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method.

I was just kidding. But seriously, what are you doing here? We haven't heard from you in eight chapters.

You might want to remind the reader what you're all about, since it's been so long.

Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. I hate it that way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. It seems like you have to give up control of your algorithms.

### Strategy



It's me, although be careful you and Factory Method are related, aren't you?

I'd heard you were on the final draft of your chapter and thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

I don't know, since Chapter 1, people have been stopping me in the street saying, I haven't heard that pattern... I think they know who I am. But for your sake I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

I'm not sure I'd put it quite like that... and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

**Template Method**

remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

Eah, well, I'm *re* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

How's that? My superclass is abstract.

Like I said strategy, I'm *re* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Got it. Don't call us, we'll call you...

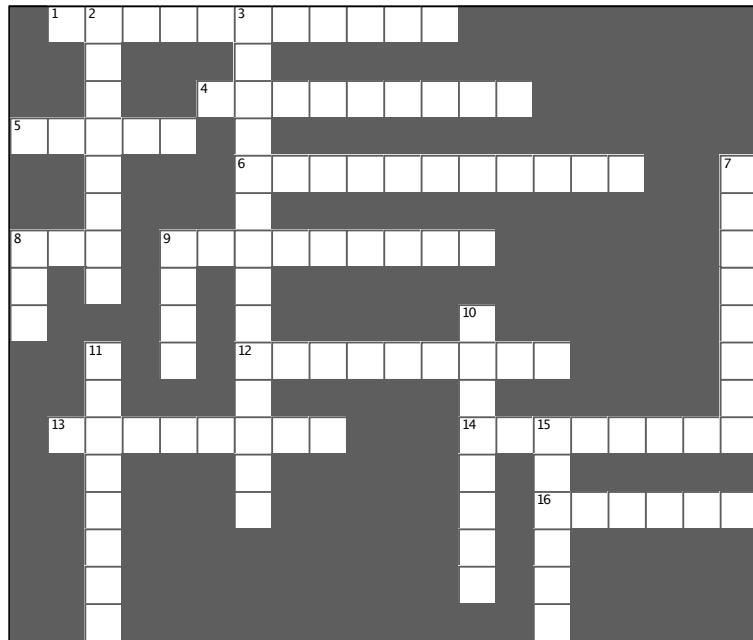
**Strategy**

You might be a little more efficient (just a little) and require fewer objects. You might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose *me* for Chapter for nothing.

Eah, I guess... but, what about dependency? You're way more dependent than me.

But you have to depend on methods implemented in your superclass, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself.

Kay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway, I'm always glad to help.

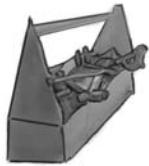


#\$\$

\$": O@ACS PNAN <<<<<< M@AM@=J  
@DAN@?A  
'"; SLA KBNK@PNA@ - M@SN  
("; DA 41M@ A DKKGI A@OK@=ORA KQAN@K@OK  
LN@O 39 PIA  
)"; DA ; AI L@A 6 A@OK@=OAM PNAN  
<<<<<<<< OK @ABME L@AI AJG@OK OK K@AM  
?#NNAN  
+/ K@AA =J@<<<  
, " 0 KJ O?=IHPN RA I@?=@HSKP E@GJKRJ =N @A  
<<<<<<<<< 8 N@?E@A  
\$%" - @I L@A I A@OK@=A@AN @A NALN KB=J  
<<<<<<<<  
\$&" 3J @EN?D=L@MRA C=Q SKP I KMA  
<<<<<<<<  
\$"; DA @I L@A I A@OK@=NP=H@A@=J  
<<<<<<< ?#NN  
\$) "/ #NN O=C@AN RA> L=CAN

" %!

%<<<<<< =OKNED@ NALN=M@ E L@AI AJ@A@  
>S DKKGI A@OK@N  
&" 1=?@NS 6 A@OK@= <<<<<<<<< KB  
; AI L@A 6 A@OK@  
\*"; DA NALN @A =OKNED@ O=CI PNO>A  
NPLLI@>S @A NP>?#NNAN =M PNP=H@?#MA@  
<<<<<<  
+" 2 PAS! 5KPA =J@0 ARAS =I@RAED <<<<<  
LKPJ@N  
, "- I A@OK@= @A =>N@?ONPLA@#NN O=O@AN  
JK@C KMLNK@AN @A@P@>AD=OKM@?@#MA@  
<<<<<< I A@OK@  
\$#". E@ DA=@@L=OAM  
\$\$" 7 PMB@QNEA ?K@AA NDKL @ 7 >FA?@#MA  
\$( "; DA - M@SN ?#NN E L@AI AJ@N@N@I L@A  
I A@OK@=N = <<<<< I A@OK@



## Tools for your Design Toolbox

We've added a plate method to your tool box! With a plate method you can reuse code like a pro while keeping control of your algorithm's flow.

### OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you.

Abstraction  
Encapsulation  
Polymorphism  
Inheritance

### OO Patterns

Strategy  
Decorator  
Adapter  
Factory  
Singleton  
Observer  
Composite  
Visitor  
Facade  
Builder  
Proxy  
Chain of Responsibility  
Command  
Interpreter  
State  
Observer  
Mediator  
Composite  
Visitor  
Facade  
Builder  
Proxy  
Chain of Responsibility  
Command  
Interpreter  
State  
Observer  
Mediator

As Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Thus, Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

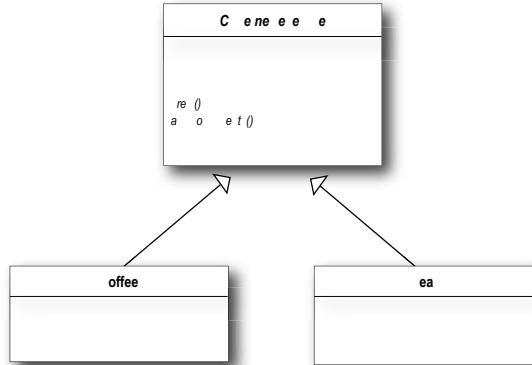
- the template method pattern
- the visitor pattern
- the composite pattern
- the observer pattern
- the facade pattern
- the builder pattern
- the proxy pattern
- the chain of responsibility pattern
- the command pattern
- the interpreter pattern
- the state pattern
- the mediator pattern



Sharpen your pencil

er ise  
solutions

Draw t e new cla dia ram now t at we e mo ed  
prepareRecipe() into t e Ca eineBe era e cla :



### \* WHO DOES WHAT? \*

Match each pattern with its description

#### Pattern

#### Description

em ate etho

n a su ate inter han ab e  
beh a iors an use e e a fion to  
e i e hi h beh a ior to use

trate

ub asses e i e ho  
to im ement ste s in an  
a orithm

a tor etho

ub asses e i e hi b  
on rete asses to reate



## er ise solutions



the iterator and from opposite extremes

## \* ell *managed* \*

# *collections*

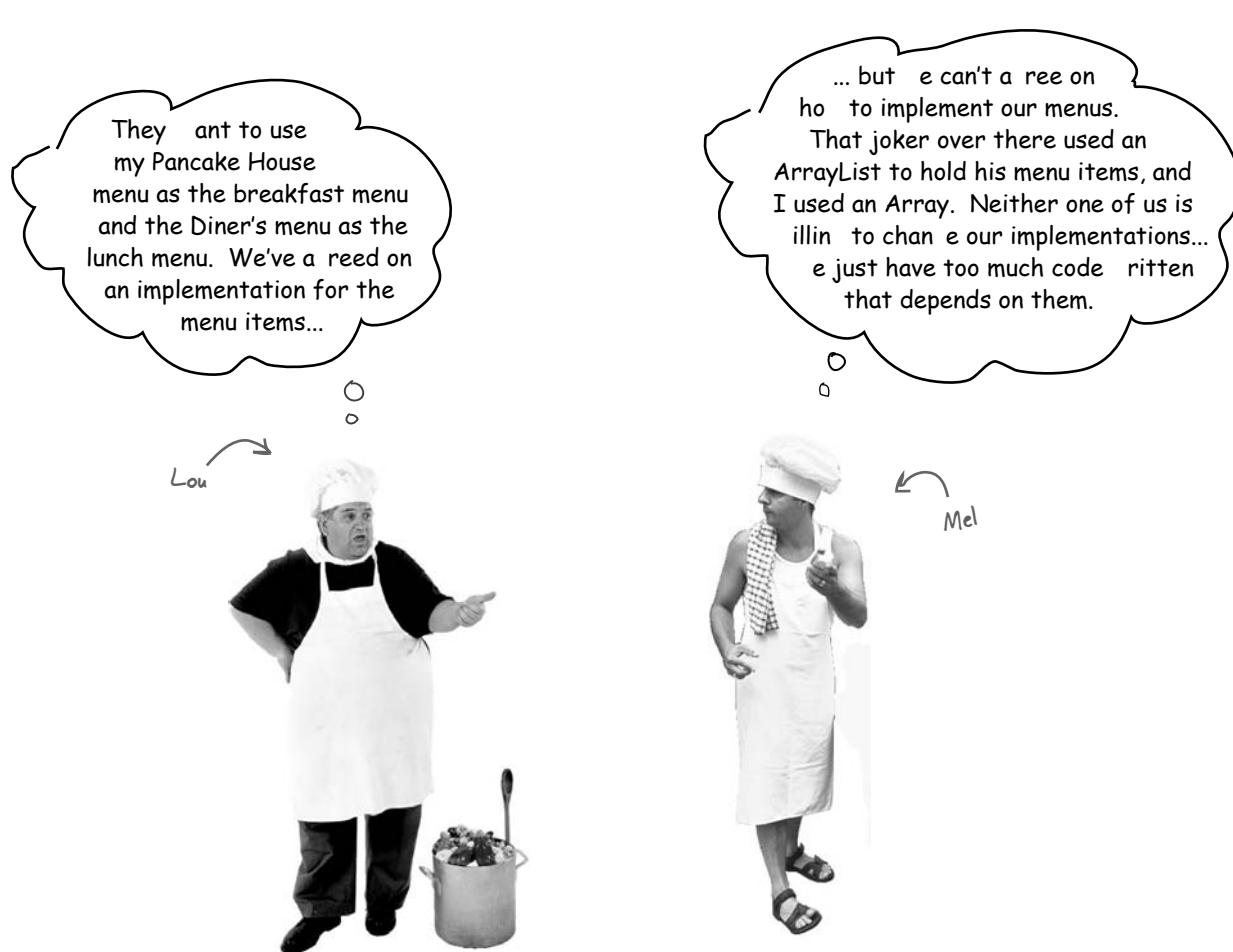


P偷偷

in an Array, a Stack, a List, a Hashtable, take your pick. Each has its own advantages and tradeoffs. But at some point your client is going to want to iterate over the collection, and when he does, are you going to show him your implementation? We certainly hope not! That's two I don't recommend. Well, you don't affect your career; you're going to see how you can allow your client to iterate through your collection without ever getting a peek at how you store your collection. You're also going to learn how to create some clever collections so that it can leap over some implementation details in a single bound. And if that's not enough, you're also going to learn about or two additional concepts.

## Breaking News: Objectville Diner and Objectville Pancake House Merge

hat's great news now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But, there seems to be a slight problem...



# Check out the Menu Items

At least Lou and Mel agree on the implementation of the Menu items. Let's check out the items on each menu, and also take a look at the implementation.

The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

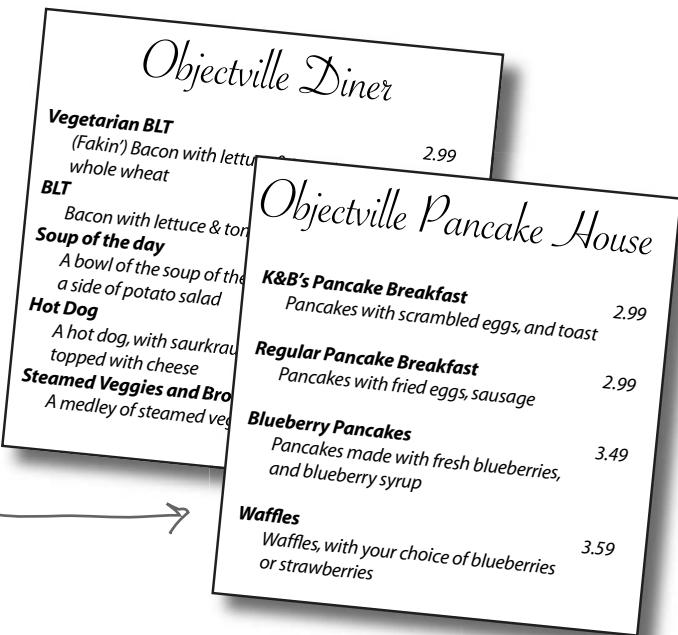
    public MenuItem(String name,
                   String description,
                   boolean vegetarian,
                   double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```



A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

# Lou and Mel's Menu implementations

ow let's take a look at what Lou and Mel are arguing about. hey both have lots of time and code invested in the way they store their menu items in a menu, and lots of other code that depends on it.

Here's Lou's implementation of the Pancake House menu.

```
public class PancakeHouseMenu {
    ArrayList menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

I used an ArrayList so I can easily expand my menu.



Lou's using an ArrayList to store his menu items

Each menu item is added to the ArrayList here, in the constructor

Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price

To add a menu item, Lou creates a new MenuItem object, passing in each argument, and then adds it to the ArrayList

The getMenuItems() method returns the list of menu items

Lou has a bunch of other menu code that depends on the ArrayList implementation. He doesn't want to have to rewrite all that code!



oo

Haah! An ArrayList... I used a REAL Array so I can control the maximum size of my menu and get my MenuItem's without havin' to use a cast.

```

public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with saukraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full! Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }
}

// other menu methods here

```

And here's Mel's implementation of the Diner menu.

Mel takes a different approach; he's using an Array so he can control the max size of the menu and retrieve menu items out without having to cast his objects.

Like Lou, Mel creates his menu items in the constructor, using the addItem() helper method.

addItem() takes all the parameters necessary to create a MenuItem and instantiates one. It also checks to make sure we haven't hit the menu size limit.

Mel specifically wants to keep his menu under a certain size (presumably so he doesn't have to remember too many recipes).

getMenuItems() returns the array of menu items.

Like Lou, Mel has a bunch of code that depends on the implementation of his menu being an Array. He's too busy cooking to rewrite all of this.

# What's the problem with having two different menu representations?

To see why having two different menu representations complicates things, let's try implementing a client that uses the two menus.

Imagine you have been hired by the new company formed by the merger of the Diner and the Pancake House to create a Java enabled waitress (this is Objectville, after all). The spec for the Java enabled waitress specifies that she can print a custom menu for customers on demand, and even tell you if a menu item is vegetarian without having to ask the cook — now that's an innovation!

Let's check out the spec, and then step through what it might take to implement her...

## The Java-Enabled Waitress Specification

```
Java-Enabled Waitress: code-name "Alice"

printMenu()
    - prints every item on the menu

printBreakfastMenu()
    - prints just breakfast items

printLunchMenu()
    - prints just lunch items

printVegetarianMenu()
    - prints all vegetarian menu items

isItemVegetarian(name)
    - given the name of an item, returns true
        if the item is vegetarian, otherwise,
        returns false
```

The Waitress is  
getting Java-enabled.



The spec for  
the Waitress

Let's start by stepping through how we'd implement the printMenu() method

- ① To print all the items on each menu, you'll need to call the getMenuItems() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
```

The method looks  
the same, but the  
calls are returning  
different types.

```
DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The implementation  
is showing through,  
breakfast items are  
in an ArrayList, lunch  
items are in an Array.

- ② Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfast items ArrayList. And to print out the dinner items we'll loop through the array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Now, we have to  
implement two different  
loops to step through  
the two implementations  
of the menu items...

...one loop for the  
ArrayList...

and another for  
the Array.

- ③ Implementing every other method in the Waitress is going to be a variation of this theme. We're always going to need to get both menus and use two loops to iterate through their items. If another restaurant with a different implementation is acquired then we'll have three loops.



## Sharpen your pencil

Based on our implementation of `printMenu()`, which of the following apply?

- A. We are coding to the `PancakeHouseMenu` and `innerMenu` concrete implementations, not to an interface.
- B. The Waitress doesn't implement the `java Waitress` API and so she isn't adhering to a standard.
- C. If we decided to switch from using `innerMenu` to another type of menu that implemented its list of menu items with a `Hashtable`, we'd have to modify a lot of code in the Waitress.
- D. The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- E. We have duplicate code: the `printMenu()` method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
- F. The implementation isn't based on `MML` (`Menu ML`) and so isn't as interoperable as it should be.

## at

Mel and Lou are putting us in a difficult position. They don't want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class. But if one of them doesn't give in, then we're going to have the job of implementing a Waitress that is going to be hard to maintain and extend.

It would really be nice if we could find a way to allow them to implement the same interface for their menus (they're already close, except for the return type of the `getMenuItems()` method).

That way we can minimize the concrete references in the Waitress code and also hopefully get rid of the multiple loops required to iterate over both menus.

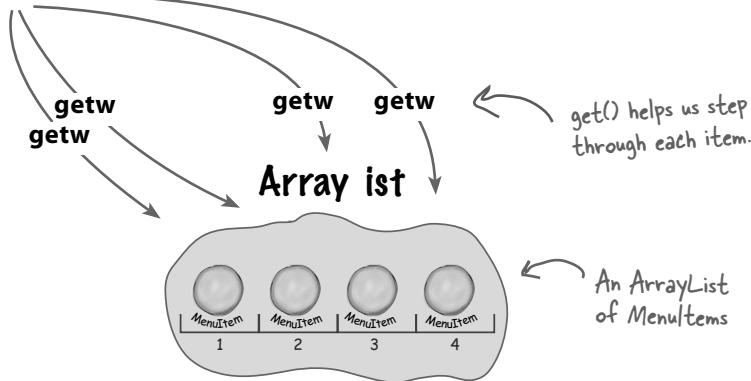
Sound good? Well, how are we going to do that?

## Can we encapsulate the iteration?

If we've learned one thing in this book, it's encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this? Let's work through the idea...

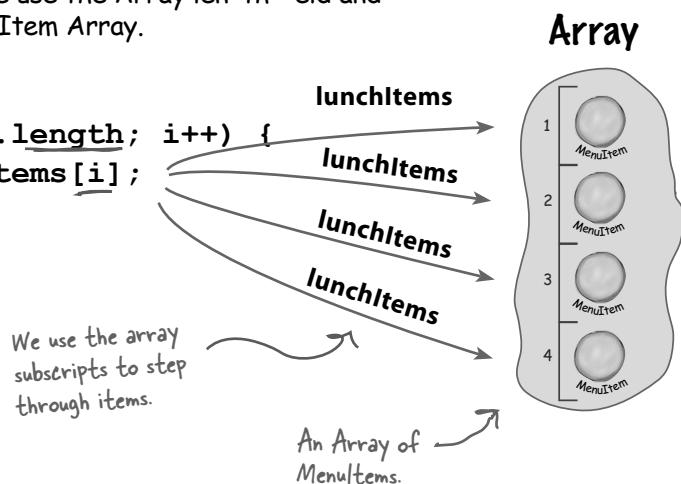
- To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



- And to iterate through the lunch items we use the length field and the array subscript notation on the MenuItem Array.

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



## en a u atin iteration

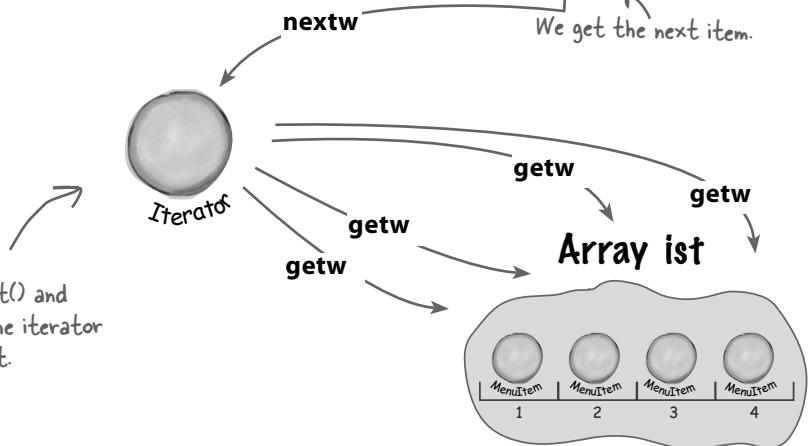
- 3 No what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList

We ask the breakfastMenu for an iterator of its MenuItem s.

```
Iterator iterator = breakfastMenu.createIterator();
```

```
while (iterator.hasNext()) {           ← And while there are more items left...
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.



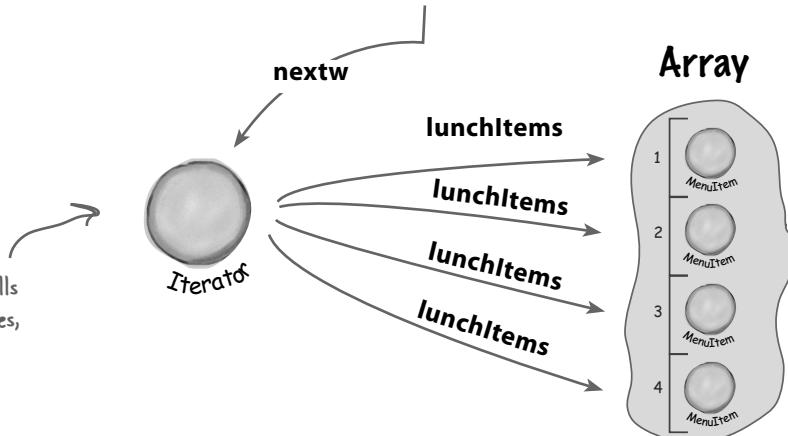
- Let's try that on the Array too:

```
Iterator iterator = lunchMenu.createIterator();
```

```
while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

Wow, this code is exactly the same as the breakfastMenu code.

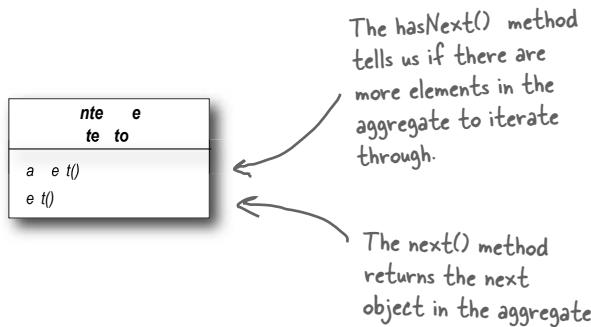
Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.



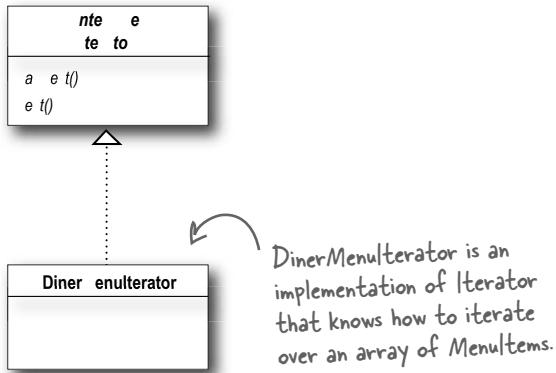
# Meet the Iterator Pattern

Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a design Pattern called the Iterator Pattern.

The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface



Now, once we have this interface, we can implement iterators for any kind of collection of objects arrays, lists, hashtables, ...pick your favorite collection of objects. Let's say we wanted to implement the iterator for the array used in the innerMenu. It would look like this



Let's go ahead and implement this iterator and hook it into the innerMenu to see how this works...

When we say COLLECTION we just mean a group of objects. They might be stored in very different data structures like lists, arrays, hashtables, but they're still collections. We also sometimes call these AGGREGATES.



## Adding an Iterator to DinerMenu

**o add an iterator to the Diner menu we first need to define the iterator interface**

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
}
```

Here's our two methods:

The `hasNext()` method returns a boolean indicating whether or not there are more elements to iterate over...  
...and the `next()` method returns the next element.

**nd now we need to implement a concrete iterator that works for the Diner menu**

```
public class DinerMenuItemIterator implements Iterator {  
    MenuItem[] items;  
    int position = 0;  
  
    public DinerMenuItemIterator(MenuItem[] items) {  
        this.items = items;  
    }  
  
    public Object next() {  
        MenuItem menuItem = items[position];  
        position = position + 1;  
        return menuItem;  
    }  
  
    public boolean hasNext() {  
        if (position >= items.length || items[position] == null) {  
            return false;  
        } else {  
            return true;  
        }  
    }  
}
```

The `hasNext()` method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

We implement the Iterator interface.

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

The `next()` method returns the next item in the array and increments the position.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

# Reworking the Diner Menu with Iterator

**ay, we've got the iterator i e to wor it into the  
Diner enu all we need to do is add one etod to create a  
Diner enu terator and return it to the client**

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // constructor here

    // addItem here

    public MenuItem[] getMenuItems() {
        return menuItems;
    }
}

public Iterator createIterator() {
    return new DinerMenuItemIterator(menuItems);
}

// other menu methods here
```

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuItemIterator is implemented. It just needs to use the iterators to step through the items in the menu.



Go ahead and implement the PancakeHouseIterator interface and make changes needed to incorporate it into the PancakeHouseMenu.

# Fixing up the Waitress code

ow we need to integrate the iterator code into the Waitress. We should be able to get rid of some of the redundancy in the process.

Integration is pretty straightforward first we create a printMenu() method that takes an iterator, then we use the createIterator() method on each menu to retrieve the iterator and pass it to the new method.



```

public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}

// other methods here
}

```

*New and improved with Iterator.*

*In the constructor the Waitress takes the two menus.*

*The printMenu() method now creates two iterators, one for each menu.*

*And then calls the overloaded printMenu() with each iterator.*

*Test if there are any more items.*

*Get the next item.*

*Note that we're down to one loop.*

*Use the item to get name, price and description and print them.*

# Testing our code

It's time to put everything to a test. Let's write some test drive code and see how the Waitress works...

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu); ← First we create the new menus.

        waitress.printMenu(); ← Then we create a
    } ← waitress and pass her the menus.

    } ← Then we print them.
```

Here's the test run...

```
File Edit Window Help GreenE &Ham
% java DinerMenuTestDrive
MENU
-----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saukraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

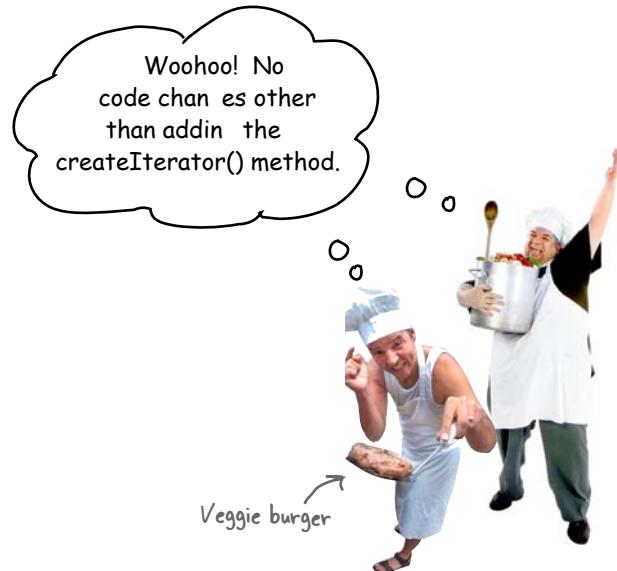
%
```

First we iterate through the pancake menu. And then the lunch menu, all with the same iteration code.

at a e e es a

For starters, we've made our Objectville cooks very happy. They settled their differences and kept their own implementations. Once we gave them a PancakeHouseMenu iterator and a DinerMenu iterator, all they had to do was add a createIterator() method and they were finished.

We've also helped ourselves in the process. The Waitress will be much easier to maintain and extend down the road. Let's go through exactly what we did and think about the consequences.



a t a ta  
at ess le e tat

e  
at ess ee b te at

The Men are not well encapsulated; we can see the Diner items in an Array and the Pancake House in an ArrayList.

We need two loops to iterate through the Men Item.

The Waitress is bound to concrete classes (Men Item and ArrayList).

The Waitress is bound to two different concrete Men classes, despite their interface being almost identical.

The Men implementation are now encapsulated. The Waitress has no idea how the Men hold their collection of men item.

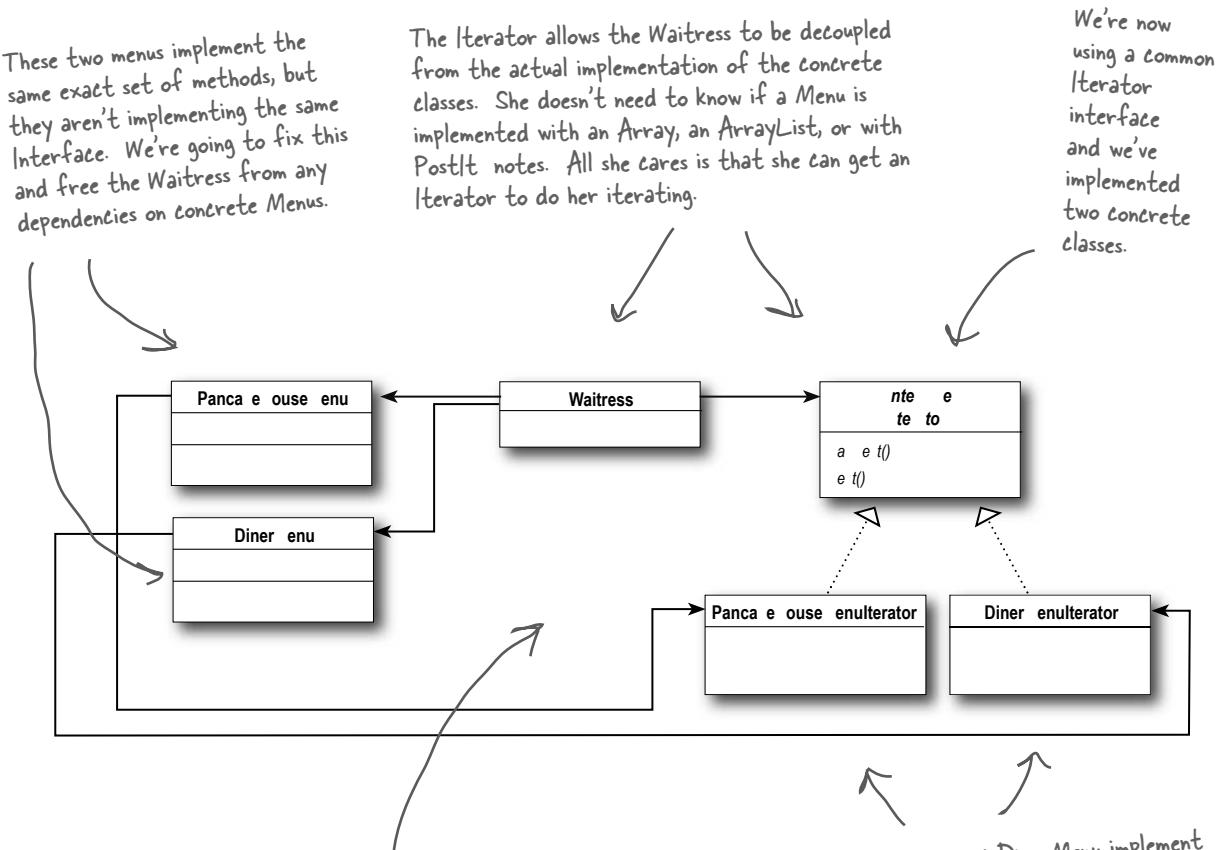
All we need is a loop that polymorphically handles any collection of item along a it implements Iterator.

The Waitress now uses an interface (Iterator).

The Men interface are now exactly the same and, so, we still don't have a common interface, which means the Waitress is still bound to two concrete Men classes. We'd better start.

# What we have so far...

Before we clean things up, let's get a bird's eye view of our current design.



Note that the iterator give us a way to step through the elements of an aggregate without forcing the aggregate to clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the iterator to live outside of the aggregate; in other words, we've encapsulated the iteration.

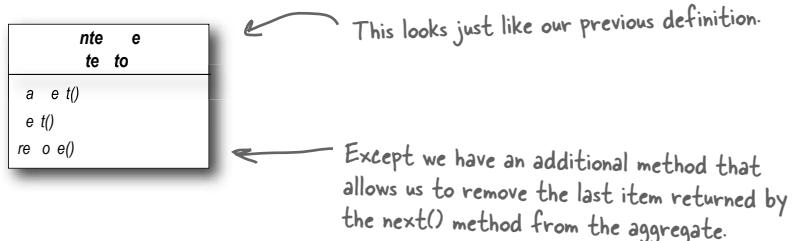
PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items implementations.

## Making some improvements...

key, we know the interfaces of PancakeHouseMenu and innerMenu are exactly the same and yet we haven't defined a common interface for them. So, we're going to do that and clean up the Waitress a little more.

You may be wondering why we're not using the Iterator interface - we did that so you could see how to build an iterator from scratch. Now that we've done that, we're going to switch to using the Iterator interface, because we'll get a lot of leverage by implementing that instead of our home grown iterator interface. What kind of leverage? You'll soon see.

First, let's check out the java.util.Iterator interface



This is going to be a piece of cake. We just need to change the interface that both PancakeHouseMenu and innerMenu extend, right? Almost... actually, it's even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator. In other words, we never needed to implement our own iterator for ArrayList. However, we'll still need our implementation for the innerMenu because it relies on an array, which doesn't support the iterator() method (or any other way to create an array iterator).

---

there are no  
Dumb Questions

**Q:** What if I don't want to provide the ability to remove something from the underlying collection of objects?

**A:**

**Q:** How does remove() behave under multiple threads that may be using different iterators over the same collection of objects?

**A:**

## Cleaning things up with java.util.Iterator

Let's start with the PancakeHouseMenu, changing it over to java.util. terator is going to be easy. We just delete the PancakeHouseMenu terator class, add an import java.util. terator to the top of PancakeHouseMenu and change one line of the PancakeHouseMenu

```
public Iterator createIterator() {
    return menuItems.iterator();
}
```

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList.

nd that's it, PancakeHouseMenu is done.

ow we need to make the changes to allow the innerMenu to work with java.util. terator.

```
import java.util.Iterator;

public class DinerMenuItemIterator implements Iterator {
    MenuItem[] list;
    int position = 0;

    public DinerMenuItemIterator(MenuItem[] list) {
        this.list = list;
    }

    public Object next() {
        //implementation here
    }

    public boolean hasNext() {
        //implementation here
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

First we import java.util.Iterator, the interface we're going to implement.

None of our current implementation changes...

...but we do need to implement remove(). Here, because the chef is using a fixed sized Array, we just shift all the elements up one when remove() is called.

## We are almost there...

We just need to give the Menus a common interface and rework the Waitress a little. The Menu interface is quite simple we might want to add a few more methods to it eventually, like addItem(), but for now we will let the chefs control their menus by keeping that method out of the public interface

```
public interface Menu {  
    public Iterator createIterator();  
}
```

This is a simple interface that just lets clients get an iterator for the items in the menu.

Now we need to add an implements Menu to both the PancakeHouseMenu and the DinerMenu class definitions and update the Waitress

```
import java.util.Iterator;
```

Now the Waitress uses the java.util.Iterator as well.

```
public class Waitress {  
    Menu pancakeHouseMenu;  
    Menu dinerMenu;  
  
    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {  
        this.pancakeHouseMenu = pancakeHouseMenu;  
        this.dinerMenu = dinerMenu;  
    }  
  
    public void printMenu() {  
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();  
        Iterator dinnerIterator = dinnerMenu.createIterator();  
        System.out.println("MENU\n----\nBREAKFAST");  
        printMenu(pancakeIterator);  
        System.out.println("\nLUNCH");  
        printMenu(dinnerIterator);  
    }  
  
    private void printMenu(Iterator iterator) {  
        while (iterator.hasNext()) {  
            MenuItem menuItem = (MenuItem) iterator.next();  
            System.out.print(menuItem.getName() + ", ");  
            System.out.print(menuItem.getPrice() + " -- ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
  
    // other methods here  
}
```

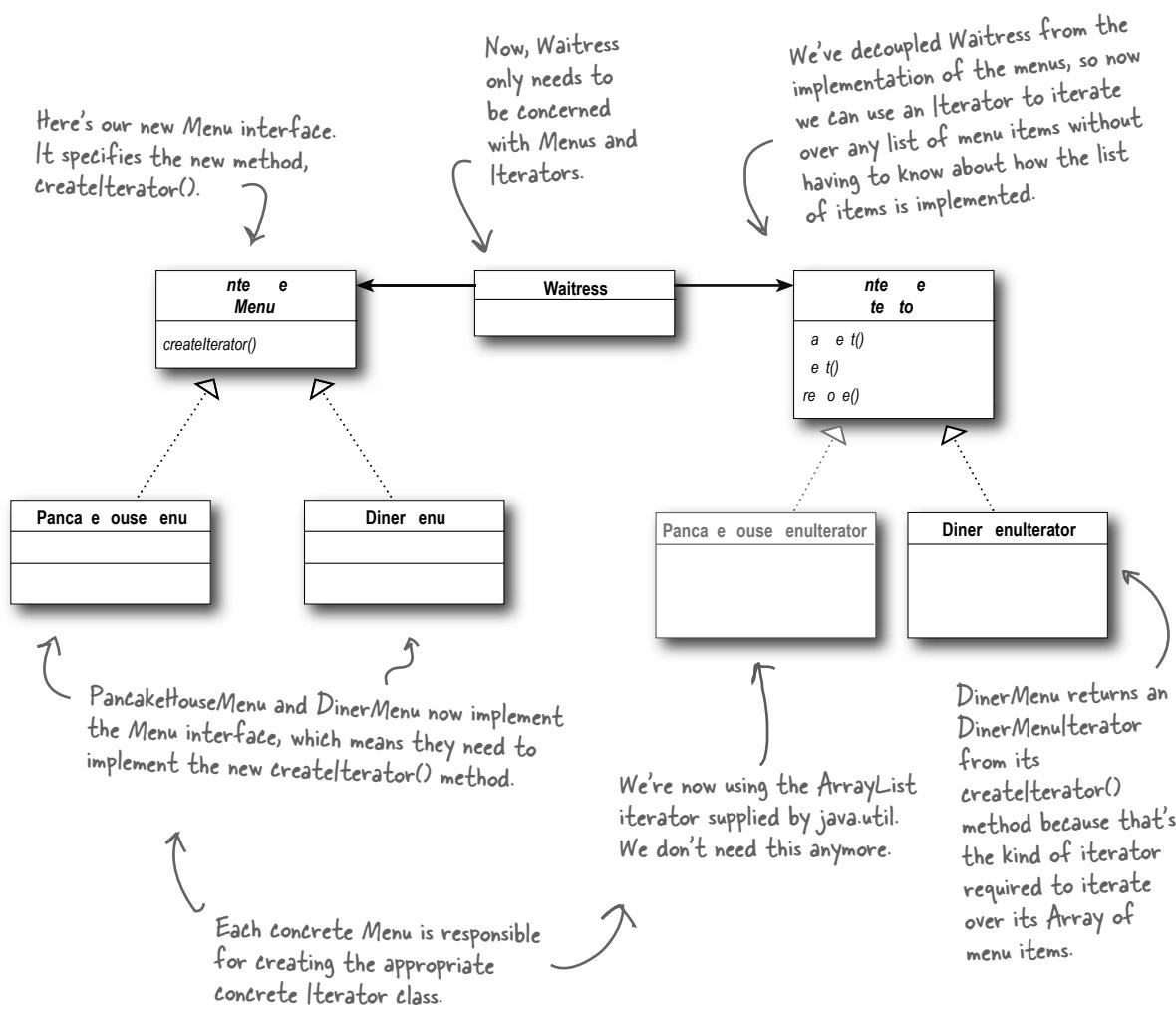
We need to replace the concrete Menu classes with the Menu Interface.

Nothing changes here.

# What does this get us?

The `PancakeHouseMenu` and `DinerMenu` classes implement an interface, `Menu`. `Waitress` can refer to each menu object using the interface rather than the concrete class. So, we're reducing the dependency between the `Waitress` and the concrete classes by programming to an interface, not an implementation.

The new Menu interface has one method, `createTerminator()`, that is implemented by `PancakeHouseMenu` and `ChineseMenu`. Each menu class assumes the responsibility of creating a concrete terminator that is appropriate for its internal implementation of the menu items.



## Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection oriented classes (the `ArrayList`). Now it's time to check out the official definition of the pattern.

**The Iterator pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates just like the `printMenu()` method, which doesn't care if the menu items are held in an array or `ArrayList` (or anything else that can create an iterator), as long as it can get hold of an iterator.

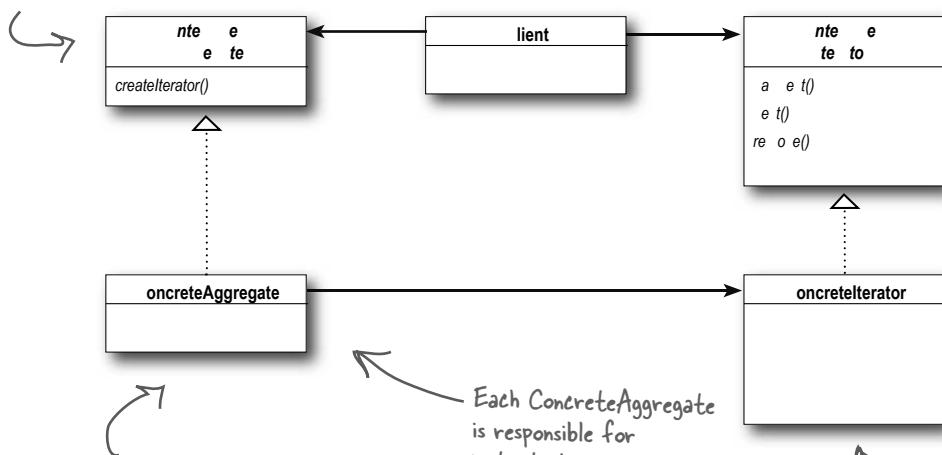
The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Let's check out the class diagram to put all the pieces in context...

**The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.**

**It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.**

Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.



The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteIterator is responsible for managing the current position of the iteration.

The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the `java.util.Iterator`. If you don't want to use Java's Iterator interface, you can always create your own.



The class diagram of the Iterator Pattern looks very similar to another Pattern you've studied; can you figure out what it is? Hint: A class decides which object to create.

## there are no Dumb Questions

**Q:** I've seen other books show the Iterator class diagram with the methods `first`, `next`, `isDone` and `currentItem`. Why are these methods different?

**A:**

**Q:** Could I implement an iterator that can go backwards as well as forwards?

**A:**

**Q:** If I'm using `ArrayList` I always want to use the `java.util.Iterator` interface so I can use my own iterator implementations with classes that are already using the `java` iterators?

**A:**

**Q:** I've heard about internal iterators and external iterators. What are they? Which kind did we implement in the example?

**A:**

**Q:** Who defines the ordering of the iteration in a collection like `ArrayList` which are inherently unordered?

**A:**

**Q:** I've seen an `Enumeration` interface in `java`; does that implement the Iterator Pattern?

**A:**

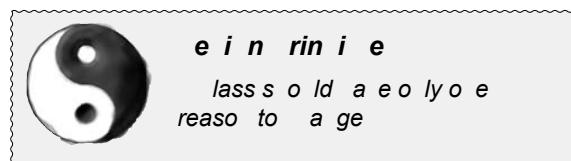
**Q:** You said we can write polymorphic code using an iterator; can you explain that more?

**A:**

# Single Responsibility

**What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?**

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Who? Up, two it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CH-G is at the center of another design principle



We know we want to avoid change in a class like the plague modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

The solution? The principle guides us to assign each responsibility to one class, and only one class.

That's right, it's as easy as that, and then again it's not separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

**Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.**

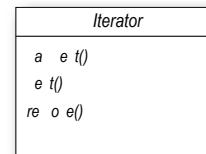
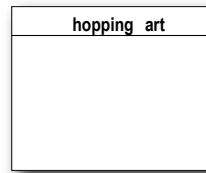
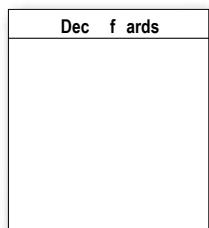
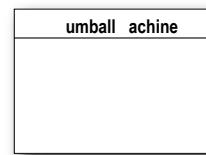
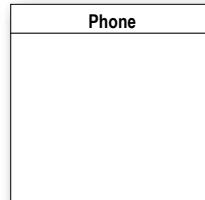
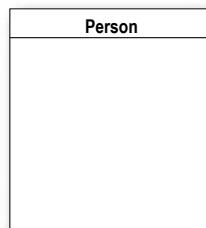
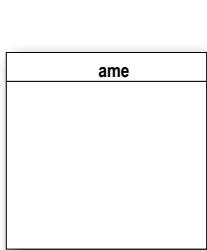
**This principle guides us to keep each class to a single responsibility.**



**o e on i** a term yo ll  
ear ed a a mea re o  
ow clo ely a cla or a  
mod le pport a in le  
p rope or re pon i lity.  
  
We ay t at a mod le or  
cla a ig o esio  
wen it i de i ned aro nd a et o  
related nction , and we ay it a lo  
o esio wen it i de i ned aro nd a  
et o nrelated nction .  
  
Co e ion i a more eneral concept  
t ant e Sin le Re pon i lity Principle,  
t t e two are clo ely related.  
Cla e t at ad ere to t e principle  
tend to a e i co e ion and are  
more maintaina let an cla e t at  
ta e on m ltiple re pon i lity and  
a e low co e ion.



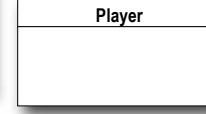
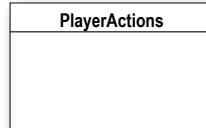
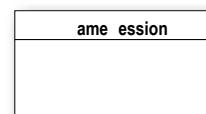
E amine t e ecla e and determine w ic one a em ltiple re pon i illtie .



**HARD HAT AREA, WATCH OUT  
FOR FALLING ASSUMPTIONS**



Determine i t e ecla e a e low or i co e ion.





o O

Good thin you're learnin  
about the Iterator pattern  
because I just heard that Objectville  
Mer ers and Ac uisitions has done  
another deal... e're mer in ith  
Objectville Caf and adoptin their  
dinner menu.



o O

Wo , and e thou ht thin s  
ere already complicated.  
No what are e oin to do?



o O

Come on,  
think positively, I'm  
sure e can nd a ay to  
ork them into the  
Iterator Pattern.

# Taking a look at the Caf Menu

Here's the Caf Menu. It doesn't look like too much trouble to integrate the Cafe Menu into our framework... let's check it out.

```

public class CafeMenu {
    Hashtable menuItems = new Hashtable(); ← The Caf is storing their menu items in a Hashtable.
    ← CafeMenu doesn't implement our new Menu
    interface, but this is easily fixed. Does that support Iterator? We'll see shortly...
    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }
    public void addItem(String name, String description,
                        boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem); ← Here's where we create a new MenuItem
                                                    and add it to the menuItems hashtable.
    }
    public Hashtable getItems() { ← the key is the item name. ← the value is the menuItem object.
        return menuItems;
    }
}

```

Like the other Menus, the menu items are initialized in the constructor.

We're not going to need this anymore.

## Sharpen your pencil

Be ore loo in att e ne t pa e, ic ly ot down t e t reet in  
we a e to do to t i code to t it into o r ramewor :

1.

2.

3.

## Reworking the Caf Menu code

Integrating the Cafe Menu into our framework is easy. Why? Because Hashtable is one of those ava collections that supports iterator. But it's not quite the same as ArrayList...

```
public class CafeMenu implements Menu {
    Hashtable menuItems = new Hashtable();
    public CafeMenu() {
        // constructor code here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Hashtable getItems() {
        return menuItems;
    }
    public Iterator createIterator() {
        return menuItems.values().iterator();
    }
}
```

*CafeMenu implements the Menu interface, so the Waitress can use it just like the other two Menus.*

*We're using Hashtable because it's a common data structure for storing values; you could also use the newer HashMap.*

*Just like before, we can get rid of getItems() so we don't expose the implementation of menuItems to the Waitress.*

*And here's where we implement the createIterator() method. Notice that we're not getting an Iterator for the whole Hashtable, just for the values.*



### Code Up Close

Hashtable is a little more complicated than ArrayList because it supports only key and value, so we can still get an Iterator for the value (which are the MenuItem).

```
public Iterator createIterator() {
    return menuItems.values().iterator();
}
```

First we get the values of the Hashtable, which is just a collection of all the objects in the hashtable.

Luckily that collection supports the iterator() method, which returns a object of type java.util.Iterator.

# Adding the Caf Menu to the Waitress

hat was easy; how about modifying the Waitress to support our new Menu? Now that the Waitress expects iterators, that should be easy too.

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;
```

The Caf menu is passed into the Waitress in the constructor with the other menus, and we stash it in an instance variable.

```
public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
    this.cafeMenu = cafeMenu;
}
```

```
public void printMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator dinnerIterator = dinerMenu.createIterator();
    Iterator cafeIterator = cafeMenu.createIterator();
    System.out.println("MENU\n---\nBREAKFAST");
    printMenu(pancakeIterator);
    System.out.println("\nLUNCH");
    printMenu(dinnerIterator);
    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

We're using the Caf's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

```
private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
        MenuItem menuItem = (MenuItem) iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
}
```

Nothing changes here

# Breakfast, lunch AND dinner

Let's update our test drive to make sure this all works.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu(); Create a CafeMenu...
        ← ... and pass it to the waitress.

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu); ←

        waitress.printMenu(); ← Now, when we print we should see all three menus.
    }
}
```

Here's the test run check out the new dinner menu from the Caf

```
File Edit Window Help Kat y&BertLi ePanca e
% java DinerMenuTestDrive
MENU
-----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saukraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
lettuce, tomato, and fries
%
```

First we iterate through the pancake menu.

And then the dinner menu.

And finally the new caf menu, all with the same iteration code.

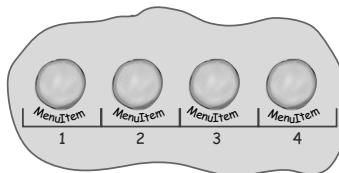
## What did we do?



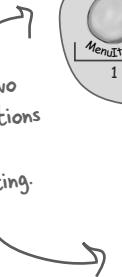
We wanted to give the Waitress an easy way to iterate over menu items...

... and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.



ArrayList



Array

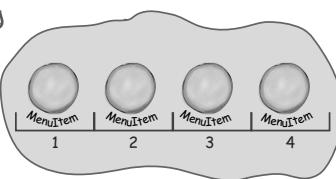
## We decoupled the Waitress....

So we gave the Waitress an Iterator for each kind of group of objects she needed to iterate over...

... one for ArrayList...

ArrayList has a built in iterator...

ArrayList



nextw

nextw

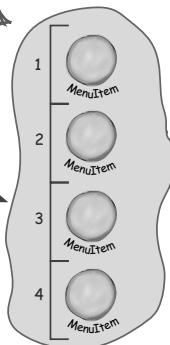
... and one for Array.

nextw

Now she doesn't have to worry about which implementation we used; she always uses the same interface - Iterator - to iterate over menu items. She's been decoupled from the implementation.

... Array doesn't have a built in Iterator so we built our own.

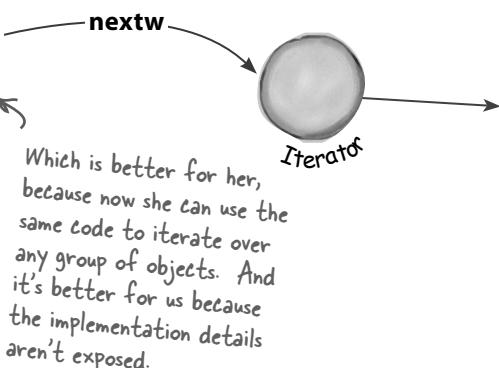
Array



## ... and we made the Waitress more extensible



By giving her an Iterator we have decoupled her from the implementation of the menu items, so we can easily add new Menus if we want.



### Hashtable



We easily added another implementation of menu items, and since we provided an Iterator, the Waitress knew what to do.

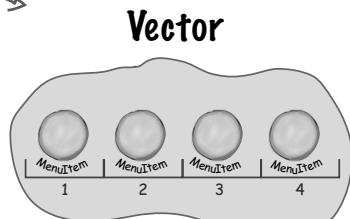
Making an Iterator for the Hashtable values was easy; when you call `values.iterator()` you get an Iterator.

## But there's more

Java gives you a lot of "collection" classes that allow you to store and retrieve groups of objects. For example, Vector and LinkedList.

Most have different interfaces.

But almost all of them support a way to obtain an Iterator.



And if they don't support Iterator, that's ok, because now you know how to build your own.

### inked list



...and more

# Iterators and Collections

We've been using a couple of classes that are part of the `java Collections Framework`.

This framework is just a set of classes and interfaces, including `ArrayList`, which we've been using, and many others like `Vector`, `LinkedList`, `Stack`, and `Priority Queue`. Each of these classes implements the `java.util.Collection` interface, which contains a bunch of useful methods for manipulating groups of objects.

Let's take a quick look at the interface



int e	Co e t o n
a ()	
a ()	
clear()	
co ta ()	
co ta ()	
e a ()	
a o e()	
t ()	
terator()	←
re o e()	
re o e()	
reta ()	
e()	
to arra ()	←

As you can see, there's all kinds of good stuff here. You can add and remove elements from your collection without even knowing how it's implemented.

Here's our old friend, the `Iterator()` method. With this method, you can get an `Iterator` for any class that implements the `Collection` interface.

Other handy methods include `size()`, to get the number of elements, and `toArray()` to turn your collection into an array.



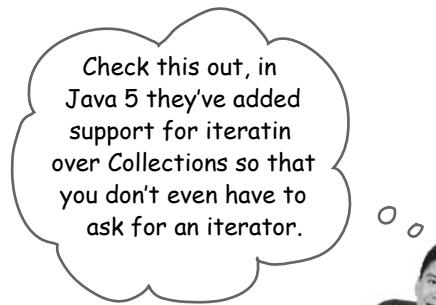
Watch it!

Ha ta le i one o a ew cla e t at i dire tly pport `Iterator`. A yo aw wen we implemented t e Ca eMen , yo co ld et an Iterator rom it, t only y r t retrie in it Collection called al e . I yo t in a o tit, t i ma e en e: t e Ha ta le old two et o o ect : ey and al e . I we want to iterate o er it al e , we r t need to retrie et em rom t e Ha ta le, and t en o tain t e iterator.

The nice thin about Collections and `Iterator` is that each Collection object kno s ho to create its o n `Iterator`. Callin `iterator()` on an `ArrayList` returns a concrete `Iterator` made for `ArrayLists`, but you never need to see or worry about the concrete class it uses; you just use the `Iterator` interface.



# Iterators and Collections in ava



ava includes a new form of the **or** statement, called **or/**, that lets you iterate over a collection or an array without creating an iterator explicitly.

o use **or/**, you use a **or** statement that looks like

Iterates over each object in the collection.

obj is assigned to the next element in the collection each time through the loop.

```
for (Object obj: collection) {  
    ...  
}
```

Here's how you iterate over an arrayList using **or/**

```
ArrayList items = new ArrayList();  
items.add(new MenuItem("Pancakes", "delicious pancakes", true, 1.59);  
items.add(new MenuItem("Waffles", "yummy waffles", true, 1.99);  
items.add(new MenuItem("Toast", "perfect toast", true, 0.59);
```

```
for (MenuItem item: items) {  
    System.out.println("Breakfast item: " + item);  
}
```

Iterate over the list and print each item.

Load up an ArrayList of MenuItem's.



Watch it!

o need to e Ja a 5 new eneric eat re to en re or/ in type a ety. Ma e re yo read p on t e detail e ore in eneric and or/in.

# Code Magnets



The hefs have decided that they want to be able to alternate their lunch menu items in other words, they will offer some items on Monday, Wednesday, Friday and Sunday, and other items on Tuesday, Thursday, and Saturday. Someone already wrote the code for a new Alternating DinerMenu Iterator so that it alternates the menu items, but they scrambled it up and put it on the fridge in the Diner as a joke. You put it back together. Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
{
```

```
public AlternatingDinerMenuItemIterator(MenuItem[] items)
```

```
this.items = items;
Calendar rightNow = Calendar.getInstance();
position = rightNow.get(Calendar.DAY_OF_WEEK) % 2;
```

```
implements Iterator
```

```
public void remove() {
```

```
MenuItem[] items;
int position;
```

```
}
```

```
public class AlternatingDinerMenuItemIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");
```

```
if (position >= items.length || items[position] == null) {
    return false;
} else {
    return true;
}
```

```
}
```



## Is the Waitress ready for prime time?

The Waitress has come a long way, but you've gotta admit those three calls to `printMenu()` are looking kind of ugly.

Let's be real, every time we add a new menu we are going to have to open up the Waitress implementation and add more code. Can you say violating the Open/Closed Principle?

```
public void printMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator dinerIterator = dinerMenu.createIterator();
    Iterator cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

Three `createIterator()` calls.

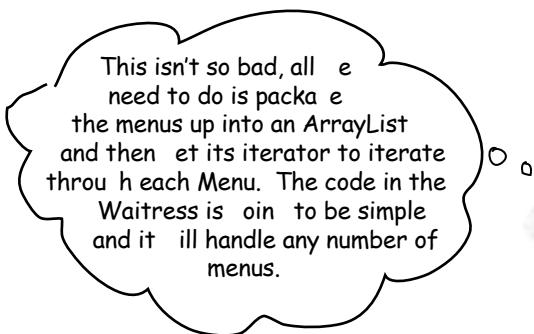
Three calls to `printMenu()`.

Everytime we add or remove a menu we're going to have to open this code up for changes.

It's not the Waitress' fault. We have done a great job of decoupling the menu implementation and extracting the iteration into an iterator. But we still are handling the menus with separate, independent objects - we need a way to manage them together.



The Waitress still needs to make three calls to `printMenu()`, one for each menu. Can you think of a way to combine the menus so that only one call needs to be made? Or perhaps at one Iterator is passed to the Waitress to iterate over all the menus?



Sounds like the chef is on to something. Let's give it a try

```
public class Waitress {
    ArrayList menus;

    public Waitress(ArrayList menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = (Menu)menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Now we just take an ArrayList of menus.

And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

No code changes here.

This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.

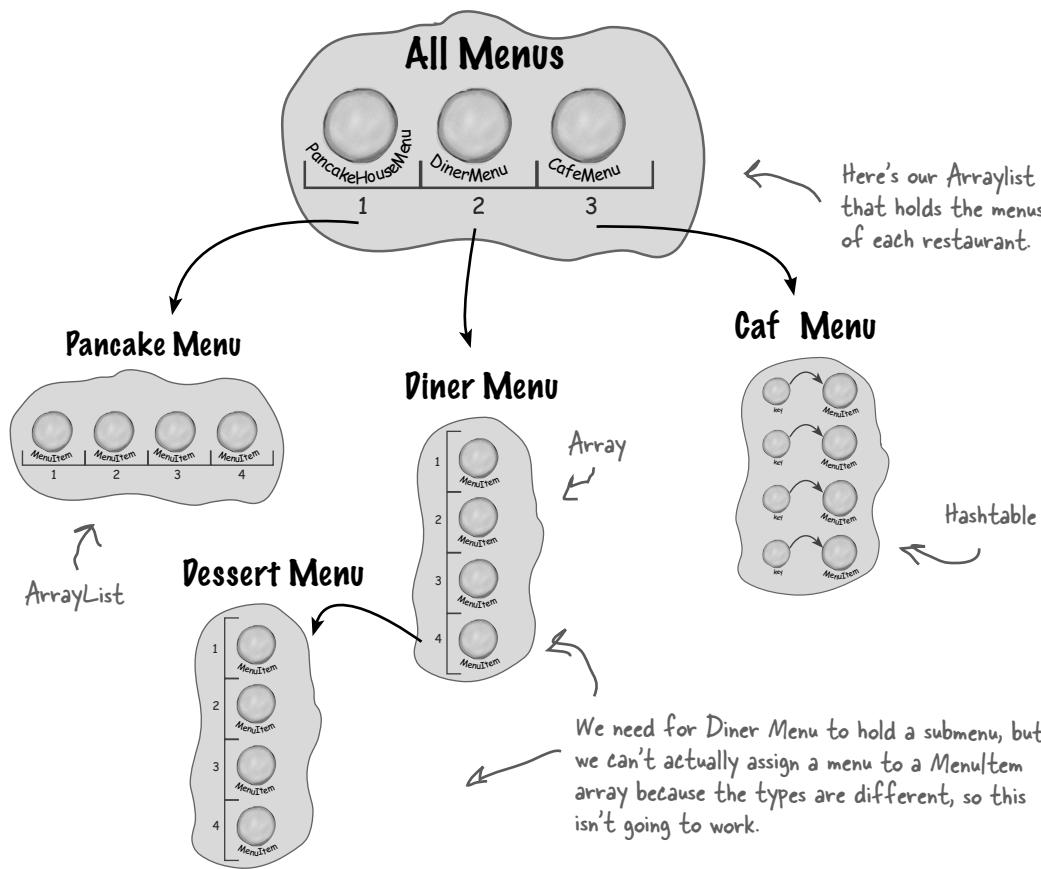
# ust when we thought it was safe...

## Now they want to add a dessert su enu

kay, now what? Now we have to support not only multiple menus, but menus within menus.

t would be nice if we could just make the dessert menu an element of the innerMenu collection, but that won't work as it is now implemented.

## hat we want so ething li e this



ut this  
won't wor

**e can't assign a dessert enu to  
a enu te array**

**i e for a change**

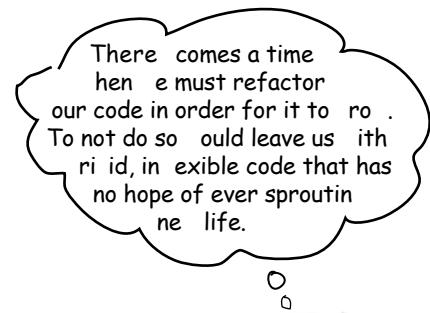
## at e ee

he time has come to make an executive decision to rework the chef's implementation into something that is general enough to work over all the menus (and now sub menus). That's right, we're going to tell the chefs that the time has come for us to reimplement their menus.

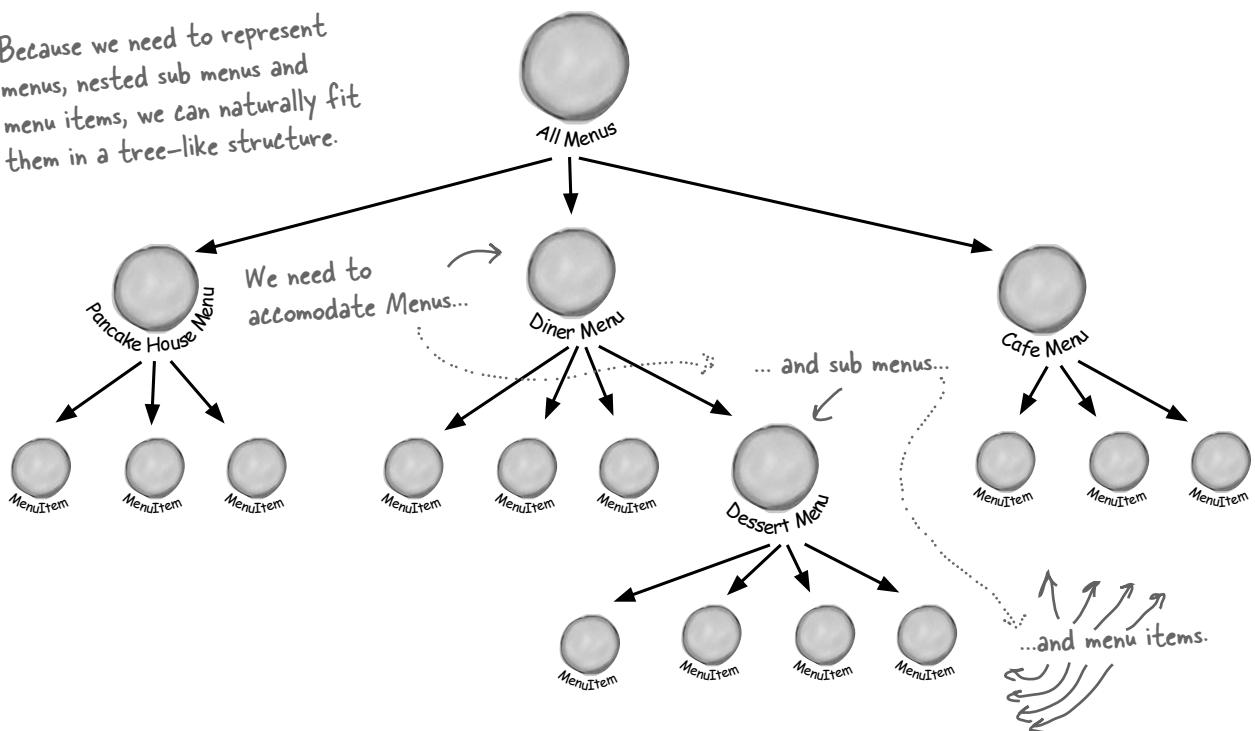
he reality is that we've reached a level of complexity such that if we don't rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

o, what is it we really need out of our new design?

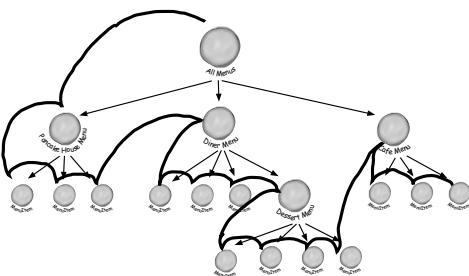
- We need some kind of a tree shaped structure that will accommodate menus, submenus and menu items.
- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
- We may need to be able to traverse the items in a more flexible manner. For instance, we might need to iterate over only the dinner's dessert menu, or we might need to iterate over the dinner's entire menu, including the dessert submenu.



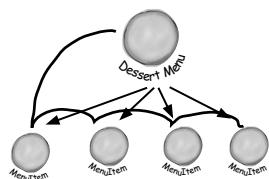
Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.



We still need to be able to traverse all the items in the tree.



We also need to be able to traverse more flexibly, for instance over one menu.



How would you handle this new writing requirement? Think about it before reading the page.

# The Composite Pattern defined

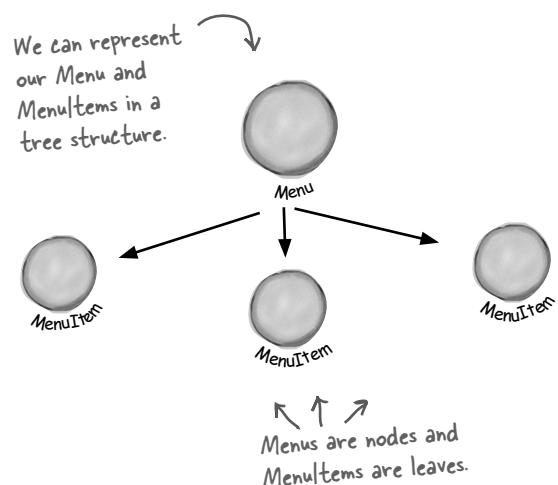
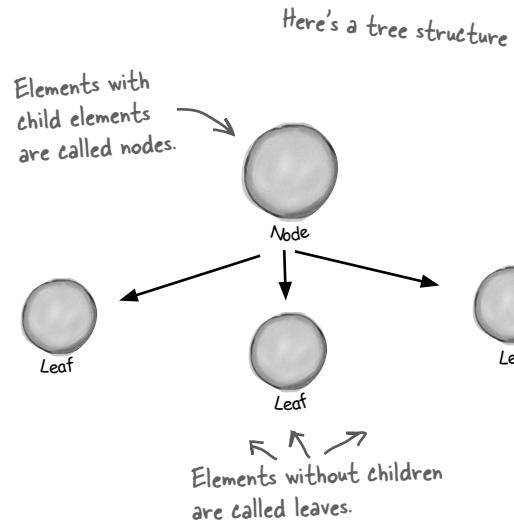
hat's right, we're going to introduce another pattern to solve this problem. We didn't give up on iterator—it will still be part of our solution—however, the problem of managing menus has taken on a new dimension that iterator doesn't solve. So, we're going to step back and solve it with the Composite Pattern.

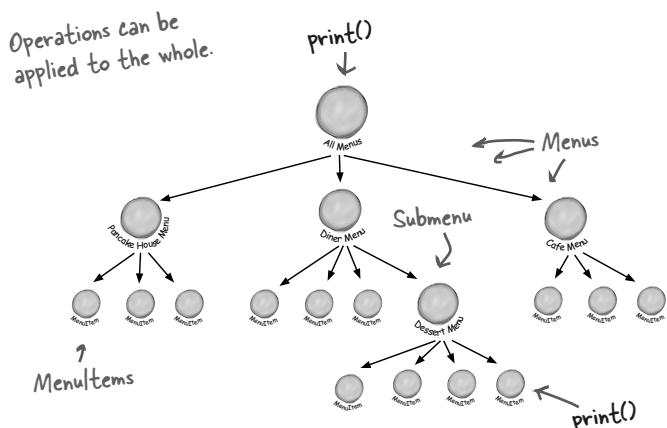
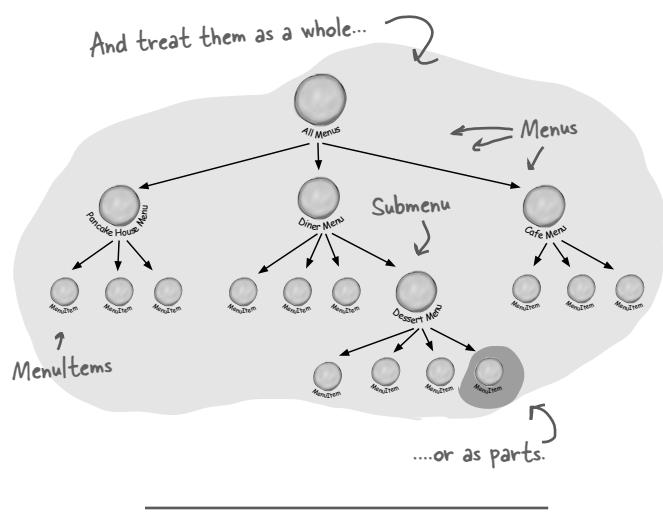
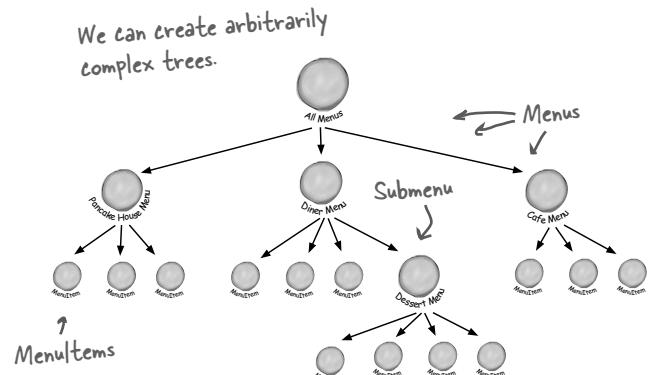
We're not going to beat around the bush on this pattern, we're going to go ahead and roll out the official definition now.

**he composite pattern** allows you to compose objects into tree structures to represent part whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus—menu items in the same structure. By putting menus and items in the same structure we create a part whole hierarchy; that is, a tree of objects that is made of parts (menus and menu items) but that can be treated as a whole, like one big menu.

Once we have our menu, we can use this pattern to treat individual objects and compositions uniformly. What does that mean? It means if we have a tree structure of menus, submenus, and perhaps subsubmenus along with menu items, then any menu is a composition because it can contain both other menus and menu items. The individual objects are just the menu items—they don't hold other objects. As you'll see, using a design that follows the Composite Pattern is going to allow us to write some simple code that can apply the same operation (like printing) over the entire menu structure.

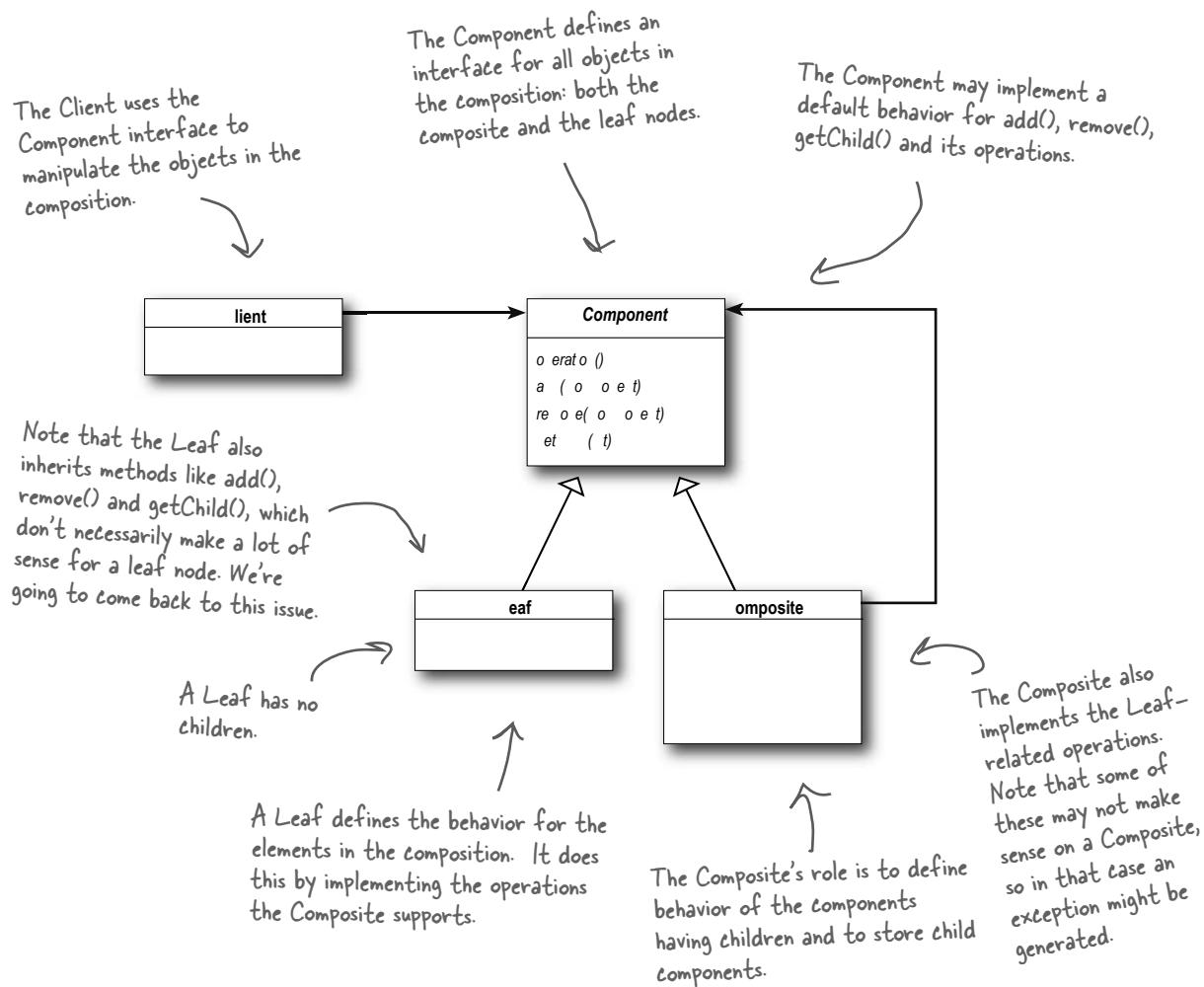




**The Composite Pattern** allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.

## o o ite attern a dia ra



Q: component composite trees?  
I'm confused

A:

there are no  
Dumb Questions

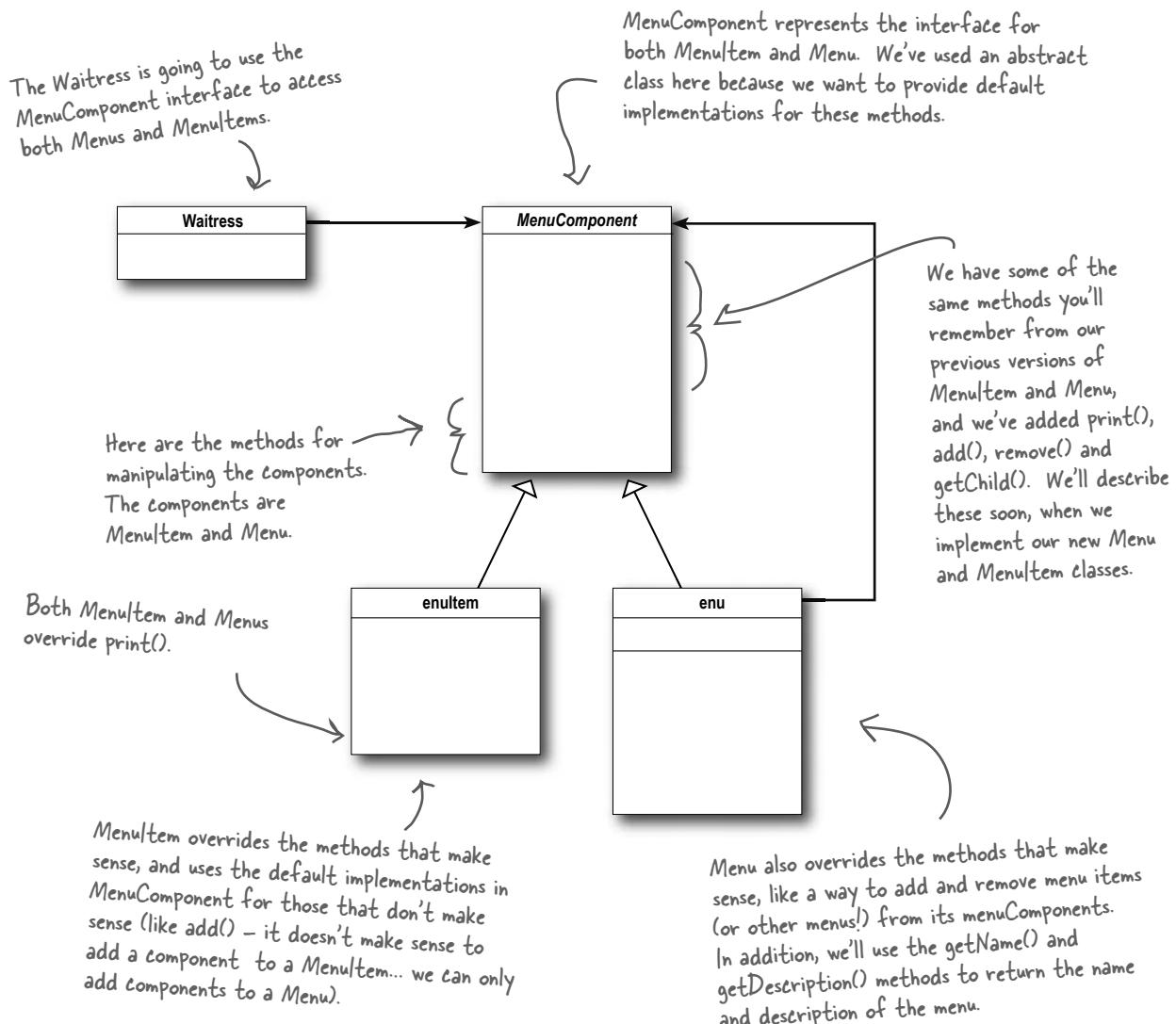
A:

Q: How does this relate to iterators?

# Designing Menus with Composite

o, how do we apply the Composite Pattern to our menus? o start with, we need to create a component interface; this acts as the common interface for both menus and menu items and allows us to treat them uniformly. n other words we can call the *s me* method on menus or menu items.

ow, it may not make *se se* to call some of the methods on a menu item or a menu, but we can deal with that, and we will in just a moment. But for now, let's take a look at a sketch of how the menus are going to fit into a Composite Pattern structure



Okay, we're going to start with the `MenuComponent` abstract class; remember, the role of the menu component is to provide an interface for the leaf nodes and the composite nodes. Now you might be asking, isn't the `MenuComponent` playing two roles? It might well be and we'll come back to that point. However, for now we're going to provide a default implementation of the methods so that if the `MenuItem` (the leaf) or the `Menu` (the composite) doesn't want to implement some of the methods (like `getChild()` for a leaf node) they can fall back on some basic behavior.

`MenuComponent` provides default implementations for every method.

```
public abstract class MenuComponent {
    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

If components just implement the `menuComponent` interface however, because leaves and nodes have different roles we can't always define a default implementation for each method that makes sense otherwise the best you can do is throw a runtime exception.

Because some of these methods only make sense for `MenuItem`s, and some only make sense for `Menus`, the default implementation is `UnsupportedOperationException`. That way, if `MenuItem` or `Menu` doesn't support an operation, they don't have to do anything, they can just inherit the default implementation.



We've grouped together the "composite" methods – that is, methods to add, remove and get `MenuComponents`.



Here are the "operation" methods; these are used by the `MenuItem`s. It turns out we can also use a couple of them in `Menu` too, as you'll see in a couple of pages when we show the `Menu` code.

`print()` is an "operation" method that both our `Menus` and `MenuItem`s will implement, but we provide a default operation here.

# Implementing the MenuItem

Okay, let's give the Menu item class a shot. Remember, this is the leaf class in the Composite diagram and it implements the behavior of the elements of the composite.

```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print(" " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println(" -- " + getDescription());
    }
}
```

First we need to extend the `MenuComponent` interface.

The constructor just takes the name, description, etc. and keeps a reference to them all. This is pretty much like our old menu item implementation.

Here's our getter methods – just like our previous implementation.

This is different from the previous implementation. Here we're overriding the `print()` method in the `MenuComponent` class. For `MenuItem` this method prints the complete menu entry: name, description, price and whether or not it's veggie.

I'm lad e're oin in  
this direction, I'm thinkin' this is  
oin to give me the exhibitiy I need  
to implement that cr pe menu I've  
al ays anted.



# Implementing the Composite Menu

Now that we have the Menu item, we just need the composite class, which we're calling Menu. Remember, the composite class can hold Menu items or other Menus.

Here's a couple of methods from MenuComponent this class doesn't implement getPrice() and isVegetarian(), because those don't make a lot of sense for a Menu.

```
Menu is also a MenuComponent,
just like MenuItem.
↓
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
    }
}
```

Menu can have any number of children of type MenuComponent, we'll use an internal ArrayList to hold these.

This is different than our old implementation: we're going to give each Menu a name and a description. Before, we just relied on having different classes for each menu.

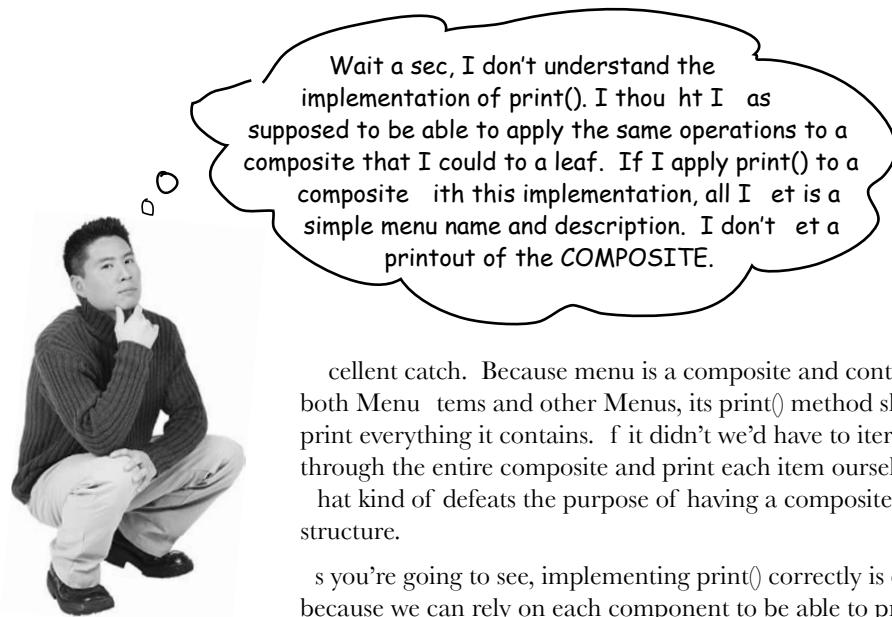
Here's how you add MenuItem or other Menus to a Menu. Because both MenuItem and Menu are MenuComponents, we just need one method to do both.

You can also remove a MenuComponent or get a MenuComponent.

Here are the getter methods for getting the name and description.

Notice, we aren't overriding getPrice() or isVegetarian() because those methods don't make sense for a Menu (although you could argue that isVegetarian() might make sense). If someone tries to call those methods on a Menu, they'll get an UnsupportedOperationException.

To print the Menu, we print the Menu's name and description.



cellent catch. Because menu is a composite and contains both Menu items and other Menus, its print() method should print everything it contains. If it didn't we'd have to iterate through the entire composite and print each item ourselves.

hat kind of defeats the purpose of having a composite structure.

s you're going to see, implementing print() correctly is easy because we can rely on each component to be able to print itself. t's all wonderfully recursive and groovy. Check it out

## Fixing the print method

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

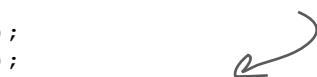
    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent) iterator.next();
            menuComponent.print();
        }
    }
}
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItem.



Look! We get to use an Iterator. We use it to iterate through all the Menu's components... those could be other Menus, or they could be MenuItem. Since both Menus and MenuItem implement print(), we just call print() and the rest is up to them.

**NOTE:** If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

# Getting ready for a test drive...

It's about time we took this code for a test drive, but we need to update the Waitress code before we do – after all she's the main client of this code.

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Yup! The Waitress code really is this simple. Now we just hand her the top level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy – all the menus, and all the menu items – is call print() on the top level menu.

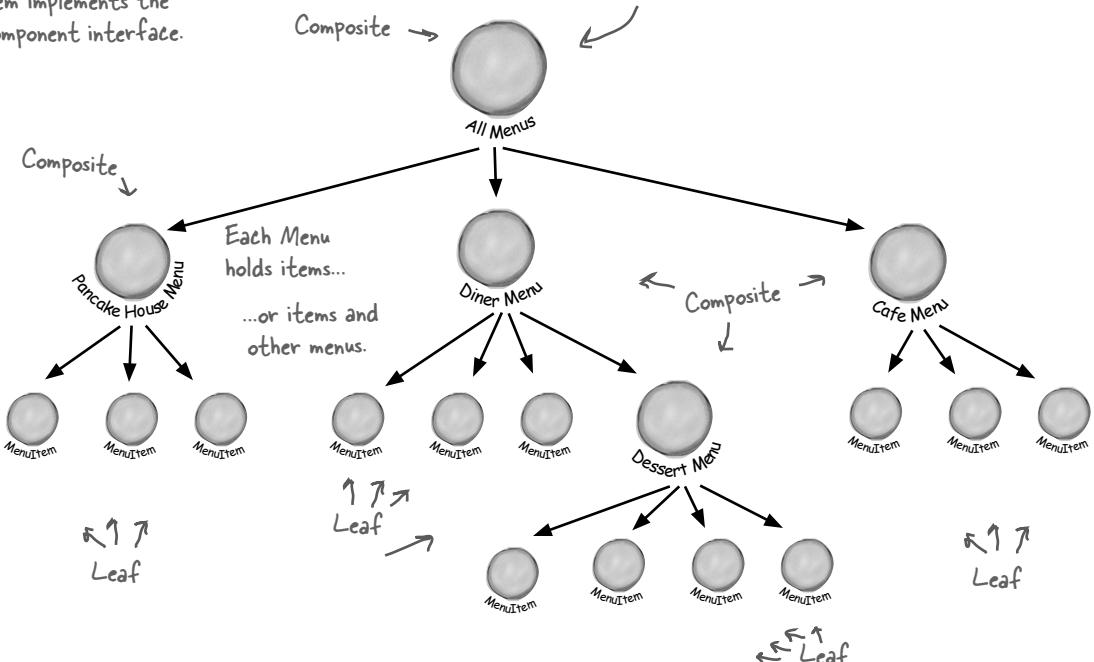
We're gonna have one happy Waitress.

Kay, one last thing before we write our test drive. Let's get an idea of what the menu composite is going to look like at runtime.

Every Menu and MenuItem implements the MenuComponent interface.

Composite →

The top level menu holds all menus and items.



## Now for the test drive...

Okay, now we just need a test drive. Unlike our previous version, we're going to handle all the menu creation in the test drive. We could ask each chef to give us his new menu, but let's get it all tested first. Here's the code

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));
        dinerMenu.add(dessertMenu);
        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flaky crust, topped with vanilla icecream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);
        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need two top level menu now that we'll name allMenus.

We're using the Composite add() method to add each menu to the top level menu, allMenus.

Now we need to add all the menu items, here's one example, for the rest, look at the complete source code.

And we're also adding a menu to a menu. All dinerMenu cares about is that everything it holds, whether it's a menu item or a menu, is a MenuComponent.

Add some apple pie to the dessert menu...

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's easy as apple pie for her to print it out.

# Getting ready for a test drive...

NOTE: this output is based on the complete source.

```

File Edit Window Help GreenE &Spam
% java MenuTestDrive
ALL MENUS, All menus combined
-----
PANCAKE HOUSE MENU, Breakfast
-----
    K&B's Pancake Breakfast(v), 2.99
        -- Pancakes with scrambled eggs, and toast
    Regular Pancake Breakfast, 2.99
        -- Pancakes with fried eggs, sausage
    Blueberry Pancakes(v), 3.49
        -- Pancakes made with fresh blueberries, and blueberry syrup
    Waffles(v), 3.59
        -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
-----
    Vegetarian BLT(v), 2.99
        -- (Fakin') Bacon with lettuce & tomato on whole wheat
    BLT, 2.99
        -- Bacon with lettuce & tomato on whole wheat
    Soup of the day, 3.29
        -- A bowl of the soup of the day, with a side of potato salad
    Hotdog, 3.05
        -- A hot dog, with saukraut, relish, onions, topped with cheese
    Steamed Veggies and Brown Rice(v), 3.99
        -- Steamed vegetables over brown rice
    Pasta(v), 3.89
        -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

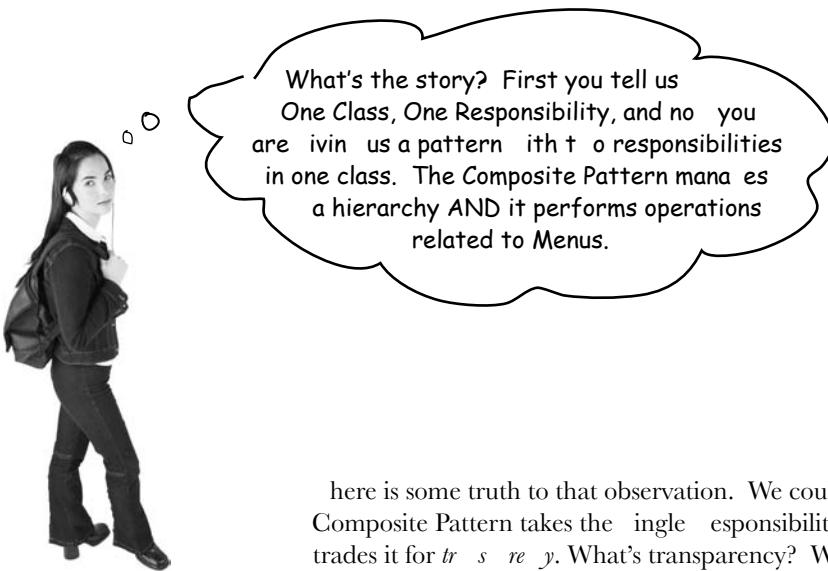
DESSERT MENU, Dessert of course!
-----
    Apple Pie(v), 1.59
        -- Apple pie with a flakey crust, topped with vanilla icecream
    Cheesecake(v), 1.99
        -- Creamy New York cheesecake, with a chocolate graham crust
    Sorbet(v), 1.89
        -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
-----
    Veggie Burger and Air Fries(v), 3.99
        -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
    Soup of the day, 3.69
        -- A cup of the soup of the day, with a side salad
    Burrito(v), 4.29
        -- A large burrito, with whole pinto beans, salsa, guacamole
%

```

Here's all our menus... we printed all this just by calling print() on the top level menu

The new dessert menu is printed when we are printing all the Diner menu components



What's the story? First you tell us One Class, One Responsibility, and now you are giving us a pattern with two responsibilities in one class. The Composite Pattern manages a hierarchy AND it performs operations related to Menus.

here is some truth to that observation. We could say that the Composite Pattern takes the single responsibility design principle and trades it for transparency. What's transparency? Well, by allowing the Component interface to contain the child management operations the leaf operations, a client can treat both composites and leaf nodes uniformly; so whether an element is a composite or leaf node becomes transparent to the client.

Now given we have both types of operations in the Component class, we lose a bit of safety because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate out the responsibilities into interfaces.

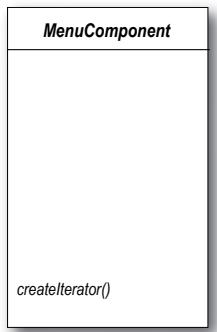
This would make our design safe, in the sense that any inappropriate calls on elements would be caught at compile time or runtime, but we'd lose transparency and our code would have to use conditionals and the `instanceof` operator.

So, to return to your question, this is a classic case of tradeoff. We are guided by design principles, but we always need to observe the effect they have on our designs. Sometimes we purposely do things in a way that seems to violate the principle. In some cases, however, this is a matter of perspective; for instance, it might seem incorrect to have child management operations in the leaf nodes (like `add()`, `remove()` and `getChild()`), but then again you can always shift your perspective and see a leaf as a node with zero children.

## las bac t te at

We promised you a few pages back that we'd show you how to use iterator with a Composite. You know that we are already using iterator in our internal implementation of the print() method, but we can also allow the Waitress to iterate over an entire composite if she needs to, for instance, if she wants to go through the entire menu and pull out vegetarian items.

To implement a Composite iterator, let's add a createIterator() method in every component. We'll start with the abstract MenuComponent class



We've added a `createIterator()` method to the `MenuComponent`. This means that each `Menu` and `MenuItem` will need to implement this method. It also means that calling `createIterator()` on a composite should apply to all children of the composite.

Now we need to implement this method in the `Menu` and `MenuItem` classes

```
public class Menu extends MenuComponent {
    Iterator iterator = null;           ← We only need one
    // other code here doesn't change   iterator per Menu.

    public Iterator createIterator() {
        if (iterator == null) {
            iterator = new CompositeIterator(menuComponents.iterator());
        }
        return iterator;
    }
}

public class MenuItem extends MenuComponent {
    // other code here doesn't change

    public Iterator createIterator() {
        return new NullIterator();          ← Whoa! What's this NullIterator?
    }                                     You'll see in two pages.
}
```

Here we're using a new iterator called `CompositeIterator`. It knows how to iterate over any composite.

← We pass it the current composite's iterator.

Now for the `MenuItem`...

← Whoa! What's this `NullIterator`?  
You'll see in two pages.

# The Composite Iterator

The Composite iterator is a iterator. It's got the job of iterating over the Menu items in the component, and of making sure all the child Menus (and child child Menus, and so on) are included.

Here's the code. Watch out, this isn't a lot of code, but it can be a little mind bending. Just repeat to yourself as you go through it recursion is my friend, recursion is my friend.

```
import java.util.*;
```

```
public class CompositeIterator implements Iterator {
    Stack stack = new Stack();
```

Like all iterators, we're implementing the java.util.Iterator interface.

```
    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }
```

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

```
    public Object next() {
```

```
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }
```

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

```
    public boolean hasNext() {
```

```
        if (stack.empty()) {
            return false;
        } else {
```

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack. In either case, we return the component.

```
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
```

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

```
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }
```

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

```
    public void remove() {
```

```
        throw new UnsupportedOperationException();
    }
```

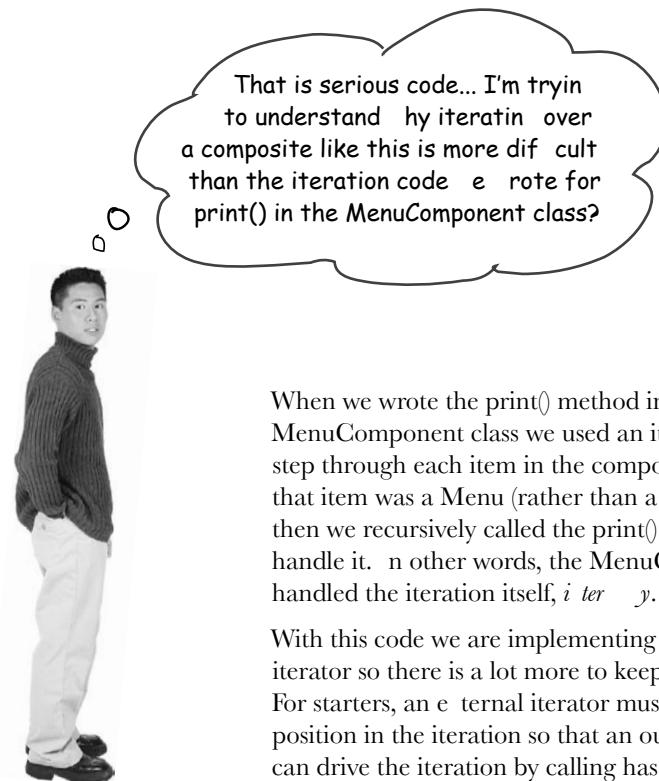
We're not supporting remove, just traversal.

```
}
```



**WATCH OUT:  
RECUSION  
ZONE AHEAD**

Otherwise there is a next element and we return true.



When we wrote the print() method in the MenuComponent class we used an iterator to step through each item in the component and if that item was a Menu (rather than a Menu item), then we recursively called the print() method to handle it. In other words, the MenuComponent handled the iteration itself, *i ter ately*.

With this code we are implementing an *external* iterator so there is a lot more to keep track of. For starters, an external iterator must maintain its position in the iteration so that an outside client can drive the iteration by calling hasNext() and next(). But in this case, our code also needs to maintain that position over a composite, recursive structure. That's why we use stacks to maintain our position as we move up and down the composite hierarchy.



Draw a diagram of the Menu and MenuItem. Then pretend you are the CompositeIterator, and you would make calls to a Next() and next(). Trace the way the CompositeIterator traverses the structure in the code indicated:

```
public void testCompositeIterator(MenuComponent component) {  
    CompositeIterator iterator = new CompositeIterator(component.iterator);  
  
    while(iterator.hasNext()) {  
        MenuComponent component = iterator.next();  
    }  
}
```

# The Null Iterator

Okay, now what is this null iterator all about? Think about it this way: a Menu item has nothing to iterate over, right? So how do we handle the implementation of its `createIterator()` method? Well, we have two choices

NOTE: Another example of the Null Object "Design Pattern."

## Choice one

### Return null

We could return null from `createIterator()`, but then we'd need conditional code in the client to see if null was returned or not.

## Choice two

### Return an iterator that always returns false when `hasNext()` is called

This seems like a better plan. We can still return an iterator, but the client doesn't have to worry about whether or not null is ever returned. In effect, we're creating an iterator that is a no op.

The second choice certainly seems better. Let's call it null iterator and implement it.

```
import java.util.Iterator;

public class NullIterator implements Iterator {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

This is the laziest iterator you've ever seen, at every step of the way it punts.

When `next()` is called, we return null.

Most importantly when `hasNext()` is called we always return false.

And the NullIterator wouldn't think of supporting remove.

# Give me the vegetarian menu

Now we've got a way to iterate over every item of the Menu. Let's take that and give our Waitress a method that can tell us exactly which items are vegetarian.

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator iterator = allMenus.createIterator();
        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent) iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

The `printVegetarianMenu()` method takes the `allMenus`'s `composite` and gets its `iterator`. That will be our `CompositeIterator`.

Iterate through every element of the composite.

Call each element's `isVegetarian()` method and if true, we call its `print()` method.

`print()` is only called on `MenuItem`s, never `composites`. Can you see why?

We implemented `isVegetarian()` on the `Menus` to always throw an exception. If that happens we catch the exception, but continue with our iteration.

# The magic of Iterator Composite together...

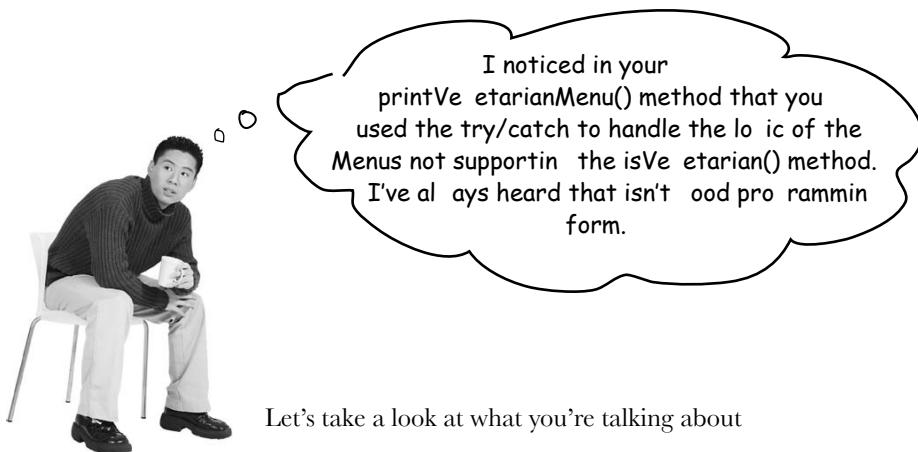
Whoooo it's been quite a development effort to get our code to this point. Now we've got a general menu structure that should last the growing inner empire for some time. Now it's time to sit back and order up some veggie food

```

File Edit Window Help Ha eU ed rIteratorToday?
% java MenuTestDrive
VEGETARIAN MENU
-----
K&B's Pancake Breakfast(v) , 2.99
-- Pancakes with scrambled eggs, and toast
Blueberry Pancakes(v) , 3.49
-- Pancakes made with fresh blueberries, and blueberry syrup
Waffles(v) , 3.59
-- Waffles, with your choice of blueberries or strawberries
Vegetarian BLT(v) , 2.99
-- (Fakin') Bacon with lettuce & tomato on whole wheat
Steamed Veggies and Brown Rice(v) , 3.99
-- Steamed vegetables over brown rice
Pasta(v) , 3.89
-- Spaghetti with Marinara Sauce, and a slice of sourdough bread
Apple Pie(v) , 1.59
-- Apple pie with a flakey crust, topped with vanilla icecream
Cheesecake(v) , 1.99
-- Creamy New York cheesecake, with a chocolate graham crust
Sorbet(v) , 1.89
-- A scoop of raspberry and a scoop of lime
Apple Pie(v) , 1.59
-- Apple pie with a flakey crust, topped with vanilla icecream
Cheesecake(v) , 1.99
-- Creamy New York cheesecake, with a chocolate graham crust
Sorbet(v) , 1.89
-- A scoop of raspberry and a scoop of lime
Veggie Burger and Air Fries(v) , 3.99
-- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
Burrito(v) , 4.29
-- A large burrito, with whole pinto beans, salsa, guacamole
%

```

The Vegetarian Menu consists of the vegetarian items from every menu.



Let's take a look at what you're talking about

```
try {
    if (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} catch (UnsupportedOperationException) {}
```

We call isVegetarian() on all MenuComponents, but Menus throw an exception because they don't support the operation.

If the menu component doesn't support the operation, we just throw away the exception and ignore it.

In general we agree; try catch is meant for error handling, not program logic. What are our other options? We could have checked the runtime type of the menu component with instanceof to make sure it's a Menu item before making the call to isVegetarian(). But in the process we'd lose transparency because we wouldn't be treating Menus and Menu items uniformly.

We could also change isVegetarian() in the Menus so that it returns false. This provides a simple solution and we keep our transparency.

In our solution we are going for clarity we really want to communicate that this is an unsupported operation on the Menu (which is different than saying isVegetarian() is false). It also allows for someone to come along and actually implement a reasonable isVegetarian() method for Menu and have it work with the existing code.

That's our story and we're stickin' to it.



## Patterns Exposed

We're here tonight speaking with the Composite Pattern. Why don't you tell us a little about yourself, Composite?

ure... I'm the pattern to use when you have collections of objects with whole part relationships and you want to be able to treat those objects uniformly.

Okay, let's dive right in here... what do you mean by whole part relationships?

Imagine a graphical user interface; there you'll often find a top level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the components that contain other components, composite objects, and components that don't contain other components, leaf objects.

So that's what you mean by treating the objects uniformly? Having common methods you can call on composites and leaves?

Right. I can tell a composite object to display or a leaf object to display and they will do the right thing. The composite object will display by telling all its components to display.

That implies that every object has the same interface. What if you have objects in your composite that do different things?

Well, in order for the composite to work transparently to the client, you must implement the same interface for all objects in the composite, otherwise, the client has to worry about which interface each object is implementing, which kind of defeats the purpose.

Obviously that means that at times you'll have objects for which some of the method calls don't make sense.

So how do you handle that?

Well there's a couple of ways to handle it; sometimes you can just do nothing, or return null or false whatever makes sense in your application. Other times you'll want to be more proactive and throw an exception. Of course, then the client has to be willing to do a little work and make sure that the method call didn't do something unexpected.

But if the client doesn't know which kind of object they're dealing with, how would they ever know which calls to make without checking the type?

If you're a little creative you can structure your methods so that the default implementations do something that does make sense. For instance, if the client is calling getChild(), on the composite this makes sense. And it makes sense on a leaf too, if you think of the leaf as an object with no children.

It's smart. But, I've heard some clients are so worried about this issue, that they require separate interfaces for different objects so they aren't allowed to make nonsensical method calls. So that's still the Composite Pattern?

Yes. It's a much safer version of the Composite Pattern, but it requires the client to check the type of every object before making a call so the object can be cast correctly.

Tell us a little more about how these composite and leaf objects are structured.

Usually it's a tree structure, some kind of hierarchy. The root is the top level composite, and all its children are either composites or leaf nodes.

So children ever point back up to their parents?

No, a component can have a pointer to a parent to make traversal of the structure easier. And, if you have a reference to a child, and you need to delete it, you'll need to get the parent to remove the child. Having the parent reference makes that easier too.

here's really quite a lot to consider in your implementation. Are there other issues we should think about when implementing the Composite Pattern?

Actually there are... one is the ordering of children. What if you have a composite that needs to keep its children in a particular order? Then you'll need a more sophisticated management scheme for adding and removing children, and you'll have to be careful about how you traverse the hierarchy.

Good point I hadn't thought of.

And did you think about caching?

Caching?

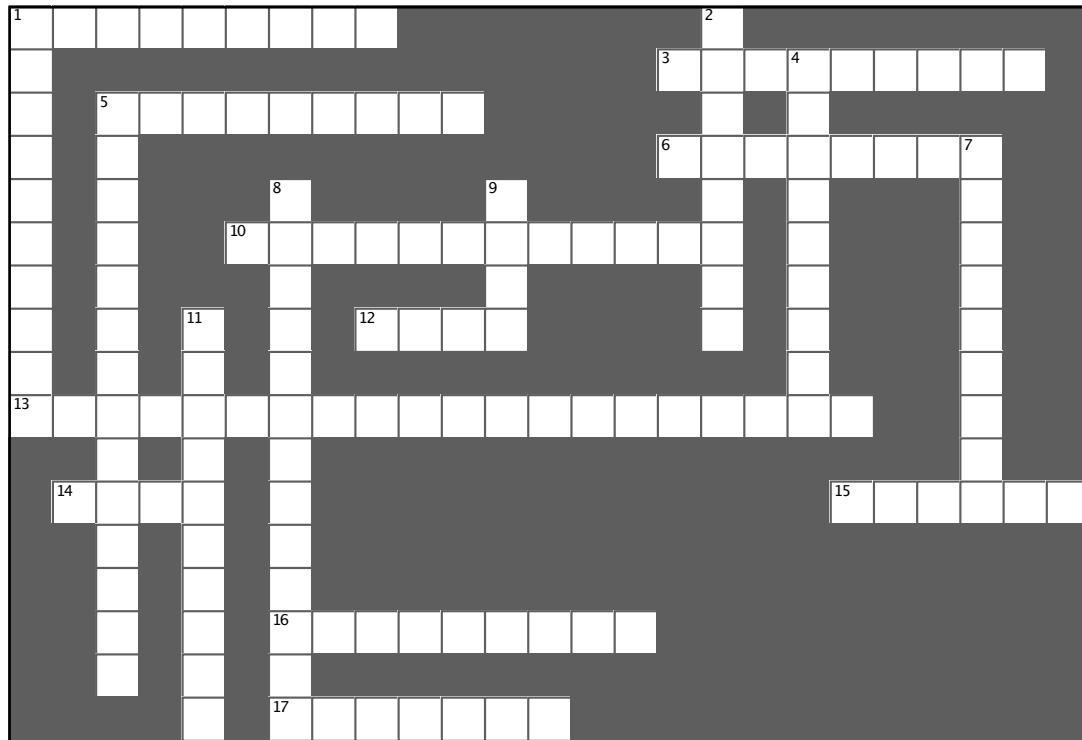
Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores the result temporarily to save traversals.

Well, there's a lot more to the Composite Patterns than ever would have guessed. Before we wrap this up, one more question: What do you consider your greatest strength?

I think I'd definitely have to say simplifying life for my clients. My clients don't have to worry about whether they're dealing with a composite object or a leaf object, so they don't have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

That does sound like an important benefit.

There's no doubt you're a useful pattern to have around for collecting and managing objects. And, with that, we're out of time... Thanks so much for joining us and come back soon for another Patterns Uncovered.



## #'\$

" 5J<I @<I>8: <G8: B8><J F@E LJ< K@G8K@IE  
 =FI K@: FD GFE<EKI  
 \$ , FC: K@E 8E; /K18KFI 8I< @K@G8: B8><  
 & 6 <E: 8GJL@K; K@  
 ' + J<G8I8K F9A: K@8K 8E K8M@IJ< 8  
 : FC: K@E  
 " ! 1 <I><; N@ K< - @<  
 "# . 8J EF : ?@I<E  
 "\$ 28D< F=G@ @G K8KJ@KJ FEC@FE<  
 I<JGFEJ@ @G<I : GJJ  
 "% 4?@ : FD G8EO8: HL@;  
 "& + : GJJ J?FLC ?8M@ FEC@FE< I<JFE K@ ; F  
 K@  
 "' 4?@: GJJ @ @: K@JLGGFIK /K18KFI  
 "( 4?@D<EL : 8LJ<; LJ K@ : ?8E>< FLI <E@  
 @G3D<E@K@E

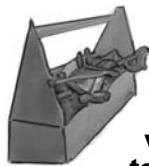
## " %!

" + : FD G@J@ ?FCJ K@  
 # 6 < A@B <E89G; ?<  
 % 6 < ; <GK; 38E: 8B<. FLJ<1 <EL/K18KFI  
 9<: 8LJ< K@: GJJ 8C@8; OGF@J 8E @K18KFI  
 & 4?< /K18KFI 38K@IE ; <: FLG@J K< : @EKF@FD  
 K< 8>>I<>8KJ 777777777  
 ( , FD G@J@ /K18KFI LJ<; 8 GKF=K@  
 ) /K18KFIJ 8I< L@L8@: I<8K; LJ@ K@  
 G8K@IE  
 \* + : FD GFE<EK: 8E 9< 8 : FD G@J@ FI K@  
 " " . 8J?K9G 8E; +I@8@K9FK @G3D<EK@  
 @<I>8: <



Match each pattern with its description

<b>Pattern</b>	<b>Description</b>
trate	ients treat o e tions o ob e ts an in i i ua ob e ts uni orm
A a ter	ro i es a a to tra erse a o e tion o ob e ts ithout e osin the o e tions im ementation
terator	im i es the inter a e o a rou o asses
a a e	han es the inter a e o one or more asses
om osite	A o s a rou o ob e ts to be noti e hen some state han es
bser er	n a suates inter han eab e behaviors an uses e e ation to e i e hi h one to uses



## Tools for your Design Toolbox

two new patterns for your tool box    two great ways  
to deal with collections of objects



### OO Principles

Encapsulate what varies

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you.

A class should have only one reason to change.

### Basics

abstraction

encapsulation

polymorphism

inheritance

### OO Patterns

Structural Patterns:  
Adapter, Composite, Decorator, Facade, Flyweight, Singleton, State, Strategy, Visitor.

**Iterator** - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

structure

Define the in an operation.

**Composite** - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Another two-for-one Chapter.

Yet another important principle based on change in a design.



## er ise solutions



### Sharpen your pencil

Based on our implementation of printMenu(), which of the following apply?

- A. We are coding to the PancakeHouseMenu and innerMenu concrete implementations, not to an interface.
- B. The Waitress doesn't implement the ava Waitress P and so isn't adhering to a standard.
- C. If we decided to switch from using innerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.
- D. The Waitress needs to know how each menu represents its internal collection of menu items is implemented, this violates encapsulation.
- E. We have duplicate code: the printMenu() method needs two separate loop implementations to iterate over the two different kinds of menus. And if we added a third menu, we might have to add yet another loop.
- F. The implementation isn't based on M ML (Menu ML) and so isn't as interoperable as it should be.



### Sharpen your pencil

Before termin t e pa e, ic ly ot down t et reet in we a e  
to do to t i code to t it into o r ramewor :

1. implement the Menu interface

  . get rid of getItems()

  . add createIterator() and return an Iterator that can step through the Hashtable values



## Code Magnets Solution

The unscrambled Alternating DinerMenu Iterator

```
import java.util.Iterator;
import java.util.Calendar;

public class AlternatingDinerMenuItemator implements Iterator {
    MenuItem[] items;
    int position;

    public AlternatingDinerMenuItemator(MenuItem[] items) {
        this.items = items;
        Calendar rightNow = Calendar.getInstance();
        position = rightNow.get(Calendar.DAY_OF_WEEK) % 2;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 2;
        return menuItem;
    }

    public void remove() {
        throw new UnsupportedOperationException(
            "Alternating Diner Menu Iterator does not support remove()");
    }
}
```

Notice that this Iterator implementation does not support remove()

# \* WHO DOES ? WHAT ? \*

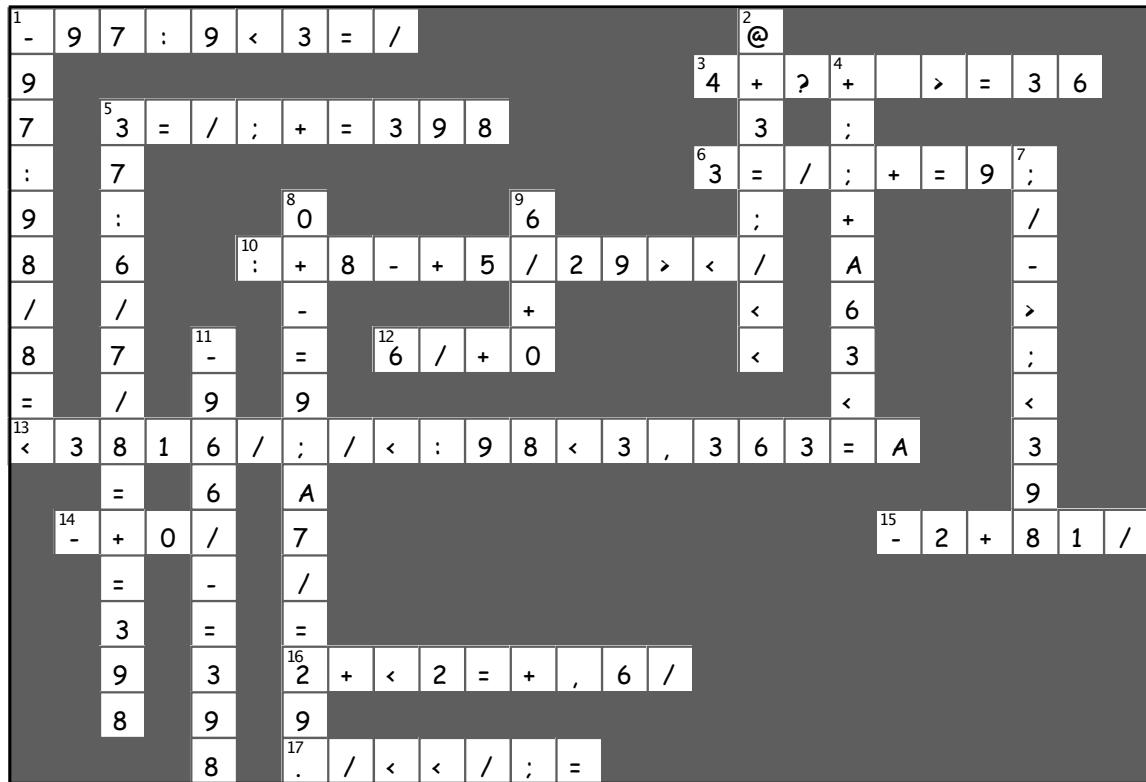
Match each pattern with its description

<b>Pattern</b>	<b>Description</b>
trate	ients treat o e tions o ob e ts an in i i ua ob e ts uni orm
A a ter	ro i es a a to tra erse a o e tion o ob e ts i thout e osin the o e tions im ementation
terator	im i es the inter a e o a rou o asses
a a e	han es the inter a e o one or more asses
om osite	A o s a rou o ob e ts to be noti e hen some state han es
bser er	n a su ates inter han eab e beh a iors an uses ee ation to e i e hi h one to uses

trate	ients treat o e tions o ob e ts an in i i ua ob e ts uni orm
A a ter	ro i es a a to tra erse a o e tion o ob e ts i thout e osin the o e tions im ementation
terator	im i es the inter a e o a rou o asses
a a e	han es the inter a e o one or more asses
om osite	A o s a rou o ob e ts to be noti e hen some state han es
bser er	n a su ates inter han eab e beh a iors an uses ee ation to e i e hi h one to uses



# er ise solutions



the state pattern

# \* he state of things \*



I thou ht thin s in Objectville  
ere oin to be so easy, but no  
every time I turn around there's  
another chan e re uest comin in.  
I'm to the breakin point! Oh, maybe  
I should have been oin to Betty's  
Wednesday ni ht patterns roup all  
alon . I'm in such a state!

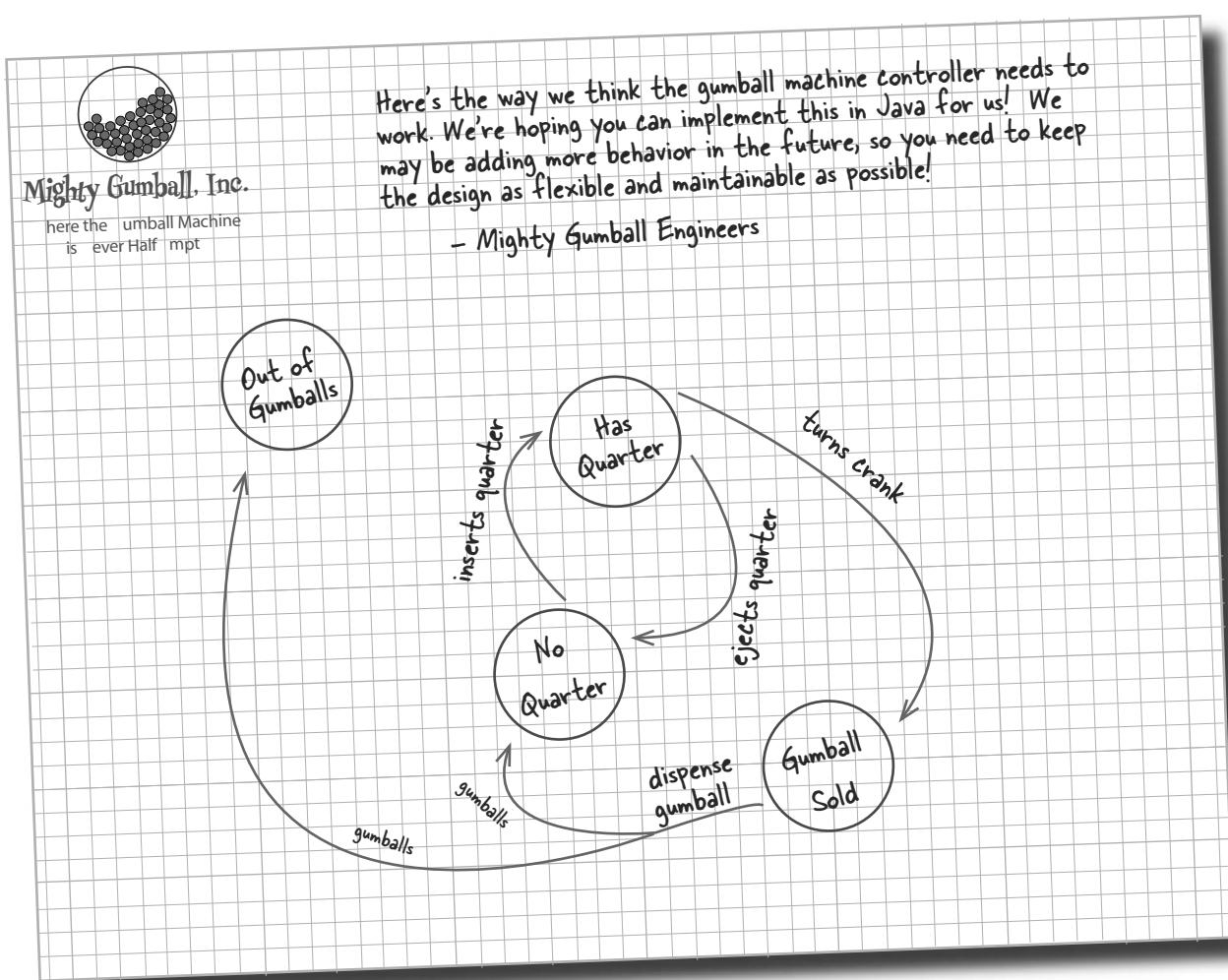
A yo now, t e Strate y Pattern went on to create a wildly  
cce I ine aro nd interc an ea leal orit m . State, owe er, too t e per ap  
more no le pat o elpin o ect to control t eir e a ior y c an in t eir internal  
tate. He o ten o er eard tellin i o ect client , J t repeat a ter me: I m ood  
eno , I m mart eno , and do onit...

# ava Breakers

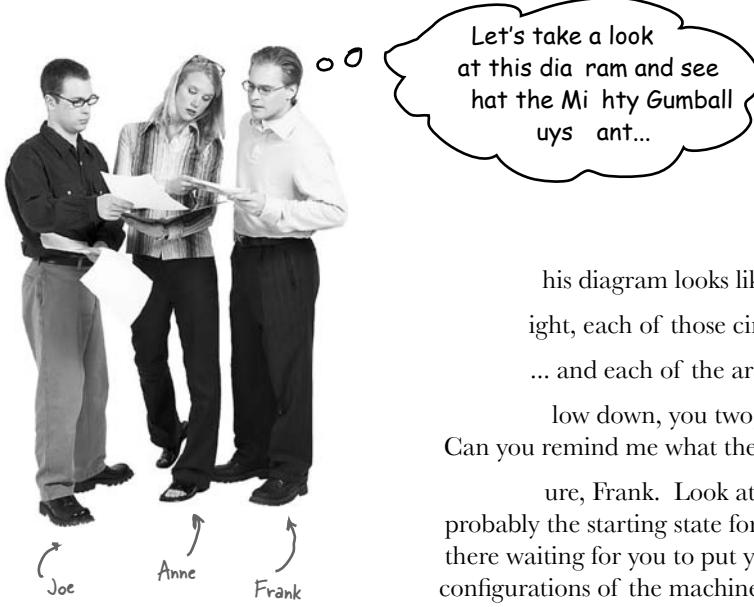
ava toasters are so s. oday people are building ava into re devices, like gumball machines. hat's right, gumball machines have gone high tech; the major manufacturers have found that by putting CP s into their machines, they can increase sales, monitor inventory over the network and measure customer satisfaction more accurately.

But these manufacturers are gumball machine e perts, not software developers, and they've asked for your help

At least that's their story - we think they just got bored with the circa 18 's technology and needed to find a way to make their jobs more exciting.



## Cubicle Conversation



his diagram looks like a state diagram.

ight, each of those circles is a state...

... and each of the arrows is a state transition.

low down, you two, it's been too long since I studied state diagrams. Can you remind me what they're all about?

ure, Frank. Look at the circles; those are states. No quarter is probably the starting state for the gumball machine because it's just sitting there waiting for you to put your quarter in. All states are just different configurations of the machine that behave in a certain way and need some action to take them to another state.

ight. See, to go to another state, you need to do something like put a quarter in the machine. See the arrow from No quarter to Has quarter?

es...

hat just means that if the gumball machine is in the No quarter state and you put a quarter in, it will change to the Has quarter state. That's the state transition.

h, see and if I'm in the Has quarter state, can turn the crank and change to the Gumball old state, or eject the quarter and change back to the No quarter state.

You got it

This doesn't look too bad then. We've obviously got four states, and I think we also have four actions: inserts quarter, ejects quarter, turns crank and dispense. But... when we dispense, we test for zero or more gumballs in the Gumball old state, and then either go to the Out of Gumballs state or the No quarter state. So we actually have five transitions from one state to another.

hat test for zero or more gumballs also implies we've got to keep track of the number of gumballs too. Any time the machine gives you a gumball, it might be the last one, and if it is, we need to transition to the Out of Gumballs state.

lso, don't forget that you could do nonsensical things, like try to eject the quarter when the gumball machine is in the No quarter state, or insert two quarters.

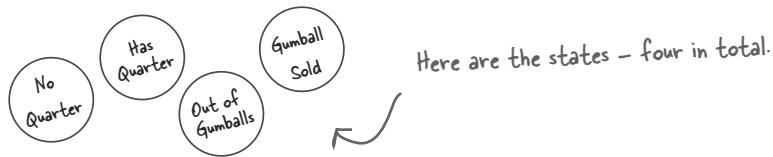
h, didn't think of that; we'll have to take care of those too.

For every possible action we'll just have to check to see which state we're in and act appropriately. We can do this. Let's start mapping the state diagram to code...

## State machines

How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines

- 1 First, gather up your states



- 2 Next, create an instance variable to hold the current state, and define values for each of the states

Let's just call "Out of Gumballs"  
"Sold Out" for short.

```
final static int SOLD_OUT = 0;  
final static int NO_QUARTER = 1;  
final static int HAS_QUARTER = 2;  
final static int SOLD = 3;
```

```
int state = SOLD_OUT;
```

Here's each state represented  
as a unique integer...

...and here's an instance variable that holds the  
current state. We'll go ahead and set it to  
"Sold Out" since the machine will be unfilled when  
it's first taken out of its box and turned on.

- 3 Now we gather up all the actions that can happen in the system

inserts quarter      turns crank  
ejects quarter

dispense

These actions are  
the gumball machine's  
interface - the things  
you can do with it.

Dispense is more of an internal  
action the machine invokes on itself.

Looking at the diagram, invoking any of these  
actions causes a state transition.

Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this

```
public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Here we're talkin' about a common technique: modeling state within an object by creating an instance variable to hold the state values and putting conditional code within our methods to handle the various states.



With that quick review, let's go implement the Gumball Machine!

## Writing the code

It's time to implement the Gumball Machine. We know we're going to have an instance variable that holds the current state. From there, we just need to handle all the actions, behaviors and state transitions that can happen. For actions, we need to implement inserting a quarter, removing a quarter, turning the crank and dispensing a gumball; we also have the empty gumball condition to implement as well.

```
public class GumballMachine {
    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
}
```

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD OUT state.

```
public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    }
}

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.
```

Now we start implementing the actions as methods....

When a quarter is inserted, if....

a quarter is already inserted we tell the customer;

otherwise we accept the quarter and transition to the HAS QUARTER state.

and if the machine is sold out, we reject the quarter.

```

public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}

public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}

public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}

// other methods here like toString() and refill()
}

```

Now, if the customer tries to remove the quarter... If there is a quarter, we return it and go back to the NO QUARTER state.

Otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

The customer tries to turn the crank... If the customer just turned the crank, we can't give a refund; he already has the gumball!

Someone's trying to cheat the machine. We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method. We're in the SOLD state; give em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give em an error, not a gumball.

# In-house testing

hat feels like a nice solid design using a well thought out methodology doesn't it? Let's do a little in house testing before we hand it off to Mighty Gumball to be loaded into their actual gumball machines. Here's our test harness

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Load it up with five gumballs total.

Print out the state of the machine.

Throw a quarter in...

Turn the crank; we should get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Ask for it back.

Turn the crank; we shouldn't get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Turn the crank; we should get our gumball.

Throw a quarter in...

Turn the crank; we should get our gumball.

Ask for a quarter back we didn't put in.

Print out the state of the machine, again.

Throw TWO quarters in...

Turn the crank; we should get our gumball.

Now for the stress testing... 😊

Print that machine state one more time.

```
File Edit Window Help mi_ty_m all.com
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

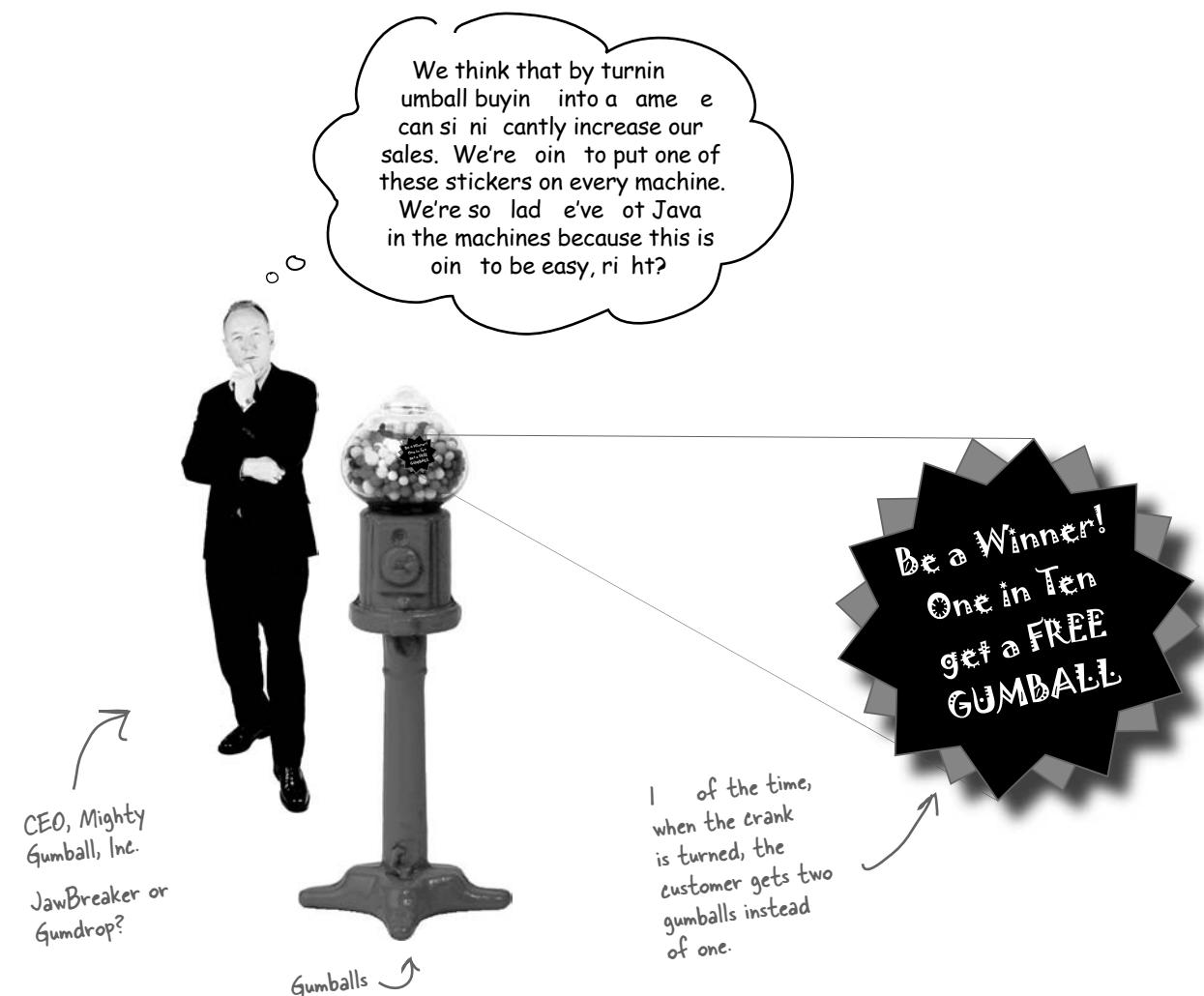
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

u a uyin a e

## You knew it was coming... a change request

t ll n lo e or o e n t o e r n e  
et ne n t e r It r n e e ert re tt n  
tt ro t e o re e r t n loo n re t ro  
ter er e t e

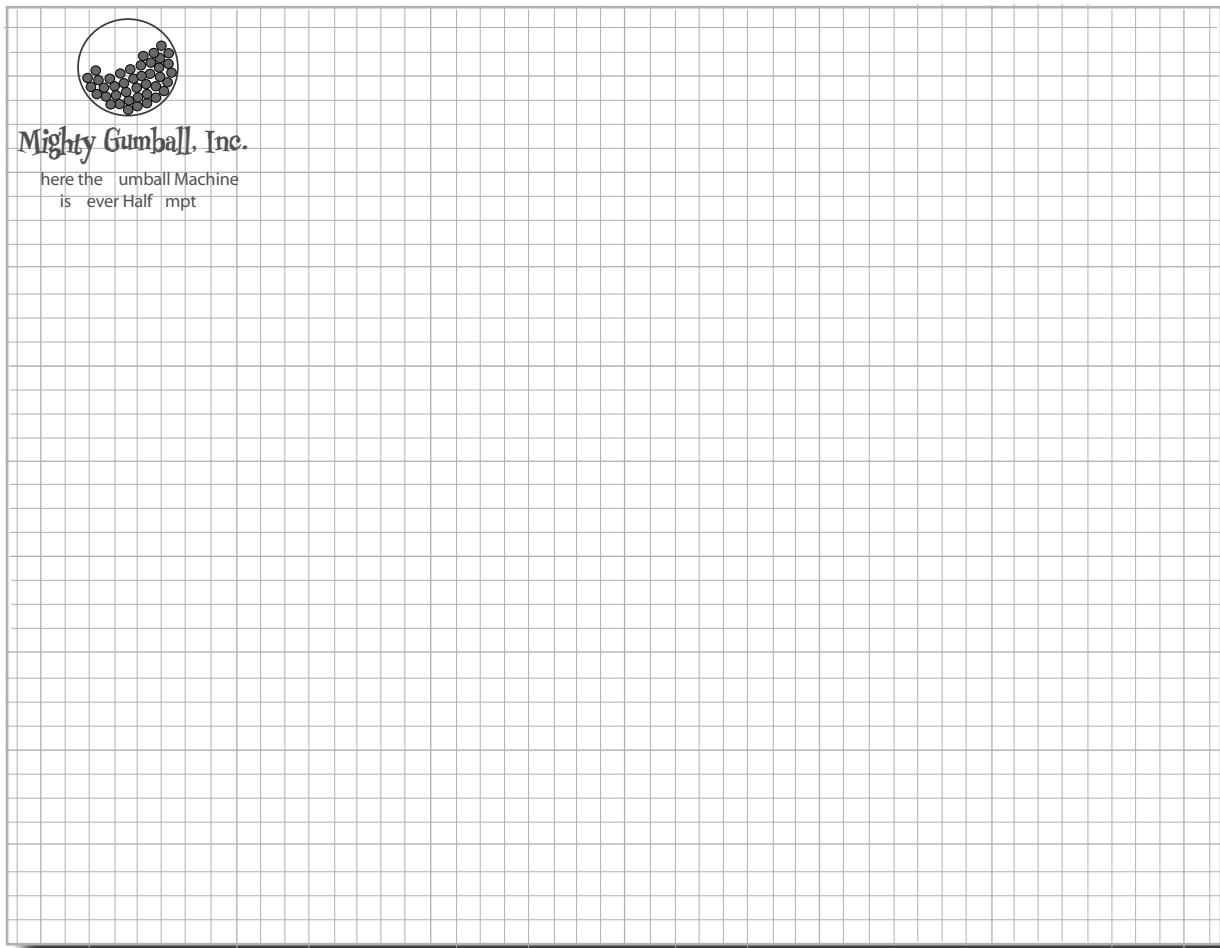
n tt n e one o oot l t e l e t o t e  
t n tot e n t e l e l





## esign Pu le

Draw a state diagram or a Gumball Machine state transition diagram in 10 minutes. In this contest, 10% of the time is sold to two all-new released, not one. Create your own diagram (at the end of each page) to make sure we are more you off...er...



Use Mighty Gumball's stationary to draw your state diagram.



## The messy STATE of things...

ust because you've written your gumball machine using a well thought out methodology doesn't mean it's going to be easy to extend. In fact, when you go back and look at your code and think about what you'll have to do to modify it, well...

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;
```

```
public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

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

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

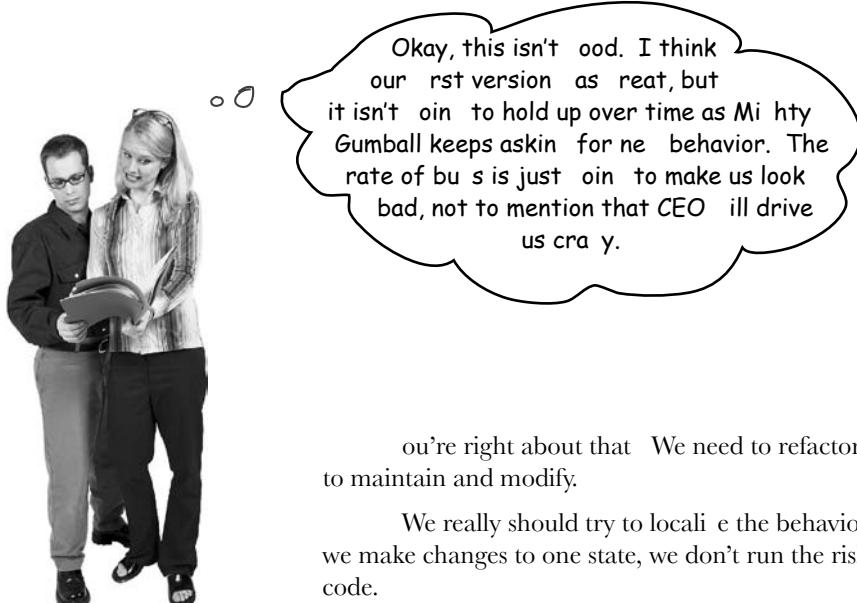
turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.



### Sharpen your pencil

Which of the following describe the state of our implementation?  
(Choose all that apply.)

- A. his code certainly isn't adhering to the Open/Closed Principle.
- B. his code would make a first-class programmer proud.
- C. his design isn't even very object oriented.
- D. state transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
- E. We haven't encapsulated anything that varies here.
- F. Further additions are likely to cause bugs in working code.



You're right about that. We need to refactor this code so that it's easy to maintain and modify.

We really should try to localize the behavior for each state so that if we make changes to one state, we don't run the risk of messing up the other code.

Right; in other words, follow that old encapsulate what varies principle.

Actually,

If we put each state's behavior in its own class, then every state just implements its own actions.

Right. And maybe the Gumball Machine can just delegate to the state object that represents the current state.

Yeah, you're good favor composition... more principles at work.

Cute. Well, I'm not sure how this is going to work, but I think we're onto something.

I wonder if this will make it easier to add new states?

I think so... We'll still have to change code, but the changes will be much more limited in scope because adding a new state will mean we just have to add a new class and maybe change a few transitions here and there.

I like the sound of that. Let's start hashing out this new design

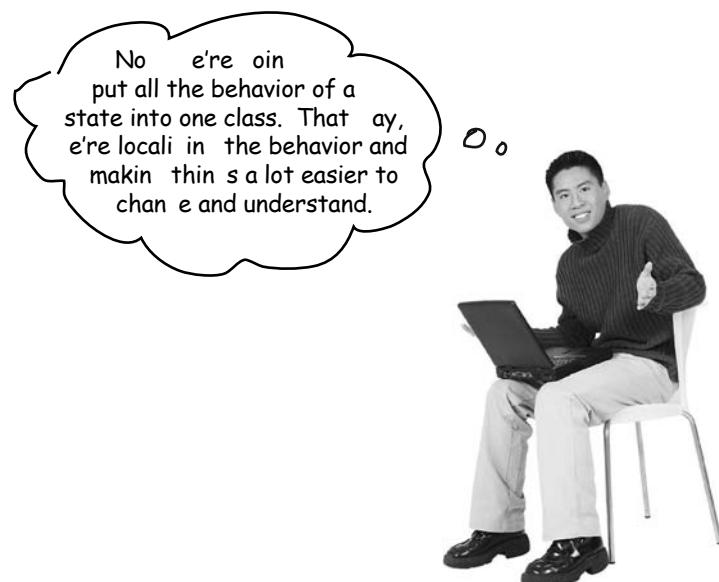
## The new design

It looks like we've got a new plan instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it

- ① First, we're going to define a state interface that contains a method for every action in the machine**
- ② Then we're going to implement a state class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state**
- ③ Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us**

Not only are we following design principles, as you'll see, we're actually implementing the State Pattern. But we'll get to all the official State Pattern stuff after we rework our code...

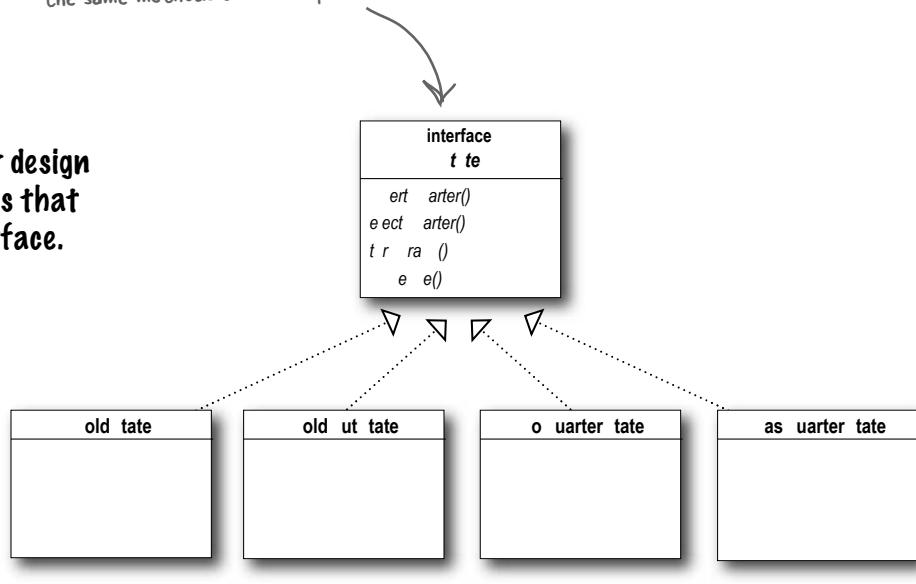


# Defining the State interfaces and classes

First let's create an interface for State, which all our states implement:

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).

Then take each state in our design and encapsulate it in a class that implements the State interface.



```

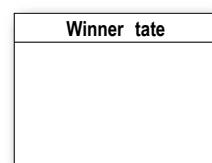
public class GumballMachine {
    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
}

```

... and we map each state directly to a class.

Don't forget, we need a new "winner" state too that implements the state interface. We'll come back to this after we reimplement the first version of the Gumball Machine.



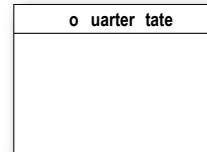
hat are a the ate

## Sharpen your pencil

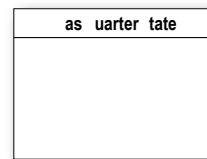
To implement our state, we first need to specify the interface class when each action is called. Annotate the diagram below with the interface for each action in each class; we've already filled in a few for you.

Go to HasQuarterState

Tell the customer, "You haven't inserted a quarter."

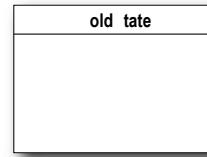


Go to SoldState

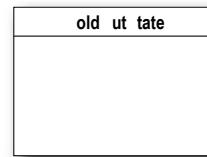


Tell the customer, "Please wait, we're already giving you a gumball."

Dispense one gumball. Check number of gumballs; if , go to NoQuarterState, otherwise, go to SoldOutState



Tell the customer, "There are no gumballs."



Go ahead and fill this out even though we're implementing it later.

ha ter

# Implementing our State classes

Time to implement a state we know what behaviors we want; we just need to get it down in code. We're going to closely follow the state machine code we wrote, but this time everything is broken out into different classes.

Let's start with the no quarter state

```
First we need to implement the State interface.
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
    }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.



What we're doing is implementing the behaviors that are appropriate for the state we're in. In some cases, this behavior includes moving the Gumball Machine to a new state.

*you are here ▶*

## Reworking the Gumball Machine

Before we finish the State classes, we're going to rework the Gumball Machine that way you can see how it all fits together. We'll start with the state-related instance variables and switch the code from using integers to using state objects:

```
public class GumballMachine {  
  
    final static int SOLD_OUT = 0;  
    final static int NO_QUARTER = 1;  
    final static int HAS_QUARTER = 2;  
    final static int SOLD = 3;  
  
    int state = SOLD_OUT;  
    int count = 0;
```

Old code

In the GumballMachine, we update the code to use the new classes rather than the static integers. The code is quite similar, except that in one class we have integers and in the other objects...

```
public class GumballMachine {  
  
    State soldOutState;  
    State noQuarterState;  
    State hasQuarterState;  
    State soldState;  
  
    State state = soldOutState;  
    int count = 0;
```

New code

All the State objects are created and assigned in the constructor.

This now holds a State object, not an integer.

## Now, let's look at the complete GumballMachine class...

```

public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // More methods here including getters for each State...
}

```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable. It also creates the State instances, one of each.

If there are more than gumballs we set the state to the NoQuarterState.

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

This includes methods like getNoQuarterState() for getting each state object, and getCount() for getting the gumball count.

## Implementing more states

Now that you're starting to get a feel for how the Gumball Machine and the states fit together, let's implement the HasQuarterState and the SoldState classes...

```
public class HasQuarterState implements State {  
    GumballMachine gumballMachine;  
  
    public HasQuarterState(GumballMachine gumballMachine) {  
        this.gumballMachine = gumballMachine;  
    }  
  
    public void insertQuarter() {  
        System.out.println("You can't insert another quarter");  
    }  
  
    public void ejectQuarter() {  
        System.out.println("Quarter returned");  
        gumballMachine.setState(gumballMachine.getNoQuarterState());  
    }  
  
    public void turnCrank() {  
        System.out.println("You turned...");  
        gumballMachine.setState(gumballMachine.getSoldState());  
    }  
    public void dispense() {  
        System.out.println("No gumball dispensed");  
    }  
}
```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.

Now, let's check out the old state class...

```
public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.



Look at the GumballMachine implementation. If the crank isn't turned and not accepted (say the customer didn't insert a quarter), we call dispense anyway, even though it's unnecessary. How many times?

*you are here ▶*

## Sharpen your pencil

We are one remainin cla we a ent implemented: SoldOutState.  
Why don't yo implement it? To do t i , care lly t in t ro o  
t e G m all Mac ine o ld e a e in eac it ation. C ec yo r  
an wer e ore mo in on...

```
public class SoldOutState implements GumballMachine {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        }

        public void insertQuarter() {
            }

            public void ejectQuarter() {
                }

                public void turnCrank() {
                    }

                    public void dispense() {
                        }

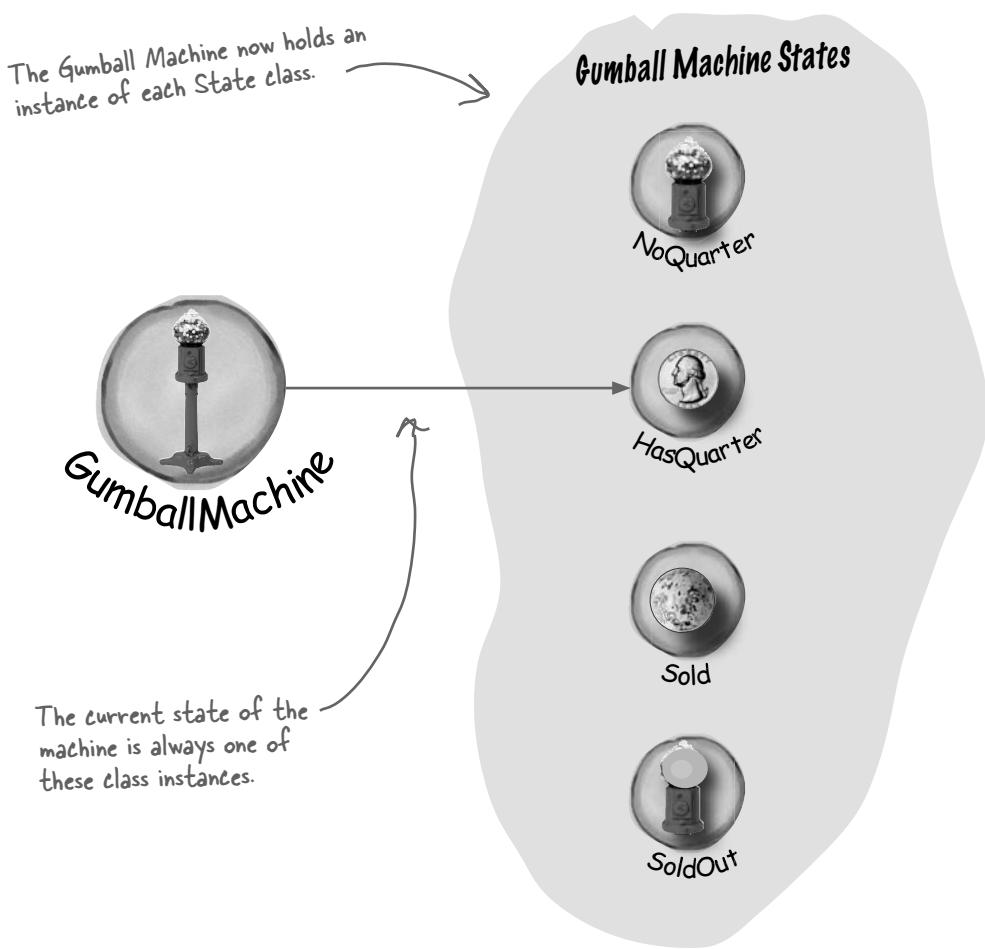
                        }
```

## let's take a look at what we've done so far...

For starters, you now have a Gumball Machine implementation that is *structured* quite different from your first version, and yet *it is exactly the same*. By structurally changing the implementation you've

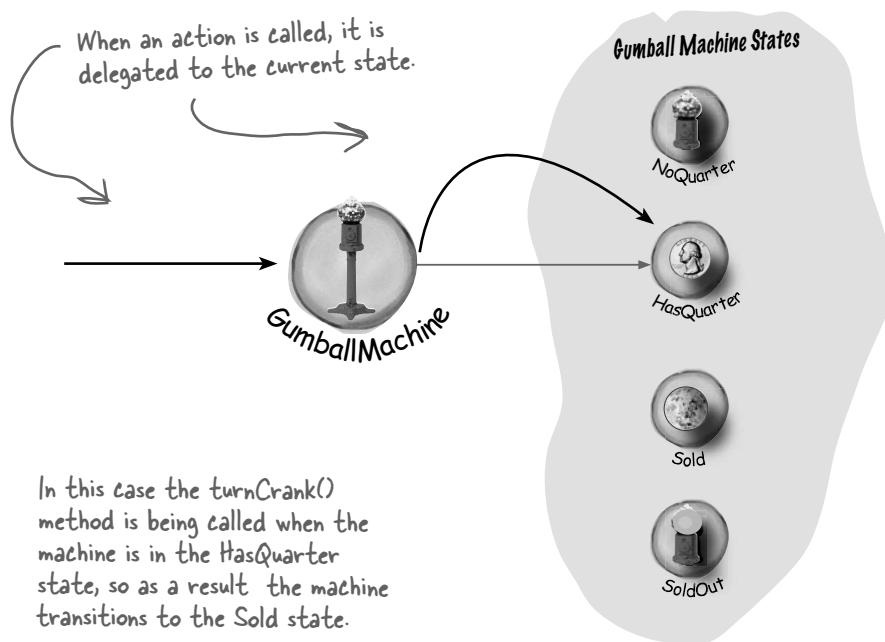
- Localized the behavior of each state into its own class.
- Removed all the troublesome **if** statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes (and we'll do this in a second).
- Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.

Now let's look a little more at the functional aspect of what we did



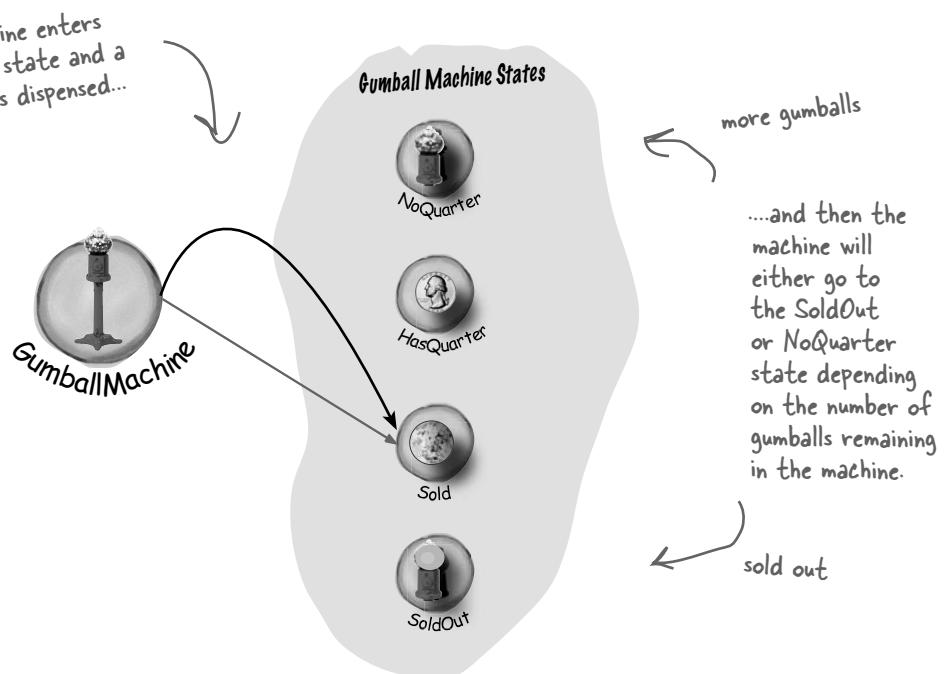
*you are here ▶*

## state transition



In this case the `turnCrank()` method is being called when the machine is in the `HasQuarter` state, so as a result the machine transitions to the `Sold` state.

TRANSITION TO SOLD STATE ↓



later



## e in t e S enes: Sel ui e our



Trace the step of the Gumball Machine starting with the NoQuarter state. Also annotate the diagram with action and output to the machine. For the receipt you can add more states. There are plenty of small inputs to the machine.

①



Gumball Machine States

- NoQuarter
- HasQuarter
- Sold
- SoldOut

②



Gumball Machine States

- NoQuarter
- HasQuarter
- Sold
- SoldOut

③



Gumball Machine States

- NoQuarter
- HasQuarter
- Sold
- SoldOut

④



Gumball Machine States

- NoQuarter
- HasQuarter
- Sold
- SoldOut

# The State Pattern defined

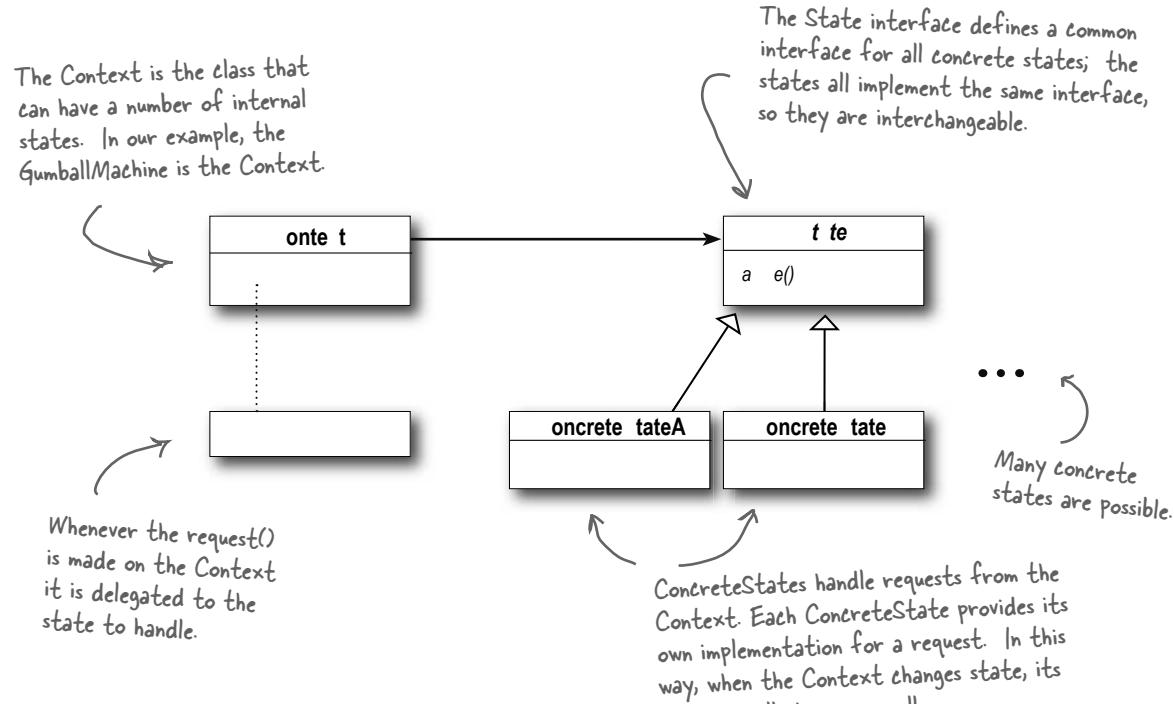
es, it's true, we just implemented the state Pattern so now, let's take a look at what it's all about

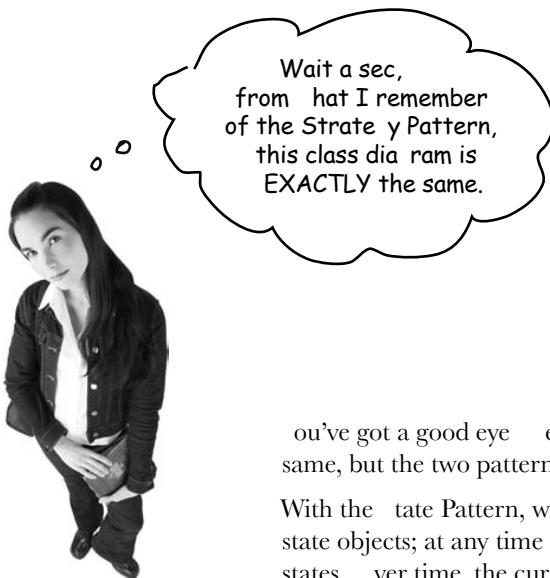
**he state patter** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

he first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example when the gumball machine is in the No Quarter state and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the Has Quarter state (the machine rejects the quarter).

What about the second part of the definition? What does it mean for an object to appear to change its class? Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Okay, now it's time to check out the state Pattern class diagram





You've got a good eye. Yes, the class diagrams are essentially the same, but the two patterns differ in their intent.

With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

With strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the ability to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

*you are here ▶*

## there are no Dumb Questions

**Q:** In the gumball machine the states decide what the next state should be. Do the concrete states always decide what state to go to next?

**A:**

**Q:** It seems like using the State Pattern always increases the number of classes in our designs. So how many more classes our gumball machine had than the original design!

**A:**

**Q:** Do clients ever interact directly with the states?

**A:**

**Q:** The State Pattern class diagram shows that State is an abstract class. But didn't you use an interface in the implementation of the gumball machine's state?

**A:**

**Q:** If I have lots of instances of the State in my application is it possible to share the state objects across them?

**A:**

# We still need to finish the Gumball in game

Remember, we're not done yet. We've got a game to implement; but now that we've got the State Pattern implemented, it should be a breeze. First, we need to add a state to the GumballMachine class

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

    // methods here
}
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

Now let's implement the Winner state class itself, it's remarkably similar to the old state class

```
public class WinnerState implements State {

    // instance variables and constructor
    // insertQuarter error message
    // ejectQuarter error message
    // turnCrank error message

    public void dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs for your quarter");
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

As long as we have a second gumball we release it.

i e entin the in a e

## Finishing the game

We've just got one more change to make we need to implement the random chance game and add a transition to the Winner state. We're going to add both to the HasQuarter state since that is where the customer turns the crank

```
public class HasQuarterState implements State {  
    Random randomWinner = new Random(System.currentTimeMillis());  
    GumballMachine gumballMachine;  
  
    public HasQuarterState(GumballMachine gumballMachine) {  
        this.gumballMachine = gumballMachine;  
    }  
  
    public void insertQuarter() {  
        System.out.println("You can't insert another quarter");  
    }  
  
    public void ejectQuarter() {  
        System.out.println("Quarter returned");  
        gumballMachine.setState(gumballMachine.getNoQuarterState());  
    }  
  
    public void turnCrank() {  
        System.out.println("You turned...");  
        int winner = randomWinner.nextInt(10);  
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {  
            gumballMachine.setState(gumballMachine.getWinnerState());  
        } else {  
            gumballMachine.setState(gumballMachine.getSoldState());  
        }  
    }  
    public void dispense() {  
        System.out.println("No gumball dispensed");  
    }  
}
```

First we add a random number generator to generate the 1 chance of winning...

...then we determine if this customer won.

If they won, and there's enough gumballs left for them to get two, we go to the WinnerState; otherwise, we go to the SoldState (just like we always did).

Wow, that was pretty simple to implement. We just added a new state to the GumballMachine and then implemented it. All we had to do from there was to implement our chance game and transition to the correct state. It looks like our new code strategy is paying off...

ha ter

# Demo for the CEO of Mighty Gumball, Inc.

The CEO of Mighty Gumball has dropped by for a demo of your new gumball game code. Let's hope those states are all in order. We'll keep the demo short and sweet (the short attention span of CEOs is well documented), but hopefully long enough so that we'll win at least once.

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

This code really hasn't changed at all; we just shortened it a bit.

Once, again, start with a gumball machine with 5 gumballs.

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

The whole engineering team is waiting outside the conference room to see if the new State Pattern-based design is going to work!!



*you are here ▶*

te tin the u a a hine



Gee, did we get lucky or what?  
In our demo to the CEO, we →  
won not once, but twice!

```
File Edit Window Help W eni a m allaaw rea er?
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

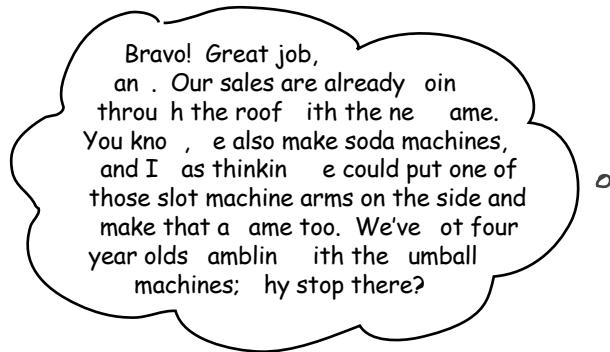
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```

there are no  
**Dumb Questions**

**Q:** Why do we need the Winner state? Couldn't we just have the old state dispense two gumballs?

**A:**



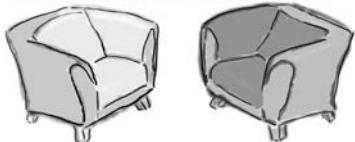
## Sanity check...

Yes, the Customer of Mighty Gumball probably needs a sanity check, but that's not what we're talking about here. Let's think through some aspects of the GumballMachine that we might want to shore up before we ship the gold version.

- We've got a lot of duplicate code in the Old and Winning states and we might want to clean those up. How would we do it? We could make State into an abstract class and build in some default behavior for the methods; after all, error messages like, "You already inserted a quarter, aren't going to be seen by the customer. So all error response behavior could be generic and inherited from the abstract State class." *Dammit Jim, I'm a gumball machine, not a computer!*
- The dispense() method always gets called, even if the crank is turned when there is no quarter. While the machine operates correctly and doesn't dispense unless it's in the right state, we could easily fix this by having turnCrank() return a boolean, or by introducing exceptions. Which do you think is a better solution?
- All of the intelligence for the state transitions is in the State classes. What problems might this cause? Would we want to move that logic into the Gumball Machine? What would be the advantages and disadvantages of that?
- Will you be instantiating a lot of GumballMachine objects? If so, you may want to move the state instances into static instance variables and share them. What changes would this require to the GumballMachine and the states?

*re ide hat tate and trate y*

## Fireside Chats



Tonight: **A Strategy and State pattern evening.**

### Strategy

Hey bro. Did you hear what was in Chapter 7?

was just over giving the template Method guys a hand they needed me to help them finish off their chapter. So, anyway, what is my noble brother up to?

don't know, you always sound like you've just copied what do and you're using different words to describe it. Think about it allow objects to incorporate different behaviors or algorithms through composition and delegation. You're just copying me.

H yeah? How so? I don't get it.

Eah, that was some work... and I'm sure you can see how that's more powerful than inheriting your behavior, right?

Sorry, you're going to have to explain that.

### State

Eah, word is definitely getting around.

ame as always helping classes to exhibit different behaviors in different states.

I admit that what we do is definitely related, but my intent is totally different than yours. And, the way I teach my clients to use composition and delegation is totally different.

Well if you spent a little more time thinking about something other than *use*, you might. Anyway, think about how you work you have a class you're instantiating and you usually give it a strategy object that implements some behavior. Like, in Chapter 7 you were handing out quack behaviors, right? Real ducks got a real quack, rubber ducks got a quack that squeaked.

Yes, of course. Now, think about how work; it's totally different.

*ha ter*

## **Strategy**

Hey, come on, I can change behavior at runtime too; that's what composition is all about

Well, I admit, I don't encourage my objects to have a well defined set of transitions between states. In fact, I typically like to control what strategy my objects are using.

Eh, yeah, keep living your pipe dreams brother. You act like you're a big pattern like me, but check it out I'm in Chapter ; they stuck you way out in Chapter . I mean, how many people are actually going to read this far?

That's my brother, always the dreamer.

## **State**

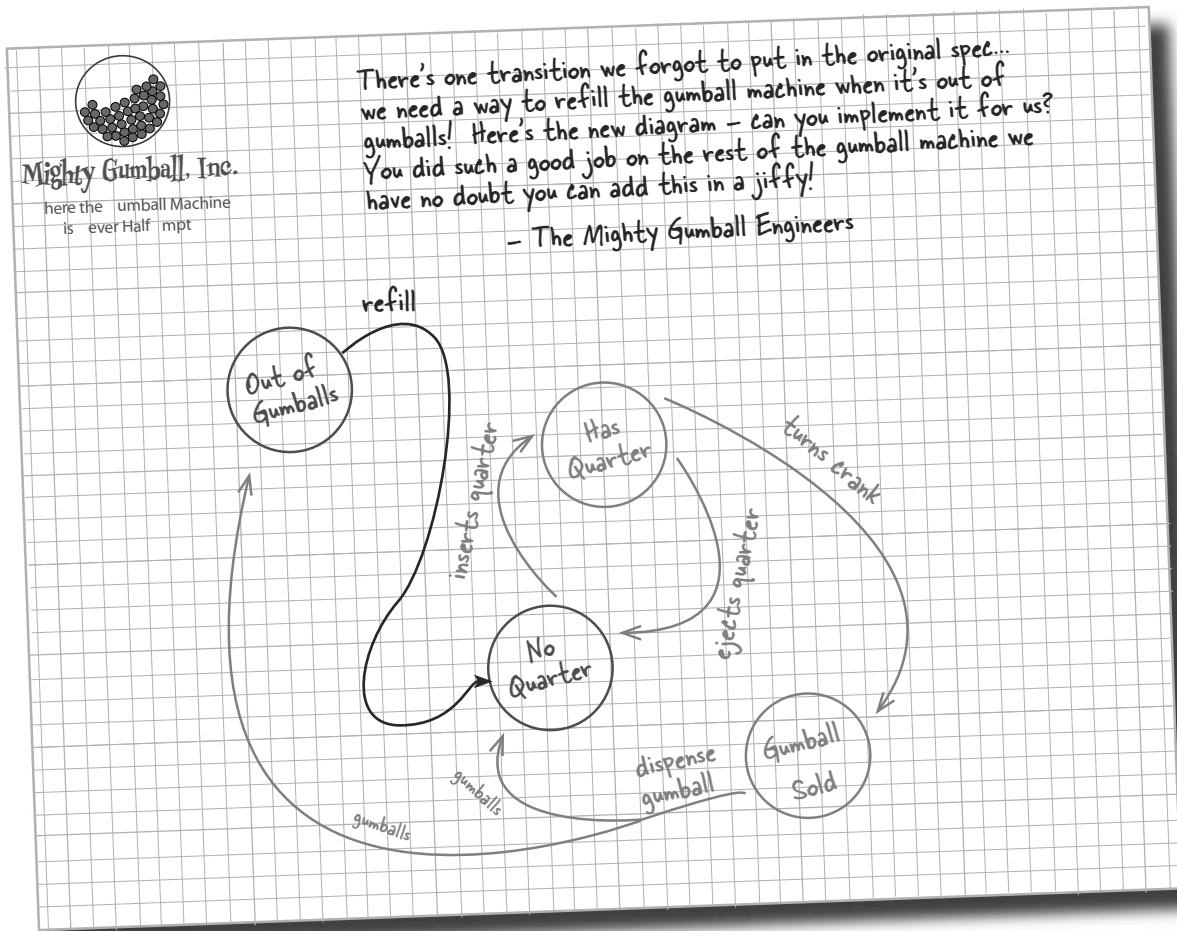
Kay, when my Content objects get created, I may tell them the state to start in, but then they change their own state over time.

Sure you can, but the way I work is built around discrete states; my Content objects change state over time according to some well defined state transitions. In other words, changing behavior is built in to my scheme – it's how I work

Look, we've already said we're alike in structure, but what we do is quite different in intent. Face it, the world has uses for both of us.

Are you kidding? This is a Head First book and Head First readers rock. Of course they're going to get to Chapter

## We almost forgot





## Sharpen your pencil

You need to write the refill method for the gumball machine. It has one argument—the number of gumballs you're adding to the machine—and should update the gumball machine count and reset the machine's state.

You've done some amazing work!  
I've got some more ideas that  
are going to change the gumball  
industry and I need you to implement  
them. Shhhhhh! I'll let you in on these  
ideas in the next chapter.



*you are here ▶*

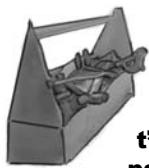
*ho doe*    *hat*

## \* WHO DOES WHAT? \*

Match each pattern with its description

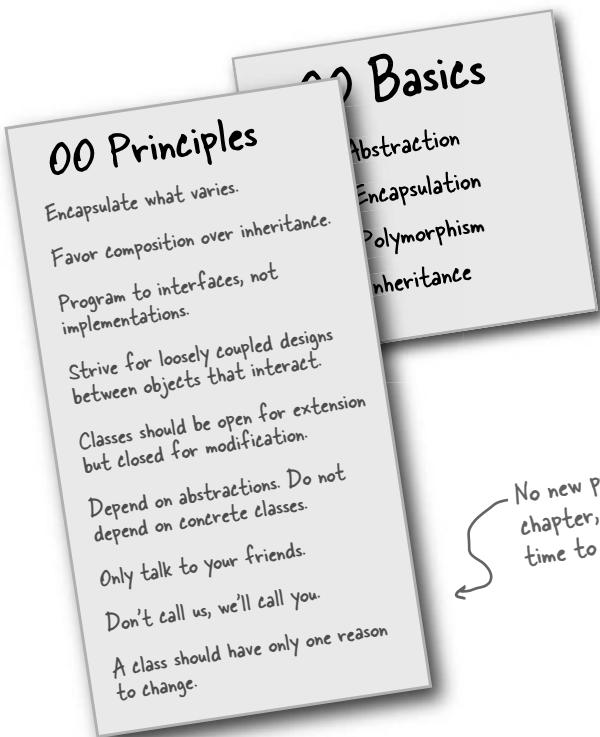
<b>Pattern</b>	<b>Description</b>
tate	n a su ate inter han eab e beha iors an use e e ation to e i e hi h beha ior to use
trate	ub asses e i e ho to im ement ste s in an a orithm
em ate etho	n a su ate state base beha ior an e e ate beha ior to the urrent state

*ha ter*

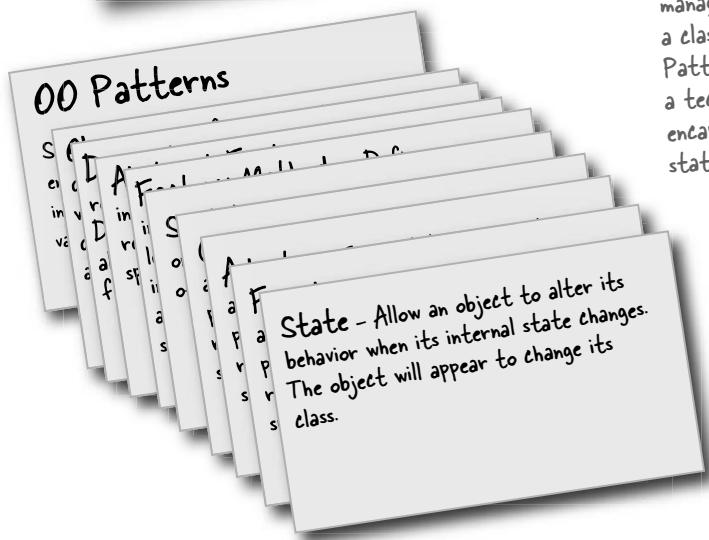


## Tools for your Design Toolbox

It's the end of another chapter you've got enough patterns here to see through any interview



No new principles this chapter, that gives you time to sleep on them.

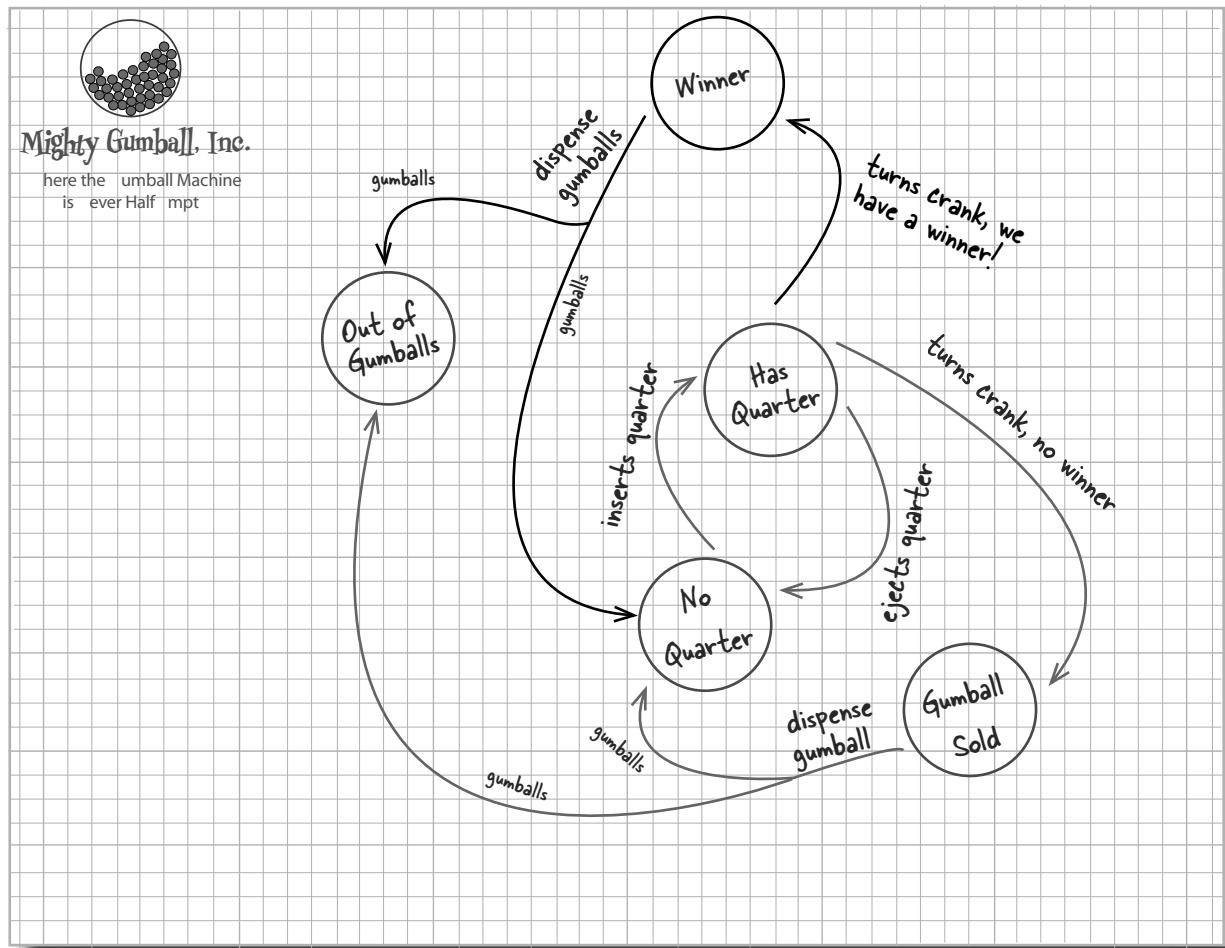


Here's our new pattern. If you're managing state in a class, the State Pattern gives you a technique for encapsulating that state.





# solutions





# Exercise solutions

## Sharpen your pencil



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

- A. his code certainly isn't adhering to the Open/Closed Principle
- B. his code would make a programmer proud.
- C. his design isn't even very object oriented.
- D. state transitions aren't explicit; they are buried in the middle of a bunch of conditional code.
- E. We haven't encapsulated anything that varies here.
- F. Further additions are likely to cause bugs in working code.

## Sharpen your pencil



We are one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully follow the Gumball Machine code below in each iteration. See you ran when we're more or less done...

In the Sold Out state, we really can't do anything until someone refills the Gumball Machine.

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

you are here ▶



## Sharpen your pencil

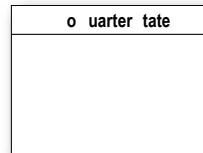
To implement the state, we first need to define what will happen when the corresponding action is called. Annotate the diagram below with the behavior of each action in each class; we've already filled in a few for you.

Go to HasQuarterState

Tell the customer "you haven't inserted a quarter"

Tell the customer "you turned, but there's no quarter"

Tell the customer "you need to pay first"

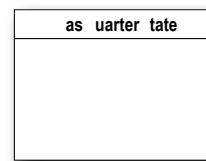


Tell the customer "you can't insert another quarter"

Give back quarter, go to No Quarter state

Go to SoldState

Tell the customer, "no gumball dispensed"

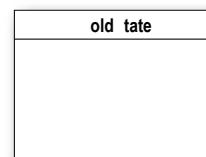


Tell the customer "please wait, we're already giving you a gumball"

Tell the customer "sorry, you already turned the crank"

Tell the customer "turning twice doesn't get you another gumball"

Dispense one gumball. Check number of gumballs; if , go to NoQuarter state, otherwise, go to Sold Out state

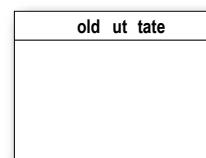


Tell the customer "the machine is sold out"

Tell the customer "you haven't inserted a quarter yet"

Tell the customer "There are no gumballs"

Tell the customer "no gumball dispensed"

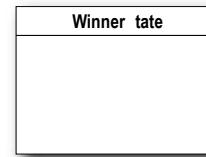


Tell the customer "please wait, we're already giving you a gumball"

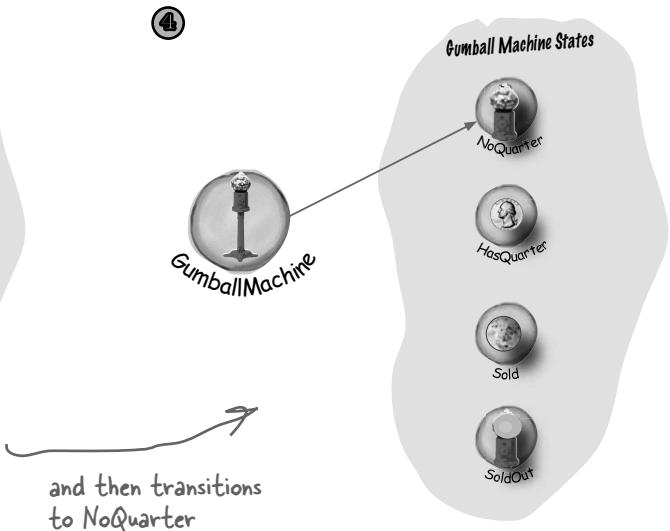
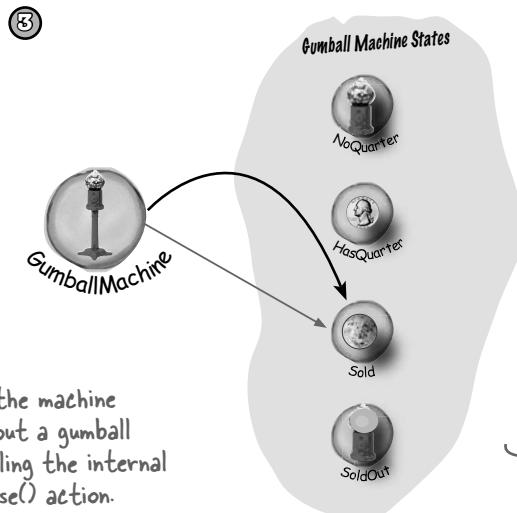
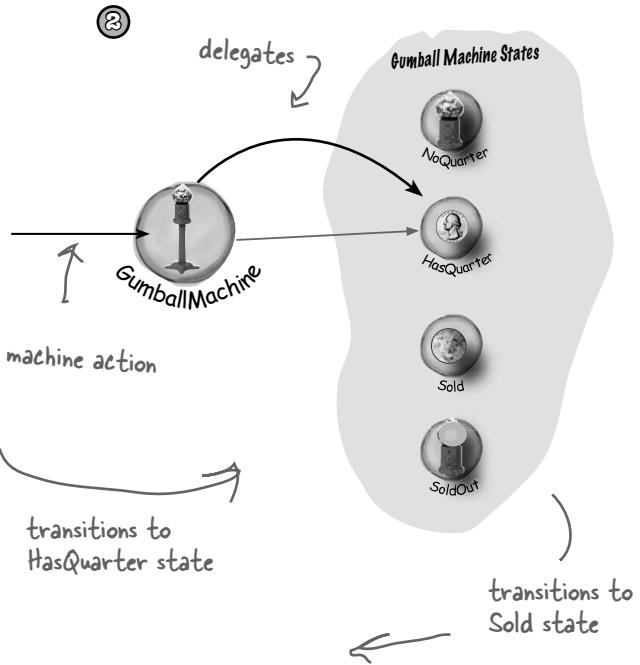
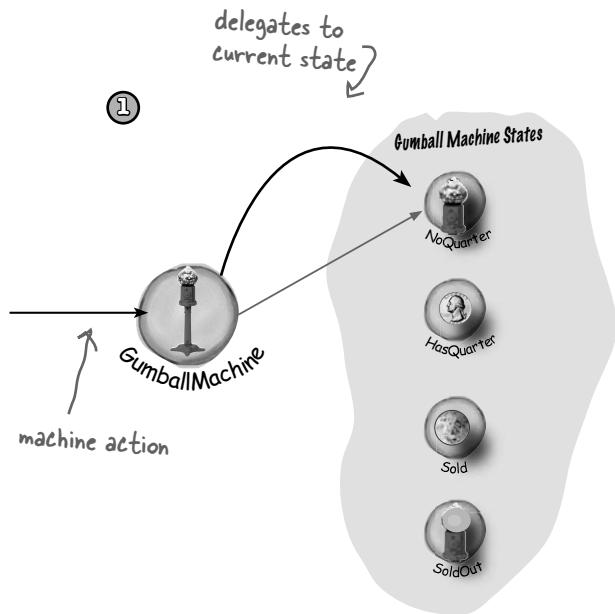
Tell the customer "sorry, you already turned the crank"

Tell the customer "turning twice doesn't get you another gumball"

Dispense two gumballs. Check number of gumballs; if , go to NoQuarter state, otherwise, go to SoldOutState



## e in t e S enes: Sel ui e our Solution



you are here ▶

## exer i e o ution

### \* WHO DOES WHAT? \*

Match each pattern with its description

Pattern	Description
tate	n a su ate inter han eab e behā iors an use e e ation to e i e hi h behā ior to use
trate	ub asses e i e ho to im ement ste s in an a orithm
em ate etho	n a su ate state base behā ior an ee ate behā ior to the urrent state



### Sharpen your pencil

We need you to write the refill() method for the Gumball machine. It has one argument, the number of gumballs you're adding to the machine, and should update the gumball machine count and reset the machine's state.

```
void refill(int count) {  
    this.count = count;  
    state = noQuarterState;  
}
```

ha ter

the ro attern

# **ontrolling ect ccess**

With you as my Proxy, I'll be able to triple the amount of lunch money I can extract from friends!



o re t e ood cop and yo pro ide all yo r er ice in a nice and riendly manner, t yo don t want e eryo e a in yo or er ice , o yo a et e ad cop o trol a ess to yo . T at w at pro ie do: control and mana e acce . A yo re oin to ee,t ere are lots o way in w ic pro ie tand in or t e o ect t ey pro y. Pro ie a e een nown to a lentire met od call o er t e Internet or t eir pro ied o ect ; t ey e al o een nown to patiently tand in t e place or ome pretty la y o ect .



Remember the CEO of  
Mighty Gumball, Inc.?

Hey team, I'd  
really like to get  
some better monitorin' for  
my gumball machines. Can you  
find a way to let me a report of  
inventory and machine state?

ounds easy enough. If you remember, we've already got methods in the gumball machine code for getting the count of gumballs (`getCount()`), and getting the current state of the machine (`getState()`).

All we need to do is create a report that can be printed out and sent back to the CEO. Hmm, we should probably add a location field to each gumball machine as well; that way the CEO can keep the machines straight.

Let's just jump in and code this. We'll impress the CEO with a very fast turnaround.

# Coding the Monitor

Let's start by adding support to the GumballMachine class so that it can handle locations

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

The location is passed into the constructor and stored in the instance variable.

Let's also add a getter method to grab the location when we need it.

Now let's create another class, GumballMonitor, that retrieves the machine's location, inventory of gumballs and the current machine state and prints them in a nice little report

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report method just prints a report with location, inventory and the machine's state.

## Testing the Monitor

We implemented that in no time. The C# is going to be thrilled and amazed by our development skills.

Now we just need to instantiate a GumballMonitor and give it a machine to monitor

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);

        // rest of test code here

        monitor.report();
    }
}
```

*↑ When we need a report on the machine, we call the report() method.*

*Pass in a location and initial count of gumballs on the command line.*

*Don't forget to give the constructor a location and count...*

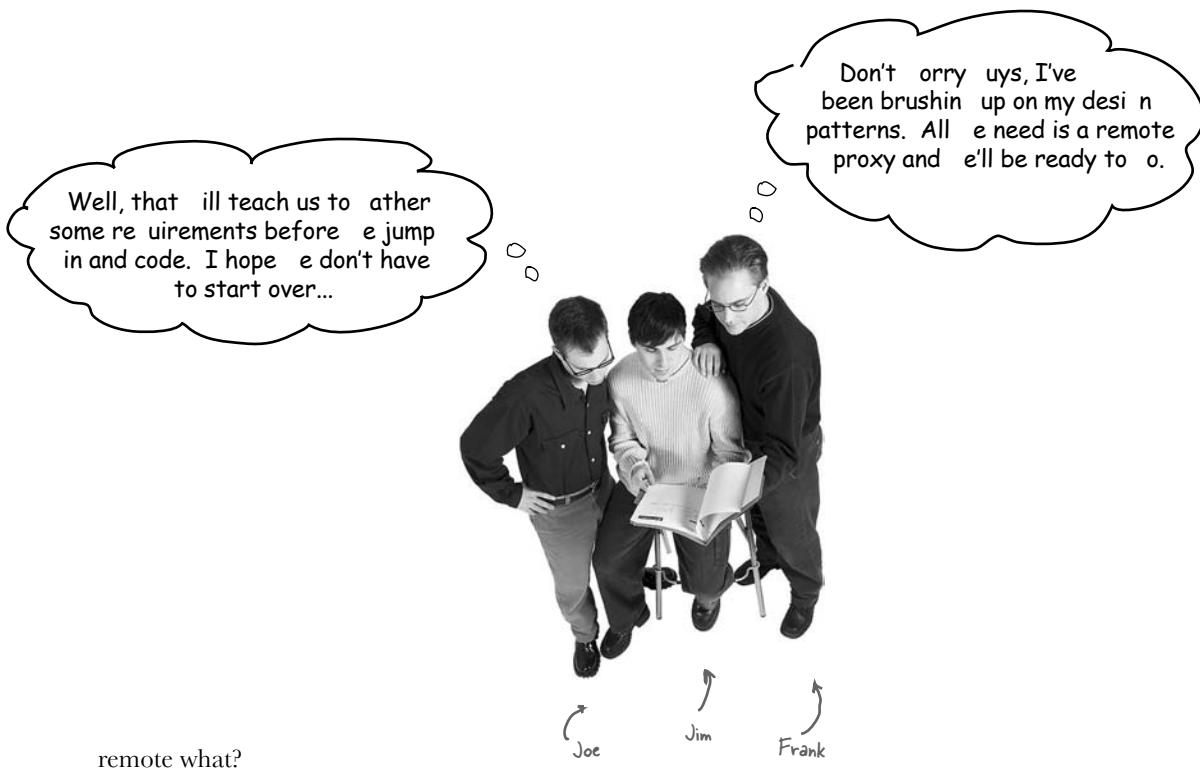
*...and instantiate a monitor and pass it a machine to provide a report on.*

```
File Edit Window Help Flyin Fi
%java GumballMachineTestDrive Seattle 112
Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```



The monitor output looks neat, but I guess I wasn't clear. I need to monitor gumball machines REMOTELY! In fact, we already have the networks in place for monitoring. Come on guys, you're supposed to be the Internet generation!

*↑ And here's the output!*



remote what?

*em te r y.* Think about it we've already got the monitor code written, right? We give the GumballMonitor a reference to a machine and it gives us a report. The problem is that monitor runs in the same M as the gumball machine and the C wants to sit at his desk and *rem te y* monitor the machines so what if we left our GumballMonitor class as is, but handed it a proxy to a *rem te* object?

'm not sure get it.

Me neither.

Let's start at the beginning... a proxy is a stand in for a *re* object. In this case, the proxy acts just like it is a Gumball Machine object, but behind the scenes it is communicating over the network to talk to the real, remote GumballMachine.

*o* you're saying we keep our code as it is, and we give the monitor a reference to a proxy version of the GumballMachine...

nd this proxy pretends it's the real object, but it's really just communicating over the net to the real object.

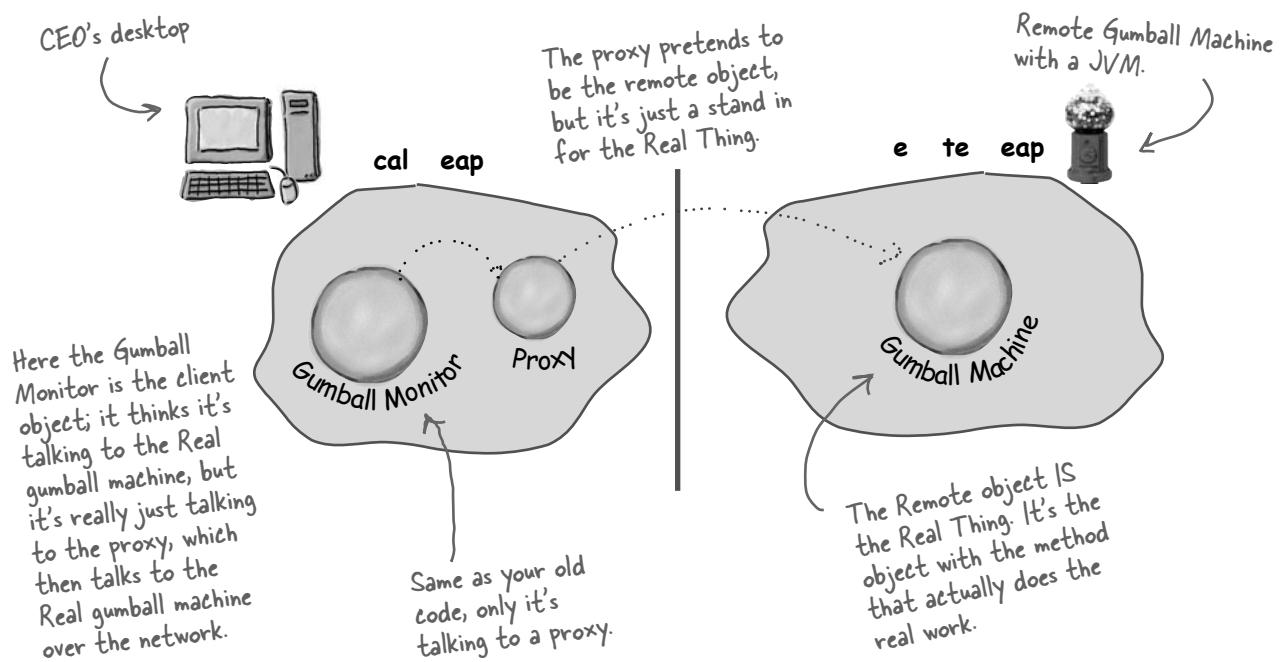
eah, that's pretty much the story.

t sounds like something that is easier said than done.

Perhaps, but don't think it'll be that bad. We have to make sure that the gumball machine can act as a service and accept requests over the network; we also need to give our monitor a way to get a reference to a proxy object, but we've got some great tools already built into Java to help us. Let's talk a little more about remote proxies first...

## The role of the remote proxy'

remote proxy acts as a *representative* for the *remote object*. What's a *remote object*? It's an object that lives in the heap of a different virtual Machine (or more generally, a remote object that is running in a different address space). What's a *local representative*? It's an object that you can call local methods on and have them forwarded on to the remote object.



our client objects like its a ing re ote et o alls  
ut at its reall oing is aling et o s on a ea  
lo al ro o e tt at an les all t e lo le el etails o  
net or o uni ation



Before going further, in a short video I will enable remote method invocation. How would you make a proxy to invoke a remote object? How would you make a remote invocation example?



Should main remote calls be totally transparent? Is that a good idea? What might be a problem with that approach?

# Adding a remote proxy to the Gumball Machine monitoring code

In paper this looks good, but how do we create a proxy that knows how to invoke a method on an object that lives in another M?

Hmmm. Well, you can't get a reference to something on another heap, right? In other words, you can't say

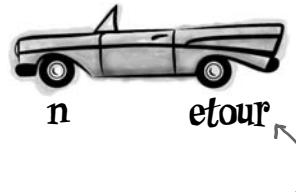
**Duck d = <object in another heap>**

Whatever the variable d is referencing must be in the same heap space as the code running the statement. So how do we approach this? Well, that's where Java's Remote Method Invocation comes in... M gives us a way to find objects in a remote M and allows us to invoke their methods.

You may have encountered M in Head First Java; if not, we're going to take a slight detour and come up to speed on M before adding the proxy support to the Gumball Machine code.

So, here's what we're going to do

- 1 First, we're going to take the Detour and check out even if you are unfamiliar with , you might want to follow along and check out the scenery**
- 2 Then we're going to take our gumball machine and make it a remote service that provides a set of methods calls that can be invoked remotely**
- 3 Then, we're going to create a proxy that can talk to a remote gumball machine, again using , and put the monitoring system all together so that the can monitor any number of remote machines**



If you're new to RMI, take the detour that runs over the next few pages; otherwise, you might want to just quickly thumb through the detour as a review.



## Remote methods

Let's say we want to design a system that allows us to call a local object that forwards each request to a remote object. How would we design it? We'd need a couple of helper objects that actually do the communicating for us. The helpers make it possible for the client to act as though it's calling a method on a local object (which in fact, it is). The client calls a method on the client helper, as if the client helper were the actual service. The client helper then takes care of forwarding that request for us.

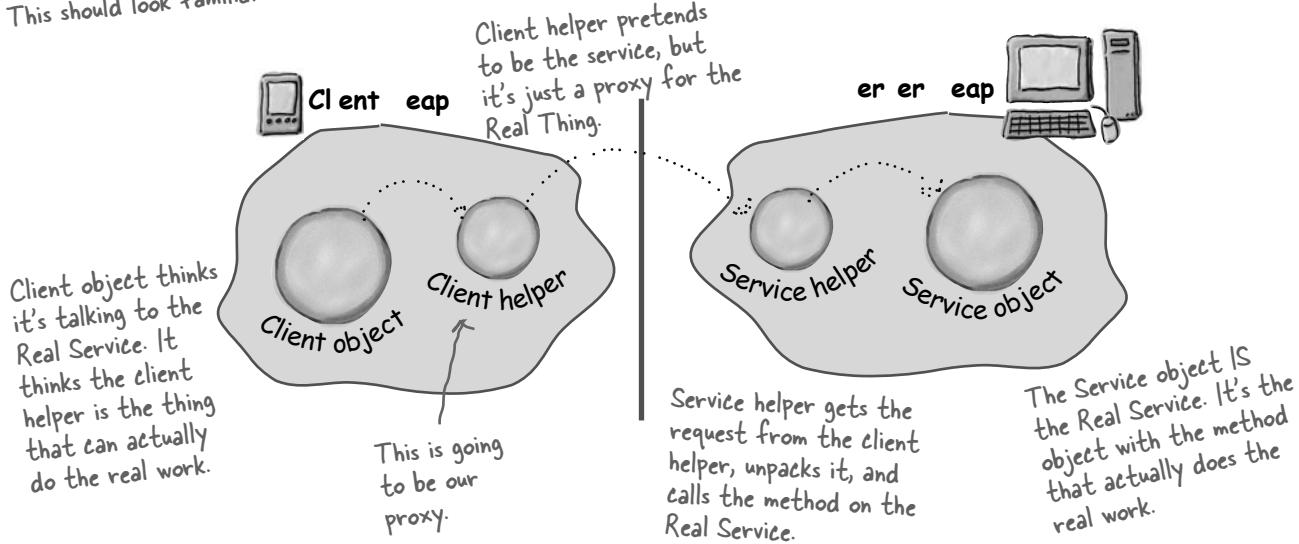
In other words, the client object thinks it's calling a method on the remote service, because the client helper is pretending to be the service object. Pretending to be the thing with the method the client wants to call.

But the client helper isn't really the remote service. Although the client helper acts like it (because it has the same method that the service is advertising), the client helper doesn't have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

On the server side, the service helper receives the request from the client helper (through a socket connection), unpacks the information about the call, and then invokes the real method on the real service object. To the service object, the call is local. It's coming from the service helper, not a remote client.

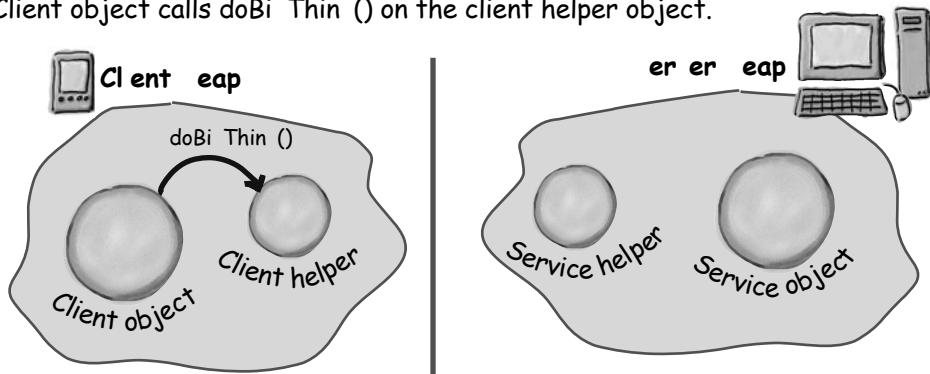
The service helper gets the return value from the service, packs it up, and ships it back (over a socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

This should look familiar...

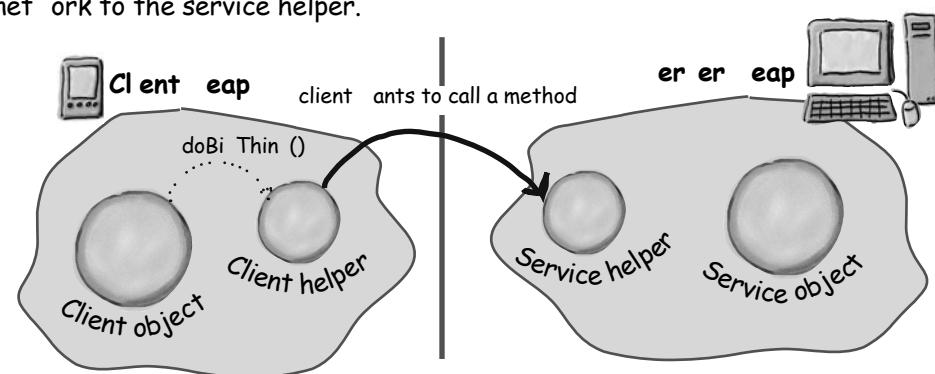


## How the method call happens

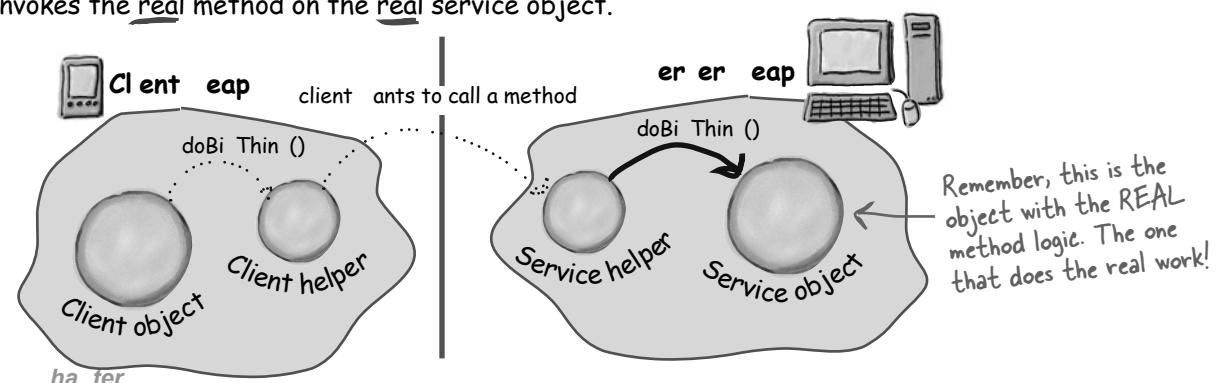
- Client object calls doBiThin() on the client helper object.

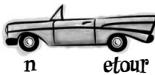


- Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.

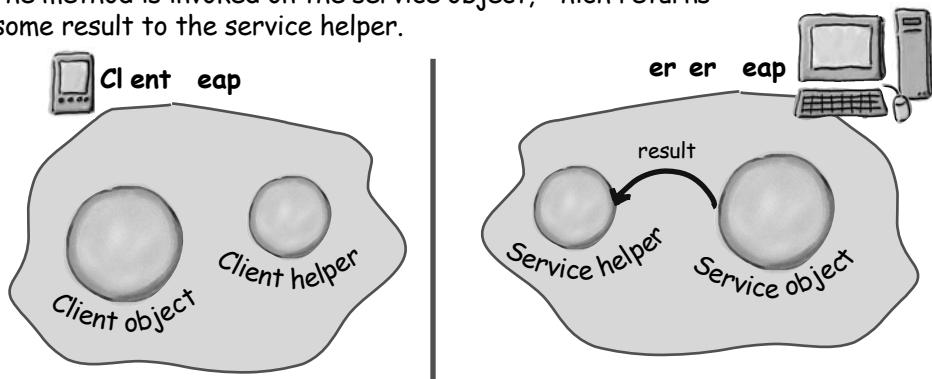


- Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.

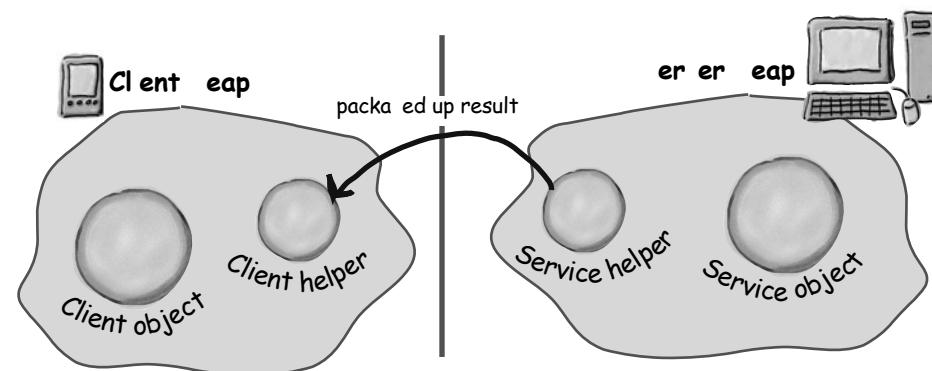




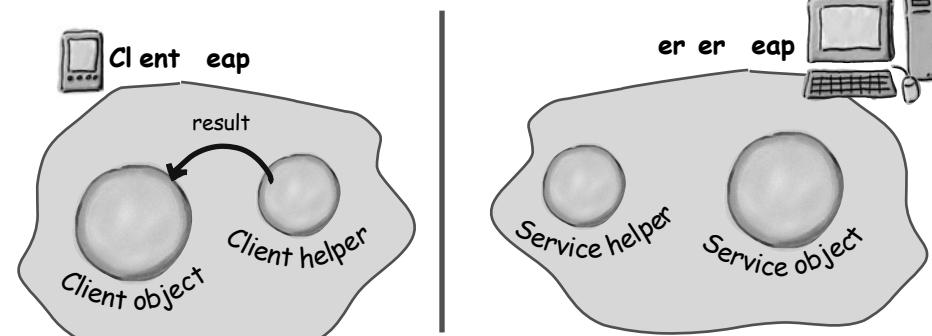
- The method is invoked on the service object, which returns some result to the service helper.



- Service helper packages up information returned from the call and ships it back over the network to the client helper.



- Client helper unpackages the returned values and returns them to the client object. To the client object, this is all transparent.



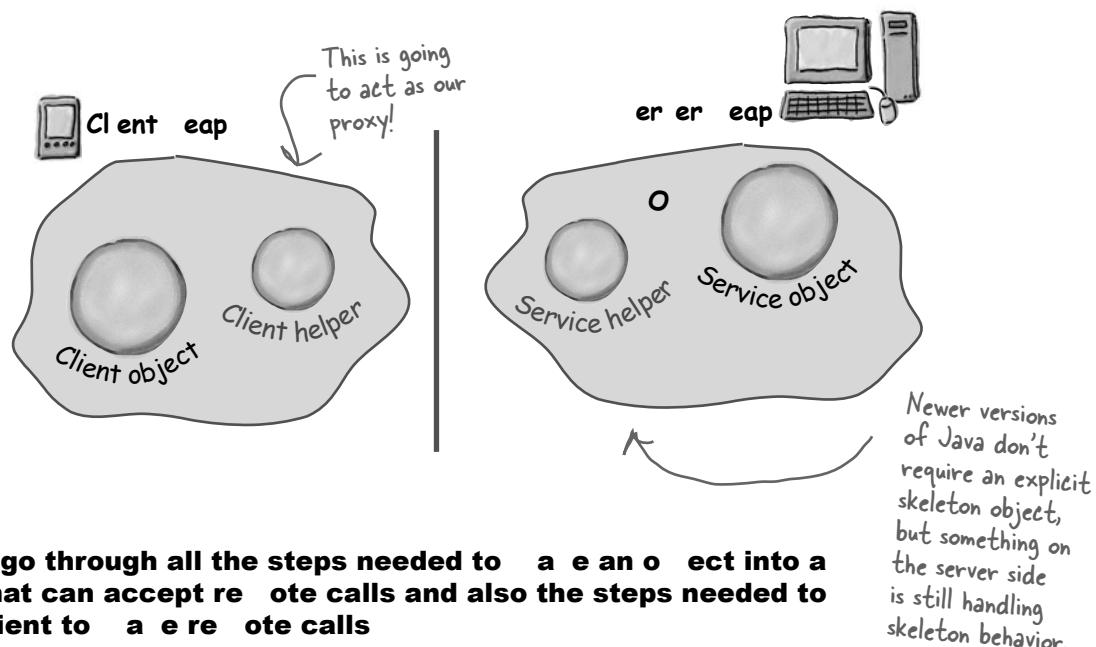
## Java RMI, the Big Picture

Okay, you've got the gist of how remote methods work; now you just need to understand how to use M to enable remote method invocation.

What M does for you is build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. The nice thing about M is that you don't have to write any of the networking or code yourself. With your client, you call remote methods (i.e., the ones the service has) just like normal method calls on objects running in the client's own local M.

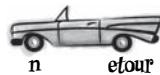
M also provides all the runtime infrastructure to make it all work, including a lookup service that the client can use to find and access the remote objects.

**No enclave in , the client helper is a stub and the service helper is a skeleton'**



**Now let's go through all the steps needed to connect into a service that can accept remote calls and also the steps needed to allow a client to make remote calls**

**You might want to ensure your seat belt is fastened there are a lot of steps and a few ups and curves but nothing to be too worried about**



## a ing the e ote service

This is an **o er ew** of the five steps for making the remote service. In other words, the steps needed to take an ordinary object and supercharge it so it can be called by a remote client. We'll be doing this later to our GumballMachine. For now, let's get the steps down and then we'll explain each one in detail.

### tep one

#### Make a e te nter ace

The remote interface defines the methods that a client can call remotely. It's what the client will use as the class type for your service. Both the stub and actual service will implement this.

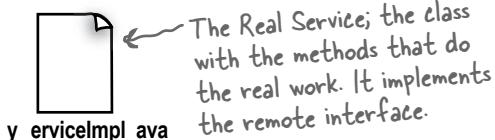


### tep two

#### Make a e te ple ementat n

This is the class that does the real work. It has the real implementation of the remote methods defined in the remote interface.

It's the object that the client wants to call methods on (e.g., our GumballMachine).



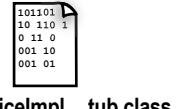
### tep three

#### Generate the st bs and s elect ns usin rmic

These are the client and server helpers'. You don't have to create these classes or ever look at the source code that generates them. It's all handled automatically when you run the rmic tool that ships with your Java development kit.

Running rmic against the actual service implementation class... → ...spits out two new classes for the helper objects.

```
File Edit Window Help Eat
% rmic MyServiceImpl
```



### tep four

#### Start the re str (rmiregistry)

The rmiregistry is like the white pages of a phone book. It's where the client goes to get the proxy (the client stub helper object).

```
File Edit Window Help Drin
% rmiregistry
```

Run this in a separate terminal.

### tep ve

#### Start the re te ser ce

You have to get the service object up and running. Our service implementation class instantiates an instance of the service and registers it with the M registry. Registering it makes the service available for clients.

```
File Edit Window Help BeMerry
% java MyServiceImpl
```

*you are here ▶*

## tep one a e a e ote interface

### ten ja a r e te

Remote is a marker interface, which means it has no methods. It has special meaning for M, though, so you must follow this rule. Notice that we say extends here. One interface is allowed to extend another interface.

```
public interface MyRemote extends Remote {
```

This tells us that the interface is going to be used to support remote calls.

### Declare that all methods t r a e te cept n

The remote interface is the one the client uses as the type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and all kinds of Bad things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*; ← Remote interface is in java.rmi
```

```
public interface MyRemote extends Remote {  
    public String sayHello() throws RemoteException;  
}
```

Every remote method call is considered risky. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

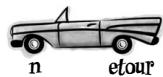
### es re ar ents an ret rn al es are pr t es r er al able

Arguments and return values of a remote method must be either primitive or serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through serialization. Same thing with return values. If you use primitives, strings, and the majority of types in the P (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

Check out Head First Java if you need to refresh your memory on Serializable.

```
public String sayHello() throws RemoteException;
```

↑ This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.



## Step two: a real remote implementation

### Implementation interface

our service has to implement the remote interface – the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() { ←
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

### UnicastRemoteObject

In order to work as a remote service object, your object needs some functionality related to being 'remote'. The simplest way is to extend `UnicastRemoteObject` (from the `java.rmi.server` package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

### ③ Create an or constructor that declares a remote exception

our new superclass, `UnicastRemoteObject`, has one little problem – its constructor throws a `RemoteException`. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the exception. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException {}
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

### Register the service + the rebind

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the `M` registry (which must be running or this line of code fails). When you register the implementation object, the `M` system actually puts the `stub` in the registry, since that's what the client really needs. Register your service using the static `rebind()` method of the `java.rmi.Naming` class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

## **tep three generate stu s and s eletons**

### **○ n r c n t e re te ple entat n class n t t e re te nter ace**

The rmic tool, which comes with the Java software development kit, takes a service implementation and creates two new classes, the stub and the skeleton. It uses a naming convention that is the name of your remote implementation, with either `.stub` or `.skeleton` added to the end. Here are other options with rmic, including not generating skeletons, seeing what the source code for these classes looked like, and even using `-P` as the protocol. The way we're doing it here is the way you'll usually do it. The classes will land in the current directory (i.e. whatever you did a `cd` to). Remember, rmic must be able to see your implementation class, so you'll probably run rmic from the directory where your remote implementation is located. (We're deliberately not using packages here, to make it simpler. In the real world, you'll need to account for package directory structures and fully qualified names).

*Notice that you don't say ".class" on the end. Just the class name.*

```
File Edit Window Help W e
%rmic MyRemoteImpl
```

*RMIC generates two new classes for the helper objects.*



`MyRemoteImplStub.class`



`MyRemoteImplSkeleton.class`

## **tep four run r iegistry**

### **○ rn p a ter nal an start t er re str**

Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your classes' directory.

File Edit Window Help H ?

%rmiregistry

## **tep ve start the service**

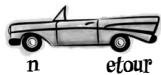
### **○ rn p a n t er ter nal an start r ser ce**

This might be from a `main()` method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a `main` method that instantiates the object and registers it with the registry.

*ha ter*

File Edit Window Help H ?

%java MyRemoteImpl



## o plete code for the server side

```

import java.rmi.*;           ← RemoteException and Remote
                            interface are in java.rmi package.
public interface MyRemote extends Remote {           ← Your interface MUST extend java.rmi.Remote
    public String sayHello() throws RemoteException;   ← All of your remote methods must
}                                                       declare a RemoteException.

```

## he e ote service the i ple entation

```

import java.rmi.*;           ← UnicastRemoteObject is in the
import java.rmi.server.*;     ← java.rmi.server package.
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {           ← You have to implement all the
        return "Server says, 'Hey'";     ← interface methods, of course. But
    }                                   notice that you do NOT have to
                                         declare the RemoteException.
    public MyRemoteImpl() throws RemoteException {}           ← You MUST implement your
                                                               remote interface!!
    public static void main (String[] args) {
        try {
            MyRemote service = new MyRemoteImpl();           ← Your superclass constructor (for
            Naming.rebind("RemoteHello", service);           ← UnicastRemoteObject) declares an exception, so
            } catch (Exception ex) {                         ← YOU must write a constructor, because it means
                ex.printStackTrace();                      ← that your constructor is calling risky code (its
                                                               super constructor).
            }
        }
    }

```

← Make the remote object, then bind' it to the rmiregistry using the static Naming.rebind(). The use to look it up in the RMI registry.

you are here ▶

## estecte t ette st b bject

The client has to get the stub object (our proxy), since that's the thing the client will call methods on. And that's where the M registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, Here's a name, and I'd like the stub that goes with that name.

Let's take a look at the code we need to lookup and retrieve a stub object.

Here's how it works.



### Code Up Close

The client always uses the remote interface as the type of the service. In fact, the client never needs to know the actual class name of your remote service.

MyRemote service =

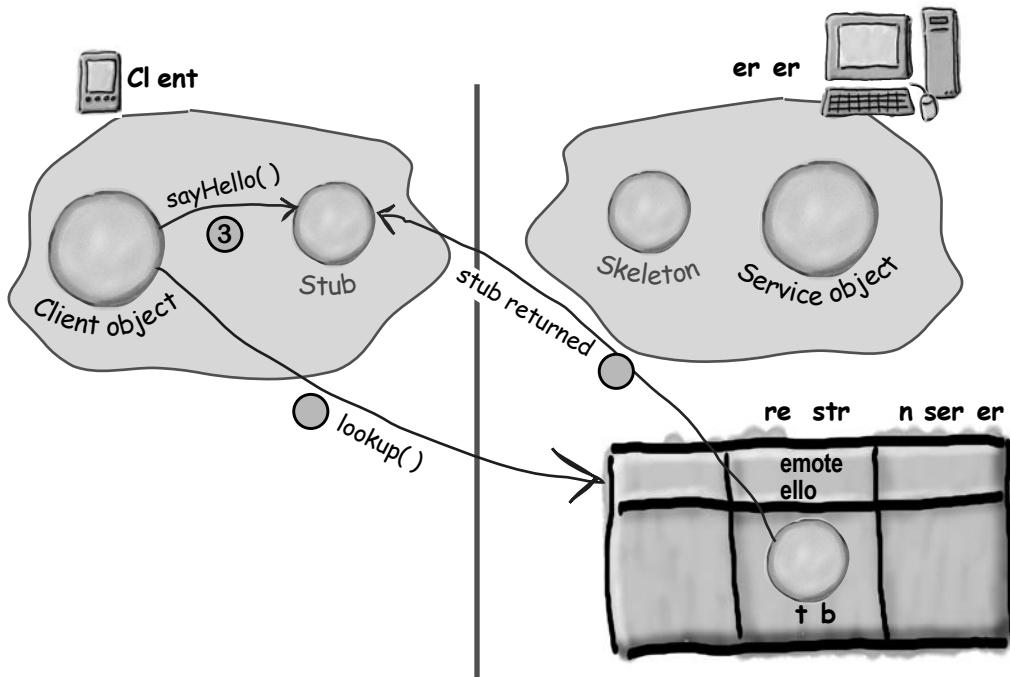
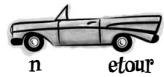
(MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

lookup() is a static method of the Naming class.

This must be the name that the service was registered under.

You have to cast it to the interface, since the lookup method returns type Object.

The host name or IP address where the service is running.



## How it works

### ① Client establishes connection

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

### ② re str ret ns t e st b object

(as the return value of the lookup method) and M deserializes the stub automatically. You must have the stub class (that rmic generated for you) on the client or the stub won't be deserialized.

### ③ Client creates a test based stub to the real service

*you are here ▶*

## o plete client code

```

import java.rmi.*;
The Naming class (for doing the rmiregistry
lookup) is in the java.rmi package.

public class MyRemoteClient {
    public static void main (String[] args) {
        new MyRemoteClient().go();
    }

    public void go() {
        try {
            MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
            String s = service.sayHello();
            System.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

It comes out of the registry as type Object, so don't forget the cast.

You need the IP address or hostname.

and the name used to bind/rebind the service.

It looks just like a regular old method call! (Except it must acknowledge the RemoteException.)



ee S

### How does the client get the stub class?

ow we get to the interesting question. Somehow, some way, the client must have the stub class that you generated earlier using rmic at the time the client does the lookup, or else the stub won't be deserialized on the client and the whole thing blows up. The client also needs classes for serialized objects returned by method calls to the remote object. In a simple system, you can simply deliver these classes to the client.

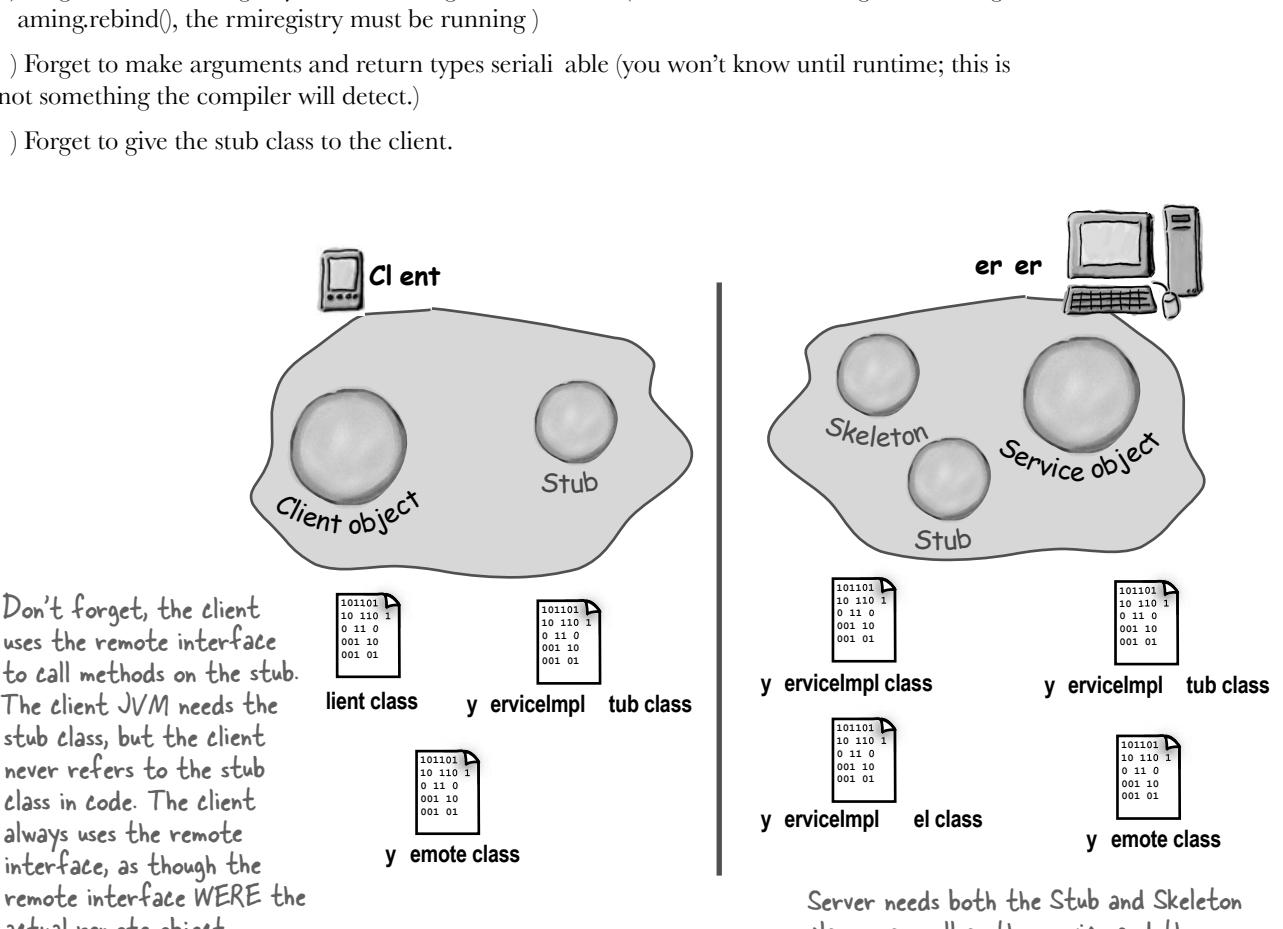
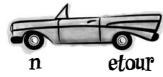
here's a much cooler way, although it's beyond the scope of this book. But just in case you're interested, the cooler way is called dynamic class downloading. With dynamic class downloading, Serialized objects like the stub are stamped with a URL that tells the JVM system on the client where to find the class file for that object. Then, in the process of deserializing an object, if the JVM can't find the class locally, it uses that URL to do an HTTP get to retrieve the class file. So you'd need a simple web server to serve up class files, and you'd also need to change some security parameters on the client. Here are a few other tricky issues with dynamic class downloading, but that's the overview.

or the stub object specification, there's another way the client can get the class. This is only available in Java 6, though. I'll briefly talk about this near the end of the chapter.



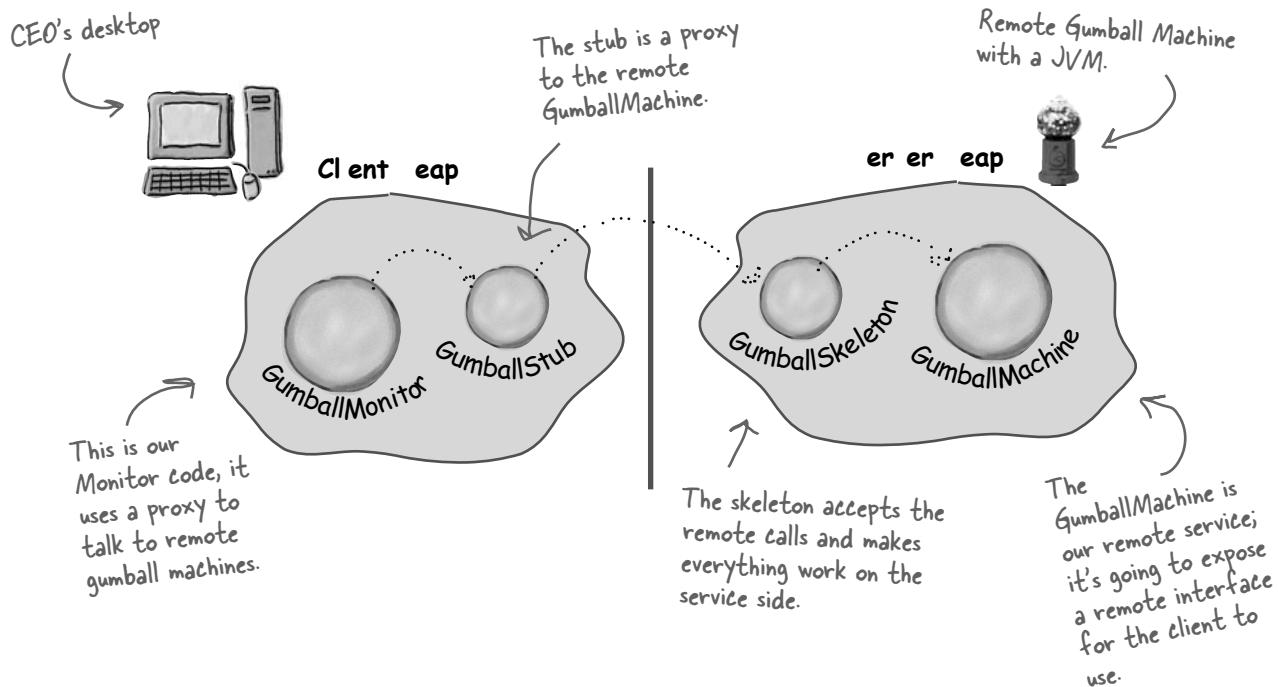
he top three things programmers do wrong with M are

- ) Forget to start rmiregistry before starting remote service (when the service is registered using Naming.rebind(), the rmiregistry must be running)
- ) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)
- ) Forget to give the stub class to the client.



## Back to our GumballMachine remote proxy

ay, now that you have the basics down, you've got the tools you need to implement the gumball machine remote proxy. Let's take a look at how the gumball machines fit into this framework.



# Getting the GumballMachine ready to be a remote service

The first step in converting our code to use the remote proxy is to enable the GumballMachine to service remote requests from clients. In other words, we're going to make it into a service. To do that, we need to

1) Create a remote interface for the GumballMachine. This will provide a set of methods that can be called remotely.

2) Make sure all the return types in the interface are serializable.

3) Implement the interface in a concrete class.

We'll start with the remote interface

```
Don't forget to import java.rmi.*;           This is the remote interface.
import java.rmi.*;                         ↗

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
↑                                         ↗
All return types need                   Here are the methods we're going to support.
to be primitive or                     Each one throws RemoteException.
Serializable...
```

We have one return type that isn't serializable—the State class. Let's fix it up...

```
import java.io.*;           ← Serializable is in the java.io package.
public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
↑
Then we just extend the Serializable
interface (which has no methods in it).
And now State in all the subclasses can
be transferred over the network.
```

*you are here ▶*

## re ote inter a e or the u a a hine

ctually, we're not done with serializable yet; we have one problem with state. As you may remember, each state object maintains a reference to a gumball machine so that it can call the gumball machine's methods and change its state. We don't want the entire gumball machine serialized and transferred with the state object. Here is an easy way to fix this

```
public class NoQuarterState implements State {  
    transient GumballMachine gumballMachine;  
  
    // all other methods here  
}
```

In each implementation of State, we add the transient keyword to the GumballMachine instance variable. This tells the JVM not to serialize this field. Note that this can be slightly dangerous if you try to access this field once its been serialized and transferred.

We've already implemented our GumballMachine, but we need to make sure it can act as a service and handle requests coming from over the network. To do that, we have to make sure the GumballMachine is doing everything it needs to implement the GumballMachine remote interface.

As you've already seen in the M detour, this is quite simple, all we need to do is add a couple of things...

First, we need to import the rmi packages.

```
import java.rmi.*;  
import java.rmi.server.*;  
  
public class GumballMachine  
    extends UnicastRemoteObject implements GumballMachineRemote  
{  
    // instance variables here  
  
    public GumballMachine(String location, int numberGumballs) throws RemoteException {  
        // code here  
    }  
  
    public int getCount() {  
        return count;  
    }  
  
    public State getState() {  
        return state;  
    }  
  
    public String getLocation() {  
        return location;  
    }  
  
    // other methods here  
}
```

GumballMachine is going to subclass the UnicastRemoteObject; this gives it the ability to act as a remote service.

GumballMachine also needs to implement the remote interface...

...and the constructor needs to throw a remote exception, because the superclass does.

That's it! Nothing changes here at all!

ha ter

# Registering with the RMI registry...

hat completes the gumball machine service. Now we just need to fire it up so it can receive requests. First, we need to make sure we register it with the RM registry so that clients can locate it.

We're going to add a little code to the test drive that will take care of this for us

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;
        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }
        try {
            count = Integer.parseInt(args[1]);
            gumballMachine =
                new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

First we need to add a try/catch block around the gumball instantiation because our constructor can now throw exceptions.

We also add the call to Naming.rebind, which publishes the GumballMachine stub under the name gumballmachine.

We're using the "official" Mighty Gumball machines, you should substitute your own machine name here.

Let's go ahead and get this running...

```
File Edit Window Help H ?
% rmiregistry
File Edit Window Help H ?
% java GumballMachineTestDrive seattle.mightygumball.com 100
```

Run this second.

This gets the GumballMachine up and running and registers it with the RMI registry.

## Now for the GumballMonitor client...

Remember the GumballMonitor? We wanted to reuse it without having to rewrite it to work over a network. Well, we're pretty much going to do that, but we do need to make a few changes.

```
import java.rmi.*;           ← We need to import the RMI package because we are
                                using the RemoteException class below...
public class GumballMonitor {   ← Now we're going to rely on the remote
    GumballMachineRemote machine; interface rather than the concrete
                                GumballMachine class.
    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

← We also need to catch any remote exceptions that might happen as we try to invoke methods that are ultimately happening over the network.



# Writing the Monitor test drive

Now we've got all the pieces we need. We just need to write some code so the CEO can monitor a bunch of gumball machines

Here's the monitor test drive. The CEO is going to run this!

```

import java.rmi.*;
public class GumballMonitorTestDrive {
    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};
        GumballMonitor[] monitor = new GumballMonitor[location.length];
        for (int i=0; i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                    (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        for (int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}

```

Here's all the locations we're going to monitor. We create an array of locations, one for each machine.

We also create an array of monitors.

Now we need to get a proxy to each remote machine.

Then we iterate through each machine and print out its report.



## Code Up Close

This returns a proxy to the remote Gumball Machine (or throws an exception if one can't be located).

```
try {
    GumballMachineRemote machine =
        (GumballMachineRemote) Naming.lookup(location[i]);
    monitor[i] = new GumballMonitor(machine);
} catch (Exception e) {
    e.printStackTrace();
}
```

Remember, Naming.lookup() is a static method in the RMI package that takes a location and service name and looks it up in the rmiregistry at that location.

Once we get a proxy to the remote machine, we create a new GumballMonitor and pass it the machine to monitor.

## Another demo for the CEO of Mighty Gumball...

Okay, it's time to put all this work together and give another demo. First let's make sure a few gumball machines are running the new code

On each machine, run rmiregistry in the background or from a separate terminal window...

...and then run the GumballMachine, giving it a location and an initial gumball count.

```
File Edit Window Help H ?
% rmiregistry &
% java GumballMachine santafe.mightygumball.com 100
```

```
File Edit Window Help H ?
% rmiregistry &
% java GumballMachine boulder.mightygumball.com 100
```

```
File Edit Window Help H ?
% rmiregistry &
% java GumballMachine seattle.mightygumball.com 250
popular machine!
```

And now let's put the monitor in the hands of the CEO.  
Hopefully this time he'll love it:

```
File Edit Window Help G m all AndBeyond
% java GumballMonitor
Gumball Machine: santafe.mightygumball.com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of crank

Gumball Machine: seattle.mightygumball.com
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

The monitor iterates over each remote machine and calls its getLocation(), getCount() and getState() methods.

This is ama in ;  
it's oin to revolutioni e my  
business and blo a ay the  
competition!

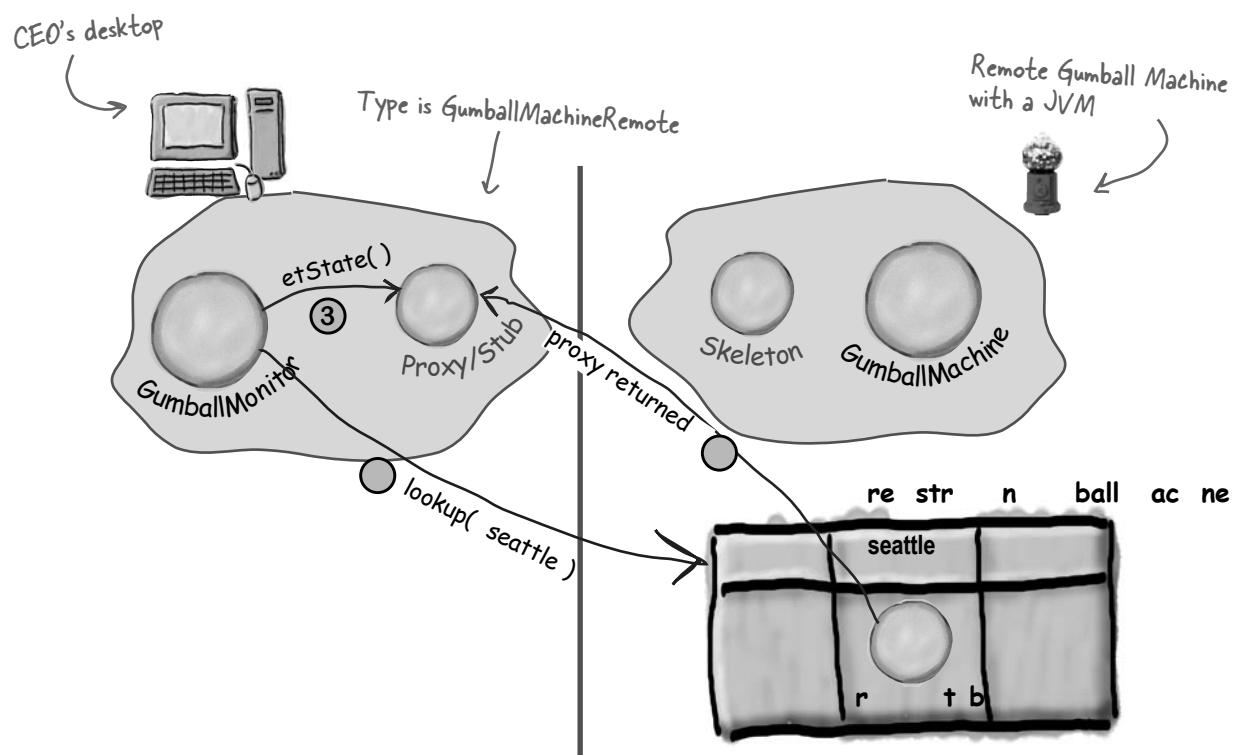


By invoking methods on the proxy, a remote call is made across the wire and a String, an integer and a State object are returned. Because we are using a proxy, the GumballMonitor doesn't know, or care, that calls are remote (other than having to worry about remote exceptions).

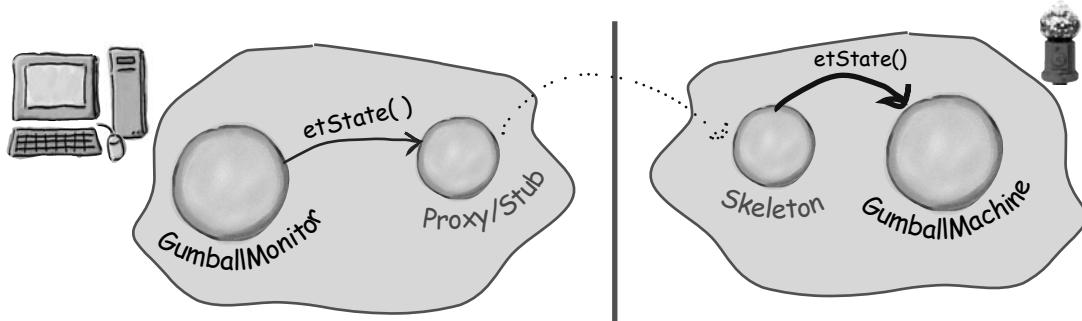
you are here ▶



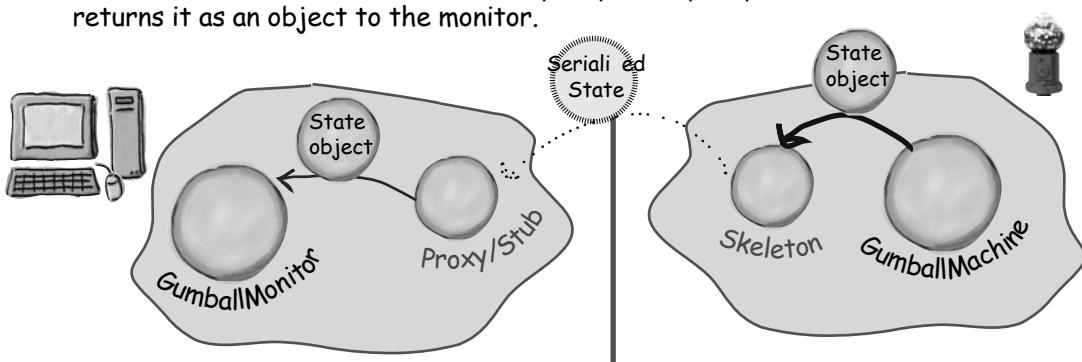
- 1 The CEO runs the monitor, which first maps the proxies to the remote gumball machines and then calls `getCount()` on each one (along with `getCount()` and `getLocation()`).



- 2 `getstate()` is called on the proxy, which forwards the call to the remote service. The skeleton receives the request and then forwards it to the gumball machine.



- 3 GumballMachine returns the state to the skeleton, which serializes it and transfers it back over the wire to the proxy. The proxy deserializes it and returns it as an object to the monitor.



The monitor hasn't changed at all, except it knows it may encounter remote exceptions. It also uses the `GumballMachineRemote` interface rather than a concrete implementation.

Likewise, the `GumballMachine` implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

We also have a small bit of code to register and locate stubs using the RMI registry. But no matter what, if we were writing something to work over the Internet, we'd need some kind of locator service.

## The Proxy Pattern defined

We've already put a lot of pages behind us in this chapter; as you can see, explaining the remote Proxy is quite involved. Despite that, you'll see that the definition and class diagram for the Proxy Pattern is actually fairly straightforward. Note that remote Proxy is one implementation of the general Proxy Pattern; there are actually quite a few variations of the pattern, and we'll talk about them later. For now, let's get the details of the general pattern down.

Here's the Proxy Pattern definition

**he ro atter** provides a surrogate or placeholder for another object to control access to it.

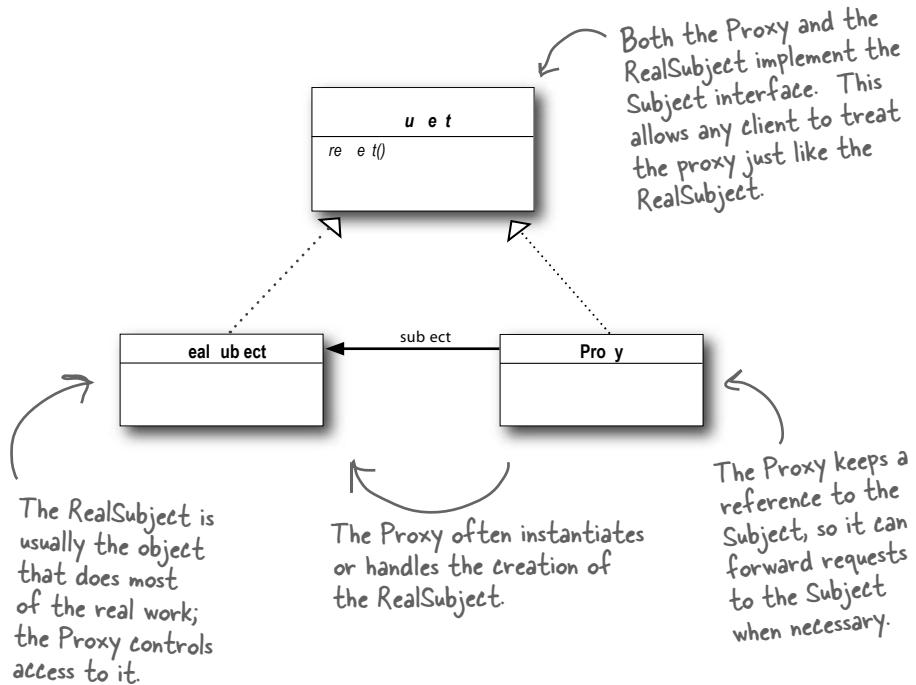
Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a representative for another object.

But what about a proxy controlling access? That sounds a little strange. So worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy controls access. We're going to talk more about this later, but for now here are a few ways proxies control access:

- As we know, a remote proxy controls access to a remote object.
- A virtual proxy controls access to a resource that is expensive to create.
- A protection proxy controls access to a resource based on access rights.

Now that you've got the gist of the general pattern, check out the class diagram...

**Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.**



Let's step through the diagram...

First we have a `Subject`, which provides an interface for the `RealSubject` and the `Proxy`. By implementing the same interface, the `Proxy` can be substituted for the `RealSubject` anywhere it occurs.

The `RealSubject` is the object that does the real work. It's the object that the `Proxy` represents and controls access to.

The `Proxy` holds a reference to the `RealSubject`. In some cases, the `Proxy` may be responsible for creating and destroying the `RealSubject`. Clients interact with the `RealSubject` through the `Proxy`. Because the `Proxy` and `RealSubject` implement the same interface (`Subject`), the `Proxy` can be substituted anywhere the `Subject` can be used. The `Proxy` also controls access to the `RealSubject`; this control may be needed if the `Subject` is running on a remote machine, if the `Subject` is expensive to create in some way or if access to the `Subject` needs to be protected in some way.

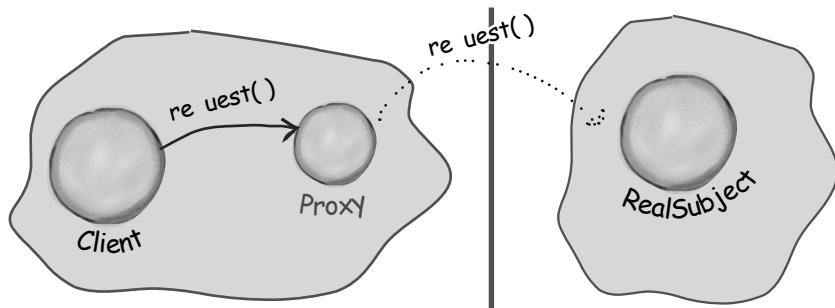
Now that you understand the general pattern, let's look at some other ways of using `Proxy` beyond the remote `Proxy`...

# Get ready for Virtual Proxy

Okay, so far you've seen the definition of the Proxy Pattern and you've taken a look at one specific example, the *emote proxy*. Now we're going to take a look at a different type of proxy, the *remote proxy*. As you'll discover, the Proxy Pattern can manifest itself in many forms, yet all the forms follow roughly the general proxy design. Why so many forms? Because the proxy pattern can be applied to a lot of different use cases. Let's check out the virtual proxy and compare it to the remote proxy.

## Remote Proxy

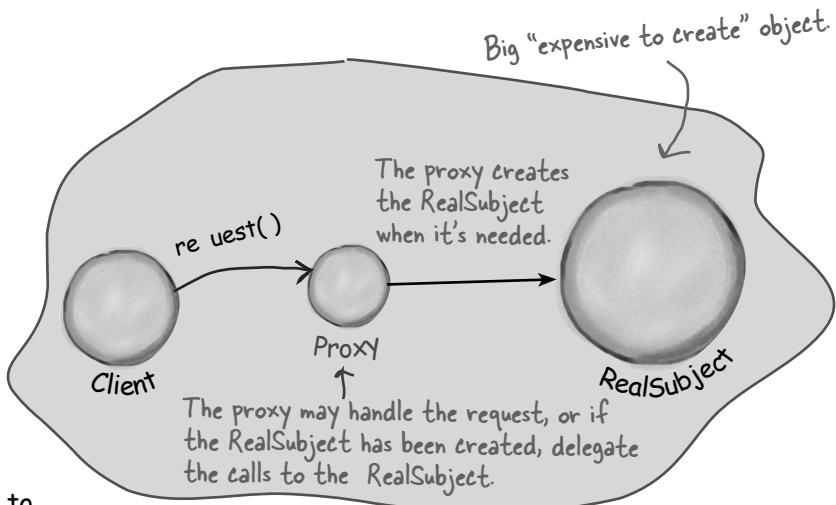
With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.



We know this diagram pretty well by now...

## Virtual Proxy

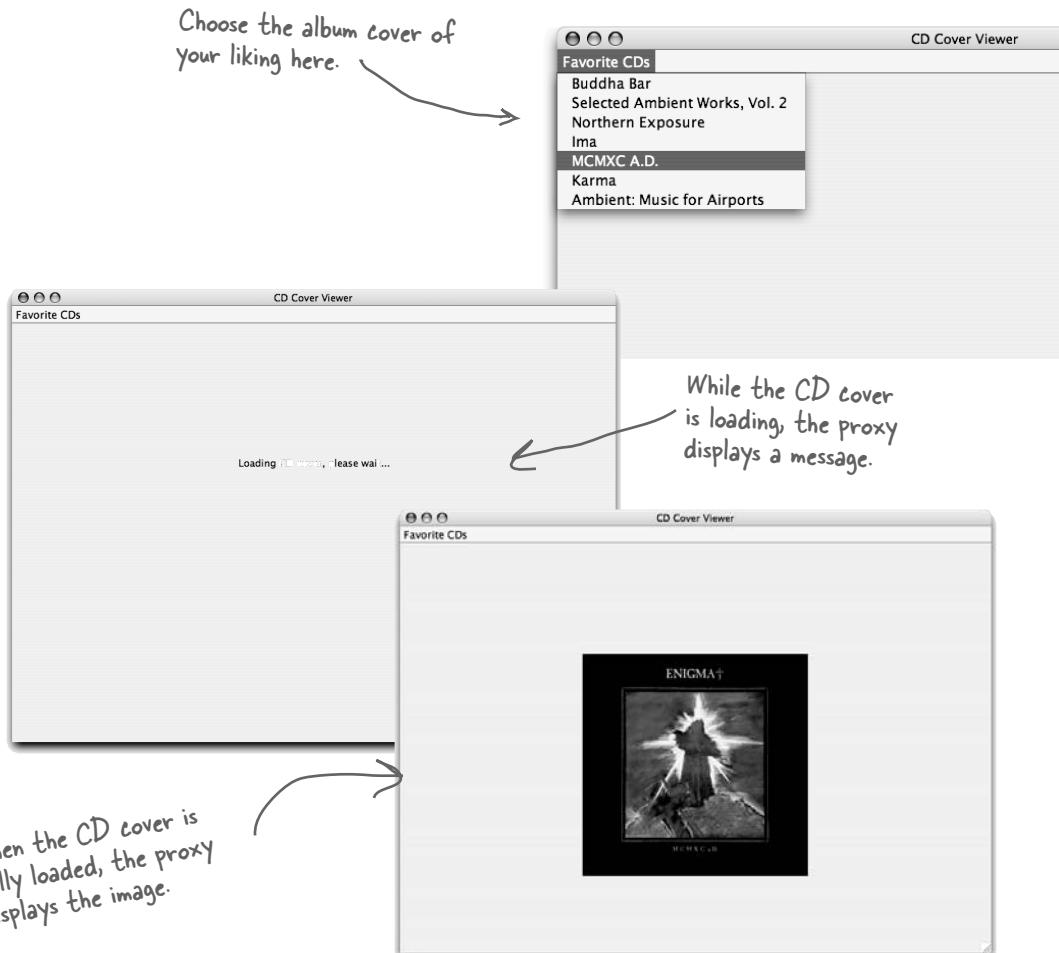
Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.



## Displaying CD covers

Let's say you want to write an application that displays your favorite compact disc covers. You might create a menu of the CD titles and then retrieve the images from an online service like [ma on.com](#). If you're using wing, you might create an icon and ask it to load the image from the network. The only problem is, depending on the network load and the bandwidth of your connection, retrieving a CD cover might take a little time, so your application should display something while you are waiting for the image to load. We also don't want to hang up the entire application while it's waiting on the image. Once the image is loaded, the message should go away and you should see the image.

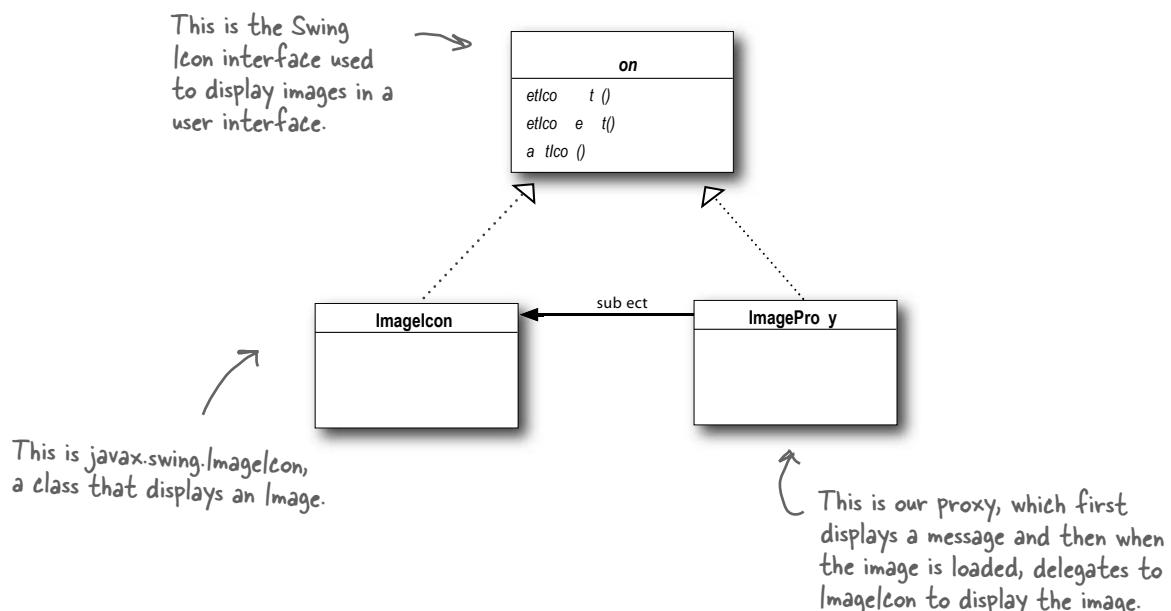
An easy way to achieve this is through a virtual proxy. The virtual proxy can stand in place of the icon, manage the background loading, and before the image is fully retrieved from the network, display "Loading CD cover, please wait...". Once the image is loaded, the proxy delegates the display to the icon.



## Designing the CD cover Virtual Proxy

Before writing the code for the CD Cover Viewer, let's look at the class diagram.

You'll see this looks just like our Remote Proxy class diagram, but here the proxy is used to hide an object that is expensive to create (because we need to retrieve the data for the icon over the network) as opposed to an object that actually lives somewhere else on the network.



### How ImageProxy is going to work:

- ① ImageProxy first creates an ImageIcon and starts loading it from a network**
- ② While the bytes of the image are being retrieved, ImageProxy displays loading CD cover, please wait**
- ③ When the image is fully loaded, ImageProxy delegates all method calls to the ImageIcon, including paintIcon(), getWidth() and getHeight()**
- If the user requests a new image, we'll create a new proxy and start the process over**

# Writing the Image Proxy

```

class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;
                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "CD Cover");
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}

```

The ImageProxy implements the Icon interface.

The imageIcon is the REAL icon that we eventually want to display when it's loaded.

We pass the URL of the image into the constructor. This is the image we need to display once it's loaded!

We return a default width and height until the imageIcon is loaded; then we turn it over to the imageIcon.

Here's where things get interesting. This code paints the icon on the screen (by delegating to the imageIcon). However, if we don't have a fully created ImageIcon, then we create one. Let's look at this closer on the next page...

on
etico t()
etico e t()
a tico()



# Code Up Close

— This method is called when it's time to paint the icon on the screen.

```
public void paintIcon(final Component c, Graphics g, int x, int y) {
    if (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } else {
        g.drawString("Loading CD cover, please wait...", x+300, y+190);
        if (!retrieving) {
            retrieving = true;
            retrievalThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        ImageIcon = new ImageIcon(imageURL, "CD Cover");
                        c.repaint();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            retrievalThread.start();
        }
    }
}
```

If we've got an icon already, we go ahead and tell it to paint itself.

Otherwise we display the "loading" message.

Here's where we load the REAL icon image. Note that the image loading with ImageIcon is synchronous: the ImageIcon constructor doesn't return until the image is loaded. That doesn't give us much of a chance to do screen updates and have our message displayed, so we're going to do this asynchronously. See the "Code Way Up Close" on the next page for more...



## Code Up Close

If we aren't already trying to retrieve the image...

```

if (!retrieving) {
    retrieving = true;
}

retrievalThread = new Thread(new Runnable() {
    public void run() {
        try {
            ImageIcon = new ImageIcon(imageURL, "CD Cover");
            c.repaint();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
});
retrievalThread.start();
}

```

...then it's time to start retrieving it (in case you were wondering, only one thread calls paint, so we should be okay here in terms of thread safety).

We don't want to hang up the entire user interface, so we're going to use another thread to retrieve the image.

In our thread we instantiate the ImageIcon object. Its constructor will not return until the image is loaded.

When we have the image, we tell Swing that we need to be repainted.

So, the next time the display is painted after the ImageIcon is instantiated, the paintIcon method will paint the image, not the loading message.

# esign Pu le

The ImageProxy class appears to have two states that are controlled by a conditional statement. Can you think of another pattern that might clean up the code? How would you redesign ImageProxy?

```
class ImageProxy implements Icon {
    // instance variables & constructor here

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            // more code here
        }
    }
}
```

The diagram consists of three handwritten annotations "Two states" with arrows pointing to each of the three conditional blocks in the code: one for the width check, one for the height check, and one for the main paintIcon method's null check.

# Testing the CD Cover Viewer



Now it's time to test out this fancy new virtual proxy. Behind the scenes we've been baking up a new ImageProxyTestDrive that sets up the window, creates a frame, installs the menus and creates our proxy.

We don't go through all that code in great detail here, but you can always grab the source code and have a look, or check it out at the end of the chapter where we list all the source code for the Virtual Proxy.

Here's a partial view of the test drive code:

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{
        // set up frame and menus
        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Finally we add the proxy to the frame so it can be displayed.

Here we create an image proxy and set it to an initial URL. Whenever you choose a selection from the CD menu, you'll get a new image proxy.

Next we wrap our proxy in a component so it can be added to the frame. The component will take care of the proxy's width, height and similar details.

Now let's run the test drive:

```
File Edit Window Help J tSomeO T eCD T atGotU T ro T i Boo
% java ImageProxyTestDrive
```

Running ImageProxyTestDrive should give you a window like this.

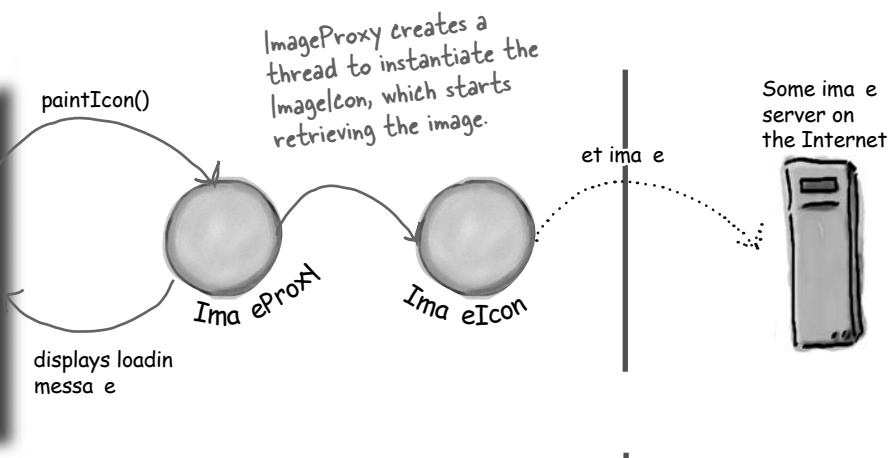
## Things to try...

- ① Use the menu to load different CD covers watch the proxy display loading until the image has arrived**
- ② Resize the window as the loading message is displayed Notice that the proxy is handling the loading without hanging up the main window**
- ③ Add your own favorite CDs to the ImageProxyTestDrive**

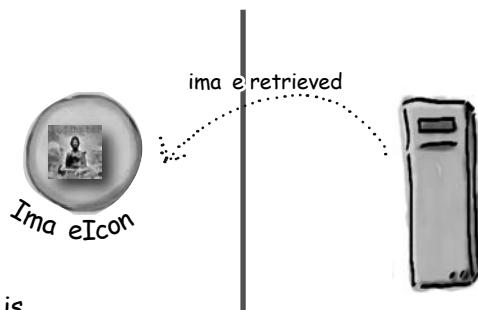


## What did we do?

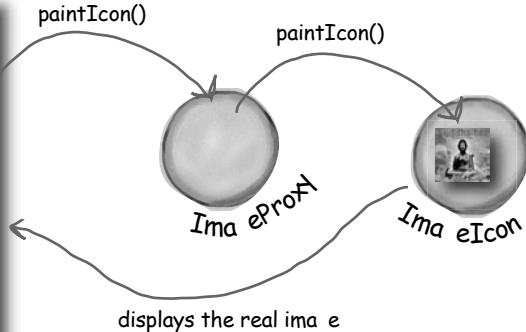
- ① We created an `ImageProxy` for the display. The `paintIcon()` method is called and `ImageProxy` sets off a thread to retrieve the image and create the `ImageIcon`.



- ② At some point the image is returned and the `ImageIcon` is fully instantiated.



- ③ After the `ImageIcon` is created, the next time `paintIcon()` is called, the proxy delegates to the `ImageIcon`.



## there are no Dumb Questions

**Q:** The remote Proxy and virtual Proxy seem so different to me are they really OOP patterns?

**A:** You'll find a lot of variants of the proxy pattern in the real world what they all have in common is that they intercept a method invocation that the client is making on the subject. This level of indirection allows us to do many things, including dispatching requests to a remote subject, providing a representative for an expensive object as it is created, or, as you'll see, providing some level of protection that can determine which clients should be calling which methods. That's just the beginning the general proxy pattern can be applied in many different ways, and we'll cover some of the other ways at the end of the chapter.

**Q:** ImageProxy seems just like a decorator to me. I mean, we are basically wrapping one object with another and then delegating the calls to the ImageIcon. What am I missing?

**A:** Sometimes proxy and Decorator look very similar, but their purposes are different: a decorator adds behavior to a class, while a proxy controls access to it. You might say, isn't the loading message adding behavior? In some

ways it is however, more important, the Image proxy is controlling access to an ImageIcon. How does it control access well, think about it this way: the proxy is decoupling the client from the ImageIcon. If they were coupled the client would have to wait until each image is retrieved before it could paint its entire interface. The proxy controls access to the ImageIcon so that before it is fully created, the proxy provides another on-screen representation. Once the ImageIcon is created the proxy allows access.

**Q:** How do I make clients use the proxy rather than the real subject?

**A:** Good question. One common technique is to provide a factory that instantiates and returns the subject. Because this happens in a factory method we can then wrap the subject with a proxy before returning it. The client never knows or cares that it's using a proxy instead of the real thing.

**Q:** I noticed in the ImageProxy example, you always create a new ImageIcon to get the image, even if the image has already been retrieved. Could you implement something similar to the ImageProxy that caches past retrievals?

**A:** You are talking about a specialized form of a Virtual proxy called a caching proxy. Caching proxy maintains a cache of previously created objects and when a request is made it returns cached object, if possible.

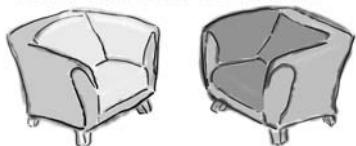
We're going to look at several other variants of the proxy pattern at the end of the chapter.

**Q:** I see how Decorator and Proxy relate, but what about Chapter 10's Adapter seems very similar as well.

**A:** Both proxy and adapter sit in front of other objects and forward requests to them. Remember that adapter changes the interface of the objects it adapts, while the proxy implements the same interface.

There is one additional similarity that relates to the protection proxy. Protection proxy may allow or disallow a client access to particular methods in an object based on the role of the client. In this way a protection proxy may only provide a partial interface to a client, which is quite similar to some adapters. We are going to take a look at protection proxy in a few pages.

## Fireside Chats



Tonight's talk: **ro y and Decorator get intent onal.**

### ro y

Hello, ecorator. I presume you're here because people sometimes get us confused?

I'm copying your ideas? Please. I control access to objects. You just decorate them. My job is so much more important than yours it's just not even funny.

Fine, so maybe you're not entirely frivolous... but I still don't get why you think I'm copying all your ideas. I'm all about representing my subjects, not decorating them.

I don't think you get it, ecorator. I stand in for my subjects; I don't just add behavior. Clients use me as a surrogate of a real object, because I can protect them from unwanted access, or keep their GUIs from hanging up while they're waiting for big objects to load, or hide the fact that their subjects are running on remote machines. I'd say that's a very different intent from yours.

*ha ter*

### Decorator

Well, I think the reason people get us confused is that you go around pretending to be an entirely different pattern, when in fact, you're just a ecorator in disguise. I really don't think you should be copying all my ideas.

Just decorate? You think decorating is some frivolous unimportant pattern? Let me tell you buddy, add e i r. That's the most important thing about objects: what they

You can call it representation but if it looks like a duck and walks like a duck... I mean, just look at your virtual Proxy; it's just another way of adding behavior to do something while some big expensive object is loading, and your remote Proxy is a way of talking to remote objects so your clients don't have to bother with that themselves. It's all about behavior, just like I said.

Call it what you want. Implement the same interface as the objects I wrap; so do you.

## ro y

kay, let's review that statement. You wrap an object. While sometimes we informally say a proxy wraps its subject, that's not really an accurate term.

Hink about a remote proxy... what object am I wrapping? The object I'm representing and controlling access to lives on another machine. Let's see you do that.

ure, okay, take a virtual proxy... think about the C viewer example. When the client first uses me as a proxy the subject doesn't even exist so what am I wrapping there?

I never knew decorators were so dumb. Of course sometimes create objects, how do you think a virtual proxy gets its subject? Kay, you just pointed out a big difference between us. We both know decorators only add window dressing; they never get to instantiate anything.

Hey, after this conversation I'm convinced you're just a dumb proxy.

Very seldom will you ever see a proxy get into wrapping a subject multiple times; in fact, if you're wrapping something times, you better go back re-examine your design.

## Decorator

Huh yeah? Why not?

Kay, but we all know remote proxies are kinda weird. Got a second example? Doubt it.

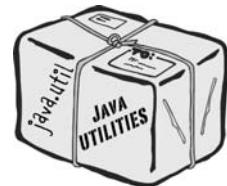
Huh huh, and the next thing you'll be saying is that you actually get to create objects.

Huh yeah? Instantiate this.

Dumb proxy? I'd like to see you recursively wrap an object with decorators and keep your head straight at the same time.

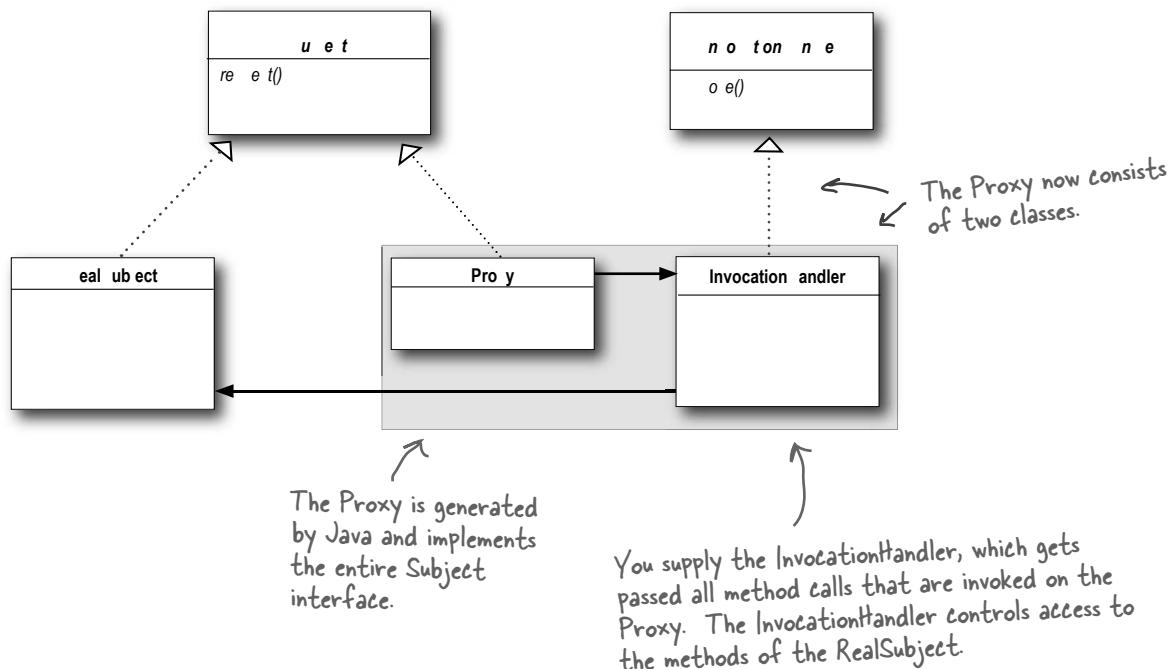
Just like a proxy, acting all real when in fact you just stand in for the objects doing the real work. You know, I actually feel sorry for you.

# Using the ava API's Proxy to create a protection proxy



ava's got its own proxy support right in the `java.lang.reflect` package. With this package, ava lets you create a proxy class `Proxy` that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this ava technology as a `dynamic proxy`.

We're going to use ava's dynamic proxy to create our new proxy implementation (a protection proxy), but before we do that, let's quickly look at a class diagram that shows how dynamic proxies are put together. Like most things in the real world, it differs slightly from the classic definition of the pattern



Because ava creates the `Proxy` class `Proxy`, you need a way to tell the `Proxy` class what to do. You can't put that code into the `Proxy` class like we did before, because you're not implementing one directly. So, if you can't put this code in the `Proxy` class, where do you put it? In an `InvocationHandler`. The job of the `InvocationHandler` is to respond to any method calls on the `Proxy`. Think of the `InvocationHandler` as the object the `Proxy` asks to do all the real work after it's received the method calls.

Okay, let's step through how to use the dynamic proxy...

# Matchmaking in Objectville



every town needs a matchmaking service, right? you've risen to the task and implemented a dating service for Objectville. you've also tried to be innovative by including a Hot or Not feature in the service where participants can rate each other you figure this keeps your customers engaged and looking through possible matches; it also makes things a lot more fun.

our service revolves around a Person bean that allows you to set and get information about a person

```
This is the interface; we'll
get to the implementation
in just a sec... ↴

public interface PersonBean {
    String getName();
    String getGender();
    String getInterests();
    int getHotOrNotRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setHotOrNotRating(int rating); ↵
}

We can also set the same
information through the
respective method calls. ↗
```

Here we can get information about the person's name, gender, interests and HotOrNot rating (1-1).

setHotOrNotRating() takes an integer and adds it to the running average for this person.

Now let's check out the implementation...

you are here ▶

# The PersonBean implementation

The PersonBeanImpl implements the PersonBean interface

```
public class PersonBeanImpl implements PersonBean {  
    String name;  
    String gender;  
    String interests;  
    int rating;  
    int ratingCount = 0;  
  
    public String getName() {  
        return name;  
    }  
  
    public String getGender() {  
        return gender;  
    }  
  
    public String getInterests() {  
        return interests;  
    }  
  
    public int getHotOrNotRating() {  
        if (ratingCount == 0) return 0;  
        return (rating/ratingCount);  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void setGender(String gender) {  
        this.gender = gender;  
    }  
  
    public void setInterests(String interests) {  
        this.interests = interests;  
    }  
  
    public void setHotOrNotRating(int rating) {  
        this.rating += rating;  
        ratingCount++;  
    }  
}
```

The instance variables.

All the getter methods; they each return the appropriate instance variable...

...except for `getHotOrNotRating()`, which computes the average of the ratings by dividing the ratings by the `ratingCount`.

And here's all the setter methods, which set the corresponding instance variable.

Finally, the `setHotOrNotRating()` method increments the total `ratingCount` and adds the rating to the running total.

I wasn't very successful finding dates. Then I noticed someone had changed my interests. I also noticed that a lot of people are bumping up their HotOrNot scores by giving themselves high ratings. You shouldn't be able to change someone else's interests or give yourself a rating!



Elroy

While we suspect other factors may be keeping Elroy from getting dates, he is right: you shouldn't be able to vote for yourself or to change another customer's data. The way our PersonBean is defined, any client can call any of the methods.

This is a perfect example of where we might be able to use a Protection Proxy. What's a Protection Proxy? It's a proxy that controls access to an object based on access rights. For instance, if we had an employee object, a protection proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like `setSalary()`), and a human resources employee to call any method on the object.

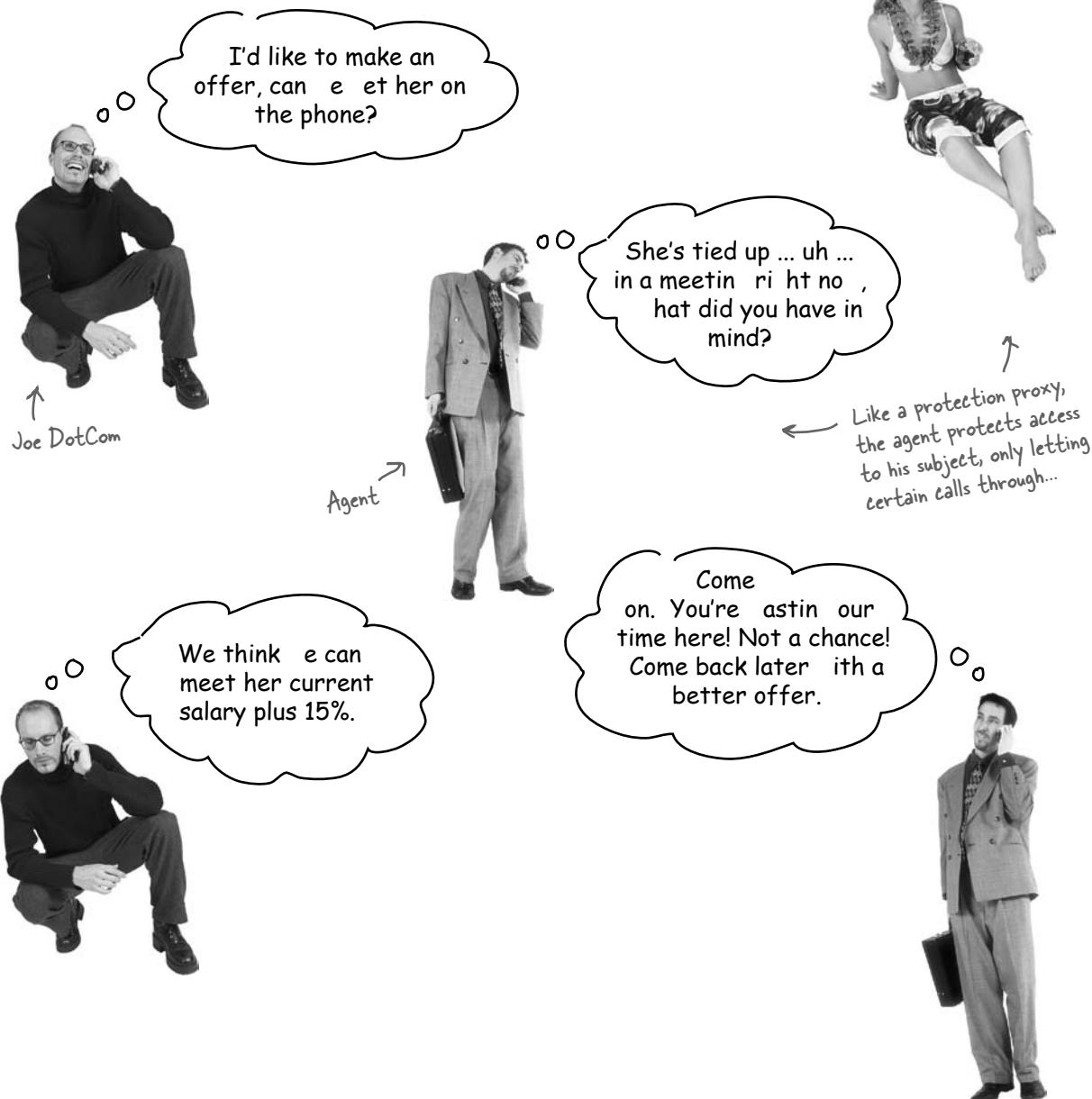
In our dating service we want to make sure that a customer can set his own information while preventing others from altering it. We also want to allow just the opposite with the Hot or Not ratings: we want the other customers to be able to set the rating, but not that particular customer. We also have a number of getter methods in the PersonBean, and because none of these return private information, any customer should be able to call them.

*you are here ▶*



## Five minute drama: protecting subjects

The Internet bubble seems a distant memory; those were the days when all you needed to do to find a better, higher-paying job was to walk across the street. Even agents for software developers were in vogue...



# Big Picture: creating a Dynamic Proxy for the PersonBean

We have a couple of problems to fix: customers shouldn't be changing their own HotSpot rating and customers shouldn't be able to change other customers' personal information. To fix these problems we're going to create two proxies: one for accessing your own PersonBean object and one for accessing another customer's PersonBean object. That way, the proxies can control what requests can be made in each circumstance.

To create these proxies we're going to use the Java API's dynamic proxy that you saw a few pages back. Java will create two proxies for us; all we need to do is supply the handlers that know what to do when a method is invoked on the proxy.

## Step one

### Create two invocation handlers

InvocationHandlers implement the behavior of the proxy. As you'll see Java will take care of creating the actual proxy class and object, we just need to supply a handler that knows what to do when a method is called on it.

## Step two

### Write the code that creates the dynamic proxies.

We need to write a little bit of code to generate the proxy class and instantiate it. We'll step through this code in just a bit.

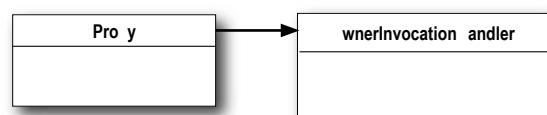
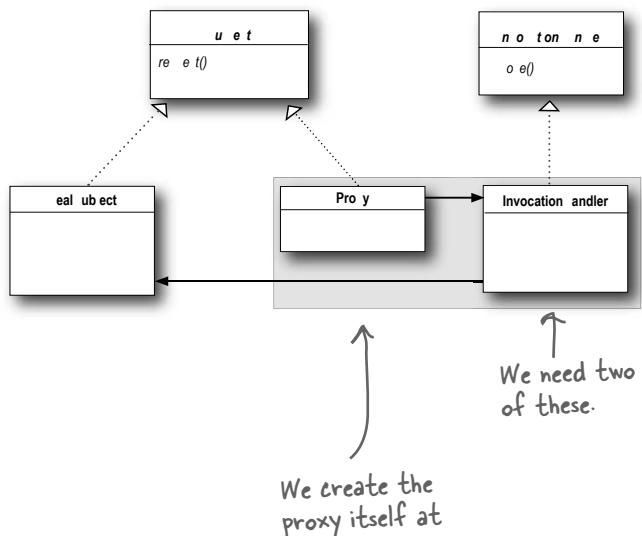
## Step three

### Wrap any PersonBean object with the appropriate proxy.

When we need to use a PersonBean object, either it's the object of the customer himself (in that case, we'll call him the owner), or it's another user of the service that the customer is checking out (in that case we'll call him non-owner).

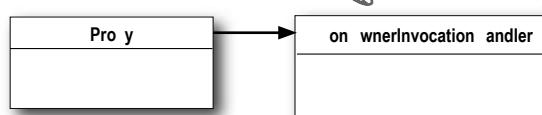
In either case, we create the appropriate proxy for the PersonBean.

Remember this diagram from a few pages back...



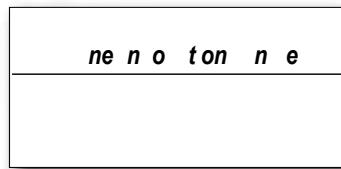
When a customer is viewing his own bean

When a customer is viewing someone else's bean



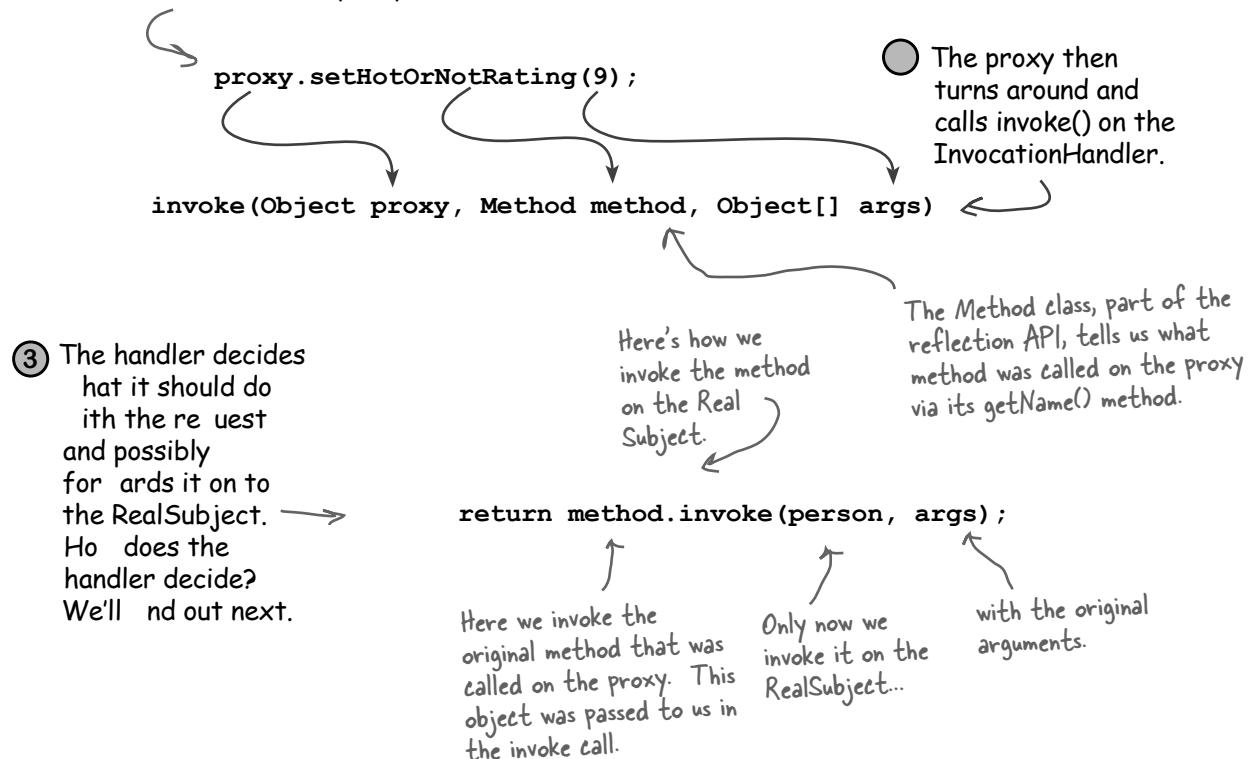
## Step one: creating Invocation Handlers

We know we need to write two invocation handlers, one for the owner and one for the non owner. But what are invocation handlers? Here's the way to think about them when a method call is made on the proxy, the proxy forwards that call to your invocation handler, but it by calling the invocation handler's corresponding method. So, what does it call? Have a look at the InvocationHandler interface



here's only one method, invoke(), and no matter what methods get called on the proxy, the invoke() method is what gets called on the handler. Let's see how this works

- Let's say the setHotOrNotRating () method is called on the proxy.



## Creating Invocation Handlers continued...

When invoke() is called by the proxy, how do you know what to do with the call?

Typically, you'll examine the method that was called on the proxy and make decisions based on the method's name and possibly its arguments. Let's implement the OwnerInvocationHandler to see how this works.

```

InvocationHandler is part of the java.lang.reflect
package, so we need to import it.
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public OwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}

If any other method is called,
we're just going to return null
rather than take a chance.

```

All invocation handlers implement the InvocationHandler interface.

We've passed the Real Subject in the constructor and we keep a reference to it.

Here's the invoke method that gets called every time a method is invoked on the proxy.

If the method is a getter, we go ahead and invoke it on the real subject.

Otherwise, if it is the setHotOrNotRating() method we disallow it by throwing a IllegalAccessException.

Because we are the owner any other set method is fine and we go ahead and invoke it on the real subject.

This will happen if the real subject throws an exception.

## *reate your o n in o ation hand er*



The on wner nvocationHandler works just like the wner nvocationHandler except that it *is* calls to setHot rot ating() and it *is* calls to any other set method. Go ahead and write this handler yourself

## Step two: creating the Proxy class and instantiating the Proxy object

Now, all we have left is to dynamically create the proxy class and instantiate the proxy object. Let's start by writing a method that takes a PersonBean and knows how to create an owner proxy for it. That is, we're going to create the kind of proxy that forwards its method calls to the OwnerInvocationHandler. Here's the code

```
This method takes a person object (the real
subject) and returns a proxy for it. Because the
proxy has the same interface as the subject, we
return a PersonBean.
↓
PersonBean getOwnerProxy(PersonBean person) {
    return (PersonBean) Proxy.newProxyInstance(
        person.getClass().getClassLoader(),
        person.getClass().getInterfaces(),
        new OwnerInvocationHandler(person));
}

This code creates the
proxy. Now this is some
mighty ugly code, so let's
step through it carefully.
↓
To create a proxy we use
the static newProxyInstance
method on the Proxy class...
← We pass it the classloader
for our subject...
...and the set of interfaces the
proxy needs to implement...
...and an invocation handler, in this
case our OwnerInvocationHandler.

We pass the real subject into the constructor
of the invocation handler. If you look back
two pages you'll see this is how the handler gets
access to the real subject.
```



While it is a little complicated, there is nothing to creating a dynamic proxy. Why don't you write `getNonOwnerProxy()`, which returns a proxy for the NonOwnerInvocationHandler:

Take it further: can you write one method `getProxy()` that takes a handler and a person and return a proxy that extends that handler?

## Testing the matchmaking service

Let's give the matchmaking service a test run and see how it controls access to the setter methods based on the proxy that is used.

```
public class MatchMakingTestDrive {  
    // instance variables here  
  
    public static void main(String[] args) {  
        MatchMakingTestDrive test = new MatchMakingTestDrive();  
        test.drive();  
    }  
  
    public MatchMakingTestDrive() {  
        initializeDatabase();  
    }  
    public void drive() {  
        PersonBean joe = getPersonFromDatabase("Joe Javabean");  
        PersonBean ownerProxy = getOwnerProxy(joe);  
        System.out.println("Name is " + ownerProxy.getName());  
        ownerProxy.setInterests("bowling, Go");  
        System.out.println("Interests set from owner proxy");  
        try {  
            ownerProxy.setHotOrNotRating(10);  
        } catch (Exception e) {  
            System.out.println("Can't set rating from owner proxy");  
        }  
        System.out.println("Rating is " + ownerProxy.getHotOrNotRating());  
        this shouldn't work!  
  
        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);  
        System.out.println("Name is " + nonOwnerProxy.getName());  
        try {  
            nonOwnerProxy.setInterests("bowling, Go");  
        } catch (Exception e) {  
            System.out.println("Can't set interests from non owner proxy");  
        }  
        nonOwnerProxy.setHotOrNotRating(3);  
        System.out.println("Rating set from non owner proxy");  
        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());  
        this shouldn't work!  
    }  
  
    // other methods like getOwnerProxy and getNonOwnerProxy here  
}
```

Main just creates the test drive and calls its `drive()` method to get things going.

The constructor initializes our DB of people in the matchmaking service.

Let's retrieve a person from the DB  
...and create an owner proxy.

Call a getter and then a setter  
and then try to change the rating.

Now create a non-owner proxy  
...and call a getter  
followed by a setter

This shouldn't work!

Then try to set the rating  
This should work!

## technique

File Edit Window Help Born2BDynamic

```
% java MatchMakingTestDrive
```

```
Name is Joe Javabean
```

```
Interests set from owner proxy
```

```
Can't set rating from owner proxy
```

```
Rating is 7
```

Our Owner proxy  
allows getting and  
setting, except for  
the HotOrNot rating.

```
Name is Joe Javabean
```

```
Can't set interests from non owner proxy
```

```
Rating set from non owner proxy
```

Our NonOwner proxy  
allows getting only, but  
also allows calls to set the  
HotOrNot rating.

```
Rating is 5
```

```
%
```

The new rating is the average of the previous rating,  
and the value set by the nonowner proxy, .

*you are here ▶*

## there are no Dumb Questions

**Q:** So what exactly is the dynamic aspect of dynamic proxies? Is it that I'm instantiating the proxy and setting it to a handler at runtime?

**A:** So, the proxy is dynamic because its class is created at runtime. Think about it: before our code runs there is no proxy class; it is created on demand from the set of interfaces you pass it.

**Q:** Why InvocationHandler seems like a very strange proxy, it doesn't implement any of the methods of the class it's proxying.

**A:** That is because the InvocationHandler isn't a proxy; it is a class that the proxy dispatches to for handling method calls. The proxy itself is created dynamically at runtime by the static `Proxy.newProxyInstance` method.

**Q:** Is there any way to tell if a class is a Proxy class?

**A:** Yes. The `Proxy` class has a static method called `isProxyClass`. Calling this method with a class will return true if the class is a dynamic proxy class. Other than that, the proxy class will act like an other class that implements a particular set of interfaces.

**Q:** Are there any restrictions on the types of interfaces I can pass into `newProxyInstance`?

**A:** Yes, there are a few. First, it is worth pointing out that we always pass new proxy instances; an array of interfaces or multiple interfaces are allowed, no classes. The major restrictions are that all non-public interfaces need to be from the same package. You also can't have interfaces with clashing method names; that is, two interfaces with a method with the same signature. There are a few other minor nuances as well, so at some point you should take a look at the fine print on dynamic proxies in the JavaDoc.

**Q:** Why are you using skeletons? I thought we got rid of those back in Java . . .

**A:** You're right; we don't need to actually generate skeletons. In Java . . ., the MI runtime can dispatch the client calls directly to the remote service using reflection. But we like to show the skeleton, because conceptually it helps you to understand that there is something under the covers that's making that communication between the client stub and the remote service happen.

**Q:** I heard that in Java . . ., I don't even need to generate stubs anymore either. Is that true?

**A:** It sure is. In Java . . ., MI and Dynamic proxy got together and now stubs are generated dynamically using Dynamic proxy. The remote object's stub is a `java.lang.reflect.Proxy` instance with an invocation handler that is automatically generated to handle all the details of getting the local method calls from the client to the remote object. So, now you don't have to use rmic at all; everything you need to get a client talking to a remote object is handled for you behind the scenes.



Match each pattern with its description

### **Pattern**

### **Description**

De orator

ra s another ob e t  
an ro i es a i erent  
inter a e to it

a a e

ra s another ob e t  
an ro i es a itiona  
beha ior or it

ro

ra s another ob e t to  
ontro a ess to it

A a ter

ra s a bun h o  
ob e ts to sim i their  
inter a e

*you are here ▶*

# The Proxy oo

Welcome to the objectville oo

You now know about the remote, virtual and protection profiles, but out in the wild you're going to see lots of mutations of this pattern.

Over here in the Proxy corner of the oo we've got a nice collection of wild proxy patterns that we've captured for your study.

Your job isn't done; we are sure you're going to see more variations of this pattern in the real world, so give us a hand in cataloging more profiles. Let's take a look at the existing collection



**re all r**  
controls access to a  
set of network  
resources, protectin  
the subject from bad clients.

Habitat: often seen in the location  
of corporate firewall systems.

Help find a habitat

---

---

---

**art e erence r**  
provides additional actions  
whenever a subject is  
referenced, such as countin  
the number of references to  
an object.



**Cac n r** provides  
temporary storage for  
results of operations  
that are expensive. It  
can also allo multiple clients to share  
the results to reduce computation or  
network latency.

Habitat: often seen in web server proxies as well  
as content management and publishing systems.

ha ter

**Concurrent**  
provides safe access to  
a subject from multiple  
threads.



Seen hanging around JavaSpaces, where it controls synchronized access to an underlying set of objects in a distributed environment.

Help find a habitat

---

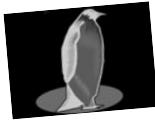
---

---

**Complex**  
hides the complexity of  
and controls access to a  
complex set of classes.  
This is sometimes called  
the Facade Proxy for obvious reasons.



The Complexity Hiding Proxy differs from the Facade Pattern in that the proxy controls access, while the Facade Pattern just provides an alternative interface.



**Copy-on-Writer**  
controls the copying of  
an object by deferring  
the copying of an  
object until it is required by  
a client. This is a variant of  
the Virtual Proxy.

Habitat: seen in the vicinity of the Java's `CopyOnWriteArrayList`.

Field Notes: please add your observations of other proxies in the wild here:

---

---

---

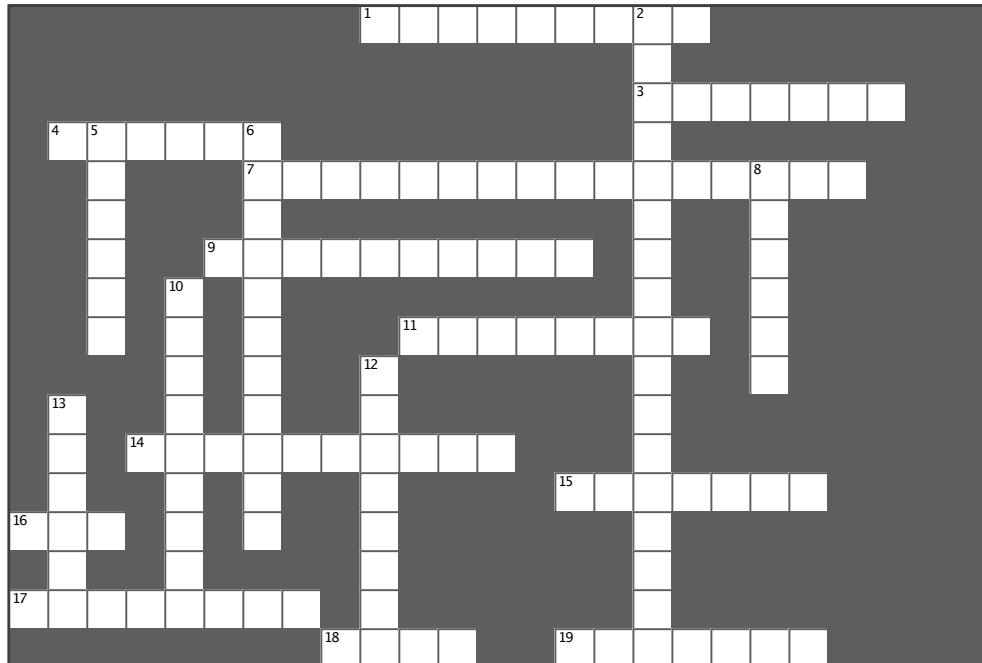
---

---

---

---

---

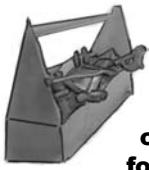


#'\$\$

%#3 FMSN MD@B0R0 1 AMTOPBGNU?VQB RUM UMBQ  
 '#8 FMMW@?RGR@LBQG @MPVNCQBC M@CARQ  
 (#< C RMM MLC MD@CQC RMJC?PL 9 6 4  
 +#9 CK MRC >>>>> U?QSQCB RM@CL NUCK CLR  
 REC ESK @JK ?AFGC K ML@MP RUM UMBQ  
 - #: MDR?PC BCTCMNC?ECLR?Q@GE REC@I GB  
 MDNMMW  
 %#4L 9 6 4 REC M@CARRE?RR1 CQREC LCRJMR  
 POOSOQRML REC QOPTAC QBC  
 %#8 FMMW@?RNFMRCARQK CRMB A?JQ@MK  
 SL?SREM@OB A?JCRQ  
 %#/ >>>>> NMWVAJ@QQ@AF?RC?RDB ?RSLRC C  
 %#8 J@AC RMJC?PL ?@SRREC K ?LWMNMWT?RGLRQ  
 %#0 MK K MLJSQCB NMW@MPUC@OPT@CQ RUM  
 UMBQ  
 %#4L 9 6 4 REC NMWVAQ@JCB REC  
 %#: FC 0 1 TGUOPSCQB REC@I GB MDNMMW

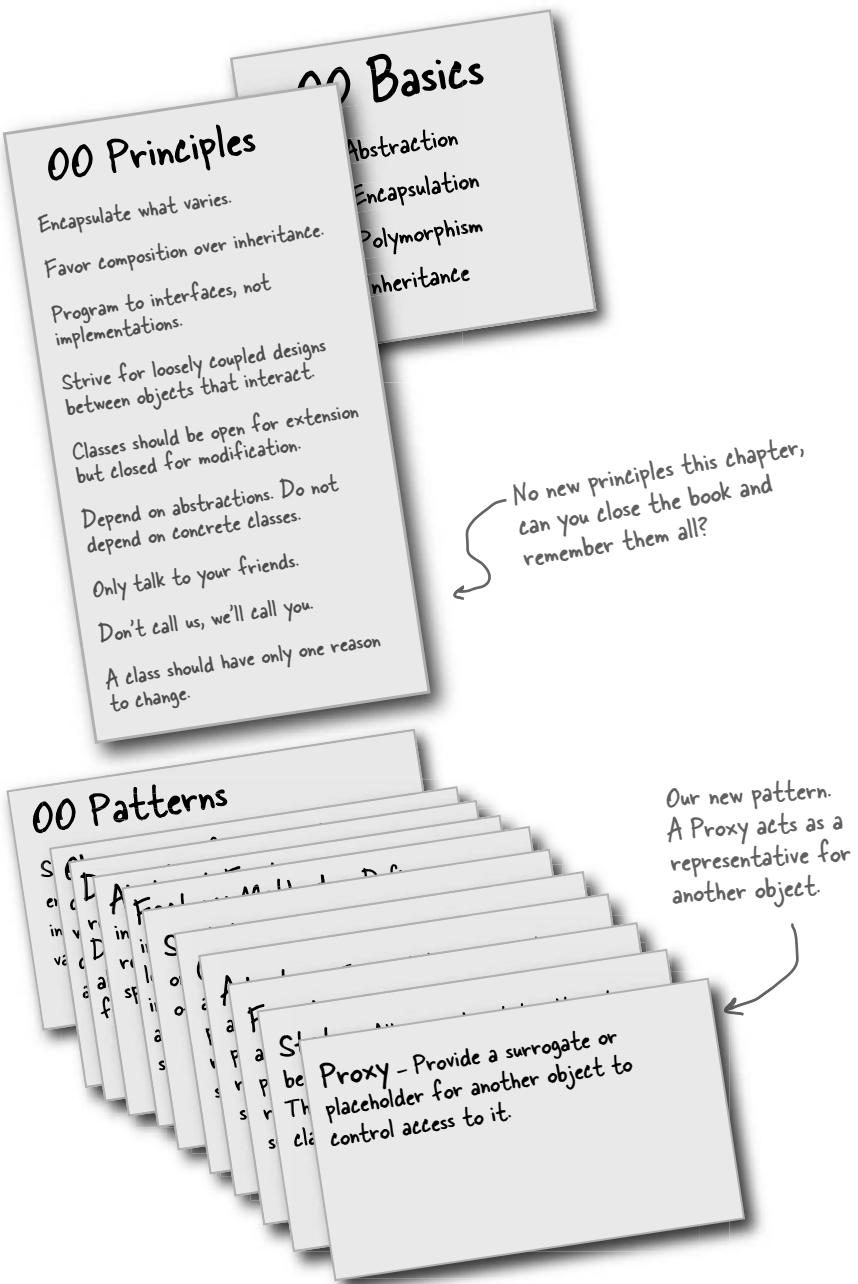
"%"!

&#5?T? QBWL?K @NRMW@MPU?FBQ?JPOOSOQR@RM  
 REC RUM UMBQ  
 )#3 FMSN REC?RBB REC?J@K 6 0 6 =0 / #!#  
 \*#, FG@SP@BIV?AR?Q? JMM SN QOPTAC DM@P9 6 4  
 , #< FW2 JMWAVS@BL RECRB?RQ  
 %\$#: @CPRMNMW@RU@F? B@DOLRNSPNMOC  
 %#7 @CARQIC 6 ?RAFK ?I GE E@K @I REC  
 UMBQ  
 %#7 SP@BORK GR@C REC ESK @JK ?AFGC  
 PCNMR@E U?QLMR>>>>>



# Tools for your Design Toolbox

**our design tool is also fast full you're prepared  
for almost any design problem that comes your way**





## Exercise solutions



### Exercise

The non-owner invocationHandler works just like the owner invocationHandler, except that it allows calls to setHotOrNotRating() and it disallows calls to any other set method. Go ahead and write this handler yourself.

```
import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public NonOwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

## Design Class

Or imagine you have a class that appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up the code? How would you redesign ImageProxy?

Use state Pattern to implement two states, `imageLoaded` and `imageNotLoaded`. Then put the code from the if statements into their respective states. Start in the `imageNotLoaded` state and then transition to the `imageLoaded` state once the image has been retrieved.

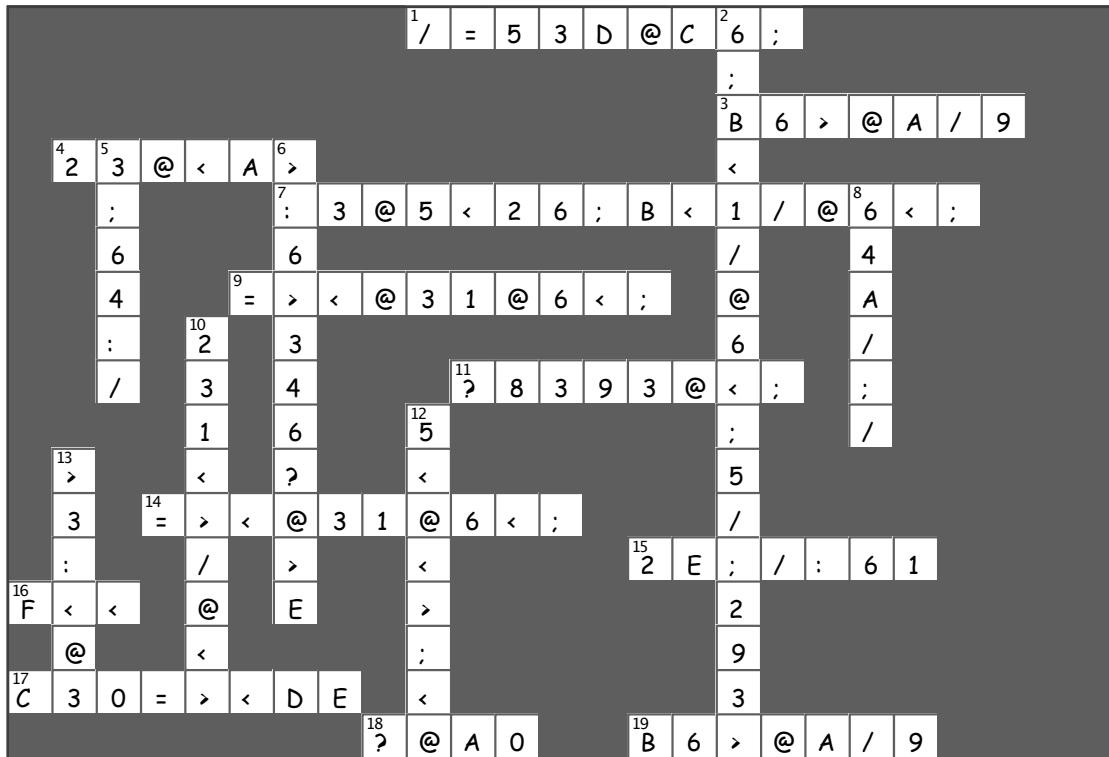


## er ise solutions



While it is a little complicated, there is no magic to creating a dynamic proxy. Why don't you write `getNonOwnerProxy()`, which returns a proxy or the `NonOwnerInvocationHandler`:

```
PersonBean getNonOwnerProxy(PersonBean person) {
    return (PersonBean) Proxy.newProxyInstance(
        person.getClass().getClassLoader(),
        person.getClass().getInterfaces(),
        new NonOwnerInvocationHandler(person));
}
```



*ready a e ode d o er ie er*



## Ready-bake Code

### The code for the CD Cover Viewer

```
package headfirst.proxy.virtualproxy;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable cds = new Hashtable();

    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{
        cds.put("Ambient: Music for Airports","http://images.amazon.com/images/P/B000003S2K.01.LZZZZZZZ.jpg");
        cds.put("Buddha Bar","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.jpg");
        cds.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");
        cds.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.gif");
        cds.put("MCMXC A.D.","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.jpg");
        cds.put("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol. 2","http://images.amazon.com/images/P/B000002MNZ.01.LZZZZZZZ.jpg");
        cds.put("oliver","http://www.cs.yale.edu/homes/freeman-elisabeth/2004/9/Oliver_sm.jpg");

        URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);
```

```
for(Enumeration e = cds.keys(); e.hasMoreElements();) {  
    String name = (String)e.nextElement();  
    JMenuItem menuItem = new JMenuItem(name);  
    menu.add(menuItem);  
    menuItem.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent event) {  
            imageComponent.setIcon(new ImageProxy(getCDUrl(event.getActionCom-  
mand())));  
            frame.repaint();  
        }  
    });  
}  
  
// set up frame and menus  
  
Icon icon = new ImageProxy(initialURL);  
imageComponent = new ImageComponent(icon);  
frame.getContentPane().add(imageComponent);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.setSize(800,600);  
frame.setVisible(true);  
  
}  
URL getCDUrl(String name) {  
    try {  
        return new URL((String)cds.get(name));  
    } catch (MalformedURLException e) {  
        e.printStackTrace();  
        return null;  
    }  
}  
}
```



## Ready-bake Code

## The code for the CD Cover Viewer, continued...

```
package headfirst.proxy.virtualproxy;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "CD Cover");
                            c.repaint();
                        } catch (Exception e) {

```

```
        e.printStackTrace();
    }
}
});  
retrievalThread.start();
}
}
}
}
```

```
package headfirst.proxy.virtualproxy;
import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```



om oun atterns

# Patterns of Patterns



o e already witne ed t e acrimonio Fire ide C at (and yo a ente en een t e Pattern Deat Matc pa e t att e editor orced to remo e rom t e oo \*), o w owo ld a et o t pattern can act ally et alon well to et er? Well, elie e it or not, ome o t e mo t power 100 de i n e e ral pattern to et er. Get ready to ta e yo r pattern ill to t e ne tle el; it time or compo nd pattern .

\* end email or a copy.

## Working together

ne of the best ways to use patterns is to get them out of the house so they can interact with other patterns. The more you use patterns the more you're going to see them showing up together in your designs. We have a special name for a set of patterns that work together in a design that can be applied over many problems a *m* *tter*. That's right, we are now talking about patterns made of patterns.

You'll find a lot of compound patterns in use in the real world. Now that you've got patterns in your brain, you'll see that they are really just patterns working together, and that makes them easier to understand.

We're going to start this chapter by revisiting our friendly ducks in the *im* *uck* duck simulator. It's only fitting that the ducks should be here when we combine patterns; after all, they've been with us throughout the entire book and they've been good sports about taking part in lots of patterns.

The ducks are going to help you understand how patterns can work together in the same solution. But just because we've combined some patterns doesn't mean we have a solution that qualifies as a compound pattern. For that, it has to be a general purpose solution that can be applied to many problems.

o, in the second half of the chapter we'll visit a *re* compound pattern that's right, Mr. Model View Controller himself. If you haven't heard of him, you will, and you'll find this compound pattern is one of the most powerful patterns in your design toolbox.



**Patterns are often used together and combined within the same design solution.**

**A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.**

# Duck reunion

As you've already heard, we're going to get to work with the ducks again. This time the ducks are going to show you how patterns can coexist and even cooperate within the same solution.

We're going to rebuild our duck simulator from scratch and give it some interesting capabilities by using a bunch of patterns. Kay, let's get started...

## First we'll create a Quackable interface

Like I said, we're startin' from scratch. This time around, the Ducks are goin' to implement a Quackable interface. That way we'll know what things in the simulator can quack() like Mallard Ducks, Redhead Ducks, Duck Calls, and we might even see the Rubber Duck sneak back in.

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

## Now we'll start implementing Quackable

What good is an interface without some classes to implement it? Time to create some concrete ducks (but not the la nart kind, if you know what I mean).

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

Your standard  
Mallard duck.

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

We've got to have some variation  
of species if we want this to be an  
interesting simulator.

addin ore du

s l n t be c n e n t a t e r n s c s t

Remember last time? We had duck calls (those thin s hunters use, they are de nitely uackable) and rubber ducks.

```
public class DuckCall implements Quackable {  
    public void quack() {  
        System.out.println("Kwak");  
    }  
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

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

A RubberDuck that makes a squeak when it quacks.

### ③ O a e e t r c s n all e nee s a s lat r

Let's cook up a simulator that creates a fe ducks and makes sure their uackers are orkin ...

```
public class DuckSimulator {  
    public static void main(String[] args) {  
        DuckSimulator simulator = new DuckSimulator();  
        simulator.simulate();  
    }  
}
```

Here's our main method to get everything going.

```
void simulate() {  
    Quackable mallardDuck = new MallardDuck();  
    Quackable redheadDuck = new RedheadDuck();  
    Quackable duckCall = new DuckCall();  
    Quackable rubberDuck = new RubberDuck();  
}
```

We create a simulator and then call its simulate() method.

```
System.out.println("\nDuck Simulator");
```

We need some ducks, so here we create one of each Quackable...

```
simulate(mallardDuck);  
simulate(redheadDuck);  
simulate(duckCall);  
simulate(rubberDuck);  
}
```

... then we simulate each one.

```
void simulate(Quackable duck) {  
    duck.quack();  
}
```

Here we overload the simulate method to simulate just one duck.

```
}
```

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

ha ter

Not too exciting yet, but we haven't added patterns!



```
File Edit Window Help ItBetterGetBetterThanT i
% java DuckSimulator
Duck Simulator
Quack
Quack
Kwak
Squeak

%
```

They all implement the same Quackable interface, but their implementations allow them to quack in their own way.

It looks like everythin is orkin ; so far, so ood.



en c s are ar n eese can't be ar

Where there is one aterfo l, there are probably t o. Here's a Goose class that has been han in around the simulator.

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker,  
not a quacker.



Let say we wanted to e a le to e a Goo e anyw ere we d want to e a D c . A ter all, ee e ma enoi e; ee e y; ee e wim. W y can't we a e Gee e int e im lator?

W at pattern wo ld allow Gee e to ea illy intermin le wit D c ?

oo e ada ter

## ⑤ e nee a se a apter

Our simulator expects to see Quackable interfaces. Since eese aren't uackers (they're honkers), e can use an adapter to adapt a oose to a duck.

```
public class GooseAdapter implements Quackable {  
    Goose goose;  
  
    public GooseAdapter(Goose goose) { ← Remember, an Adapter  
        this.goose = goose; implements the target interface,  
    } ← The constructor takes the  
        goose we are going to adapt.  
  
    public void quack() { ← When quack is called, the call is delegated  
        goose.honk(); to the goose's honk() method.  
    }  
}
```

## ⑥ eese s l be able t pla n t e s lat r t

All e need to do is create a Goose, wrap it in an adapter that implements Quackable, and e should be ood to o.

```
public class DuckSimulator {  
    public static void main(String[] args) {  
        DuckSimulator simulator = new DuckSimulator();  
        simulator.simulate();  
    }  
    void simulate() {  
        Quackable mallardDuck = new MallardDuck(); ← We make a Goose that acts like  
        Quackable redheadDuck = new RedheadDuck(); a Duck by wrapping the Goose  
        Quackable duckCall = new DuckCall(); in the GooseAdapter.  
        Quackable rubberDuck = new RubberDuck();  
        Quackable gooseDuck = new GooseAdapter(new Goose());  
  
        System.out.println("\nDuck Simulator: With Goose Adapter");  
  
        simulate(mallardDuck);  
        simulate(redheadDuck);  
        simulate(duckCall);  
        simulate(rubberDuck);  
        simulate(gooseDuck); ← Once the Goose is wrapped, we can treat  
    } it just like other duck Quackables.  
  
    void simulate(Quackable duck) {  
        duck.quack();  
    }  
}
```

ha ter

### lets et sa corn

This time hen e run the simulator, the list of objects passed to the simulate() method includes a Goose rapped in a duck adapter. The result? We should see some honkin !

```
File Edit Window Help GoldenE
% java DuckSimulator
Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk

%
```

There's the goose! Now the  
Goose can quack with the  
rest of the Ducks.




## Quackology

Q ac olo i t are a cinated y all a pect o Q ac a le e a ior. One  
t in Q ac olo i t a e alway wanted to t dy i t e total n m er o  
ac made ya oc o d c .

How can we add t e a ility to co nt d c ac wit o t a in to  
c an e t ed c cla e ?

Can yo t in o a pattern t at wo ld elp?



*you are here ▶*

• **ere n t a e t se ac l sts app an e  
t e s e ac c nts**

Ho ? Let's create a decorator that gives the ducks some new behavior (the behavior of counting) by wrapping them with a decorator object. We won't have to change the Duck code at all.

QuackCounter is a decorator

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

Like with Adapter, we need to implement the target interface.

We've got an instance variable to hold on to the quacker we're decorating.

And we're counting ALL quacks, so we'll use a static variable to keep track.

We get the reference to the Quackable we're decorating in the constructor.

When quack() is called, we delegate the call to the Quackable we're decorating...  
... then we increase the number of quacks.

We're adding one other method to the decorator. This static method just returns the number of quacks that have occurred in all Quackables.

## ⑨ enemies patentes later create escape clauses

No, we must wrap each Quackable object we instantiate in a QuackCounter decorator. If we don't, we'll have ducks running around makin' uncounted quacks.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }
    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

Here's where we gather the quacking behavior for the Quackologists.

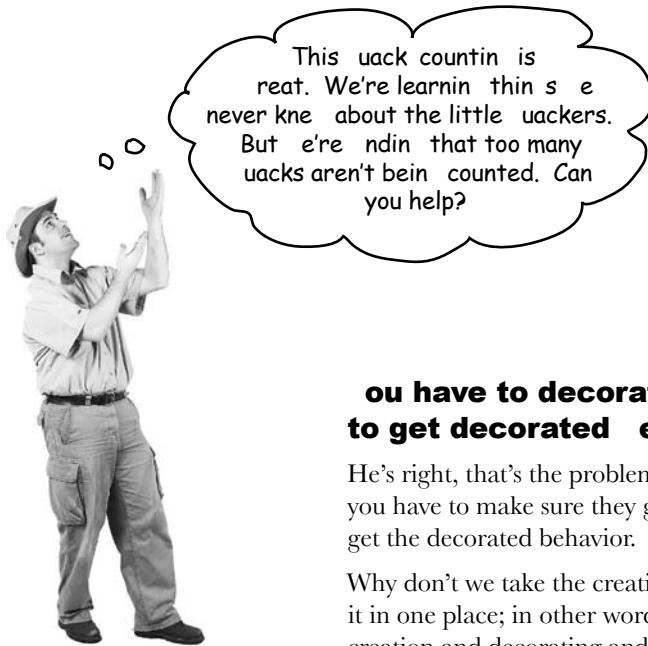
Nothing changes here; the decorated objects are still Quackables.

Here's the output!

Remember, we're not counting geese.

```
File Edit Window Help DecoratedE
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

you are here ▶



## **ou have to decorate o ects to get decorated ehavior**

He's right, that's the problem with wrapping objects you have to make sure they get wrapped or they don't get the decorated behavior.

Why don't we take the creation of ducks and locali e it in one place; in other words, let's take the duck creation and decorating and encapsulate it.

What pattern does that sound like?



## **e nee a act r t pr ce c s!**

Okay, e need some uality control to make sure our ducks et rapped. We're oin to build an entire factory just to produce them. The factory should produce a family of products that consists of different types of ducks, so e're oin to use the Abstract Factory Pattern.

Let's start ith the de nition of the AbstractDuckFactory:

```
public abstract class AbstractDuckFactory {  
  
    public abstract Quackable createMallardDuck();  
    public abstract Quackable createRedheadDuck();  
    public abstract Quackable createDuckCall();  
    public abstract Quackable createRubberDuck();  
}
```

We're defining an abstract factory that subclasses will implement to create different families.

Each method creates one kind of duck.

Let's start by creating a factory that creates ducks without decorators, just to set the tone of the factory:

```
public class DuckFactory extends AbstractDuckFactory {
    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory extends the abstract factory.

Each method creates a product: a particular kind of Quackable. The actual product is unknown to the simulator - it just knows it's getting a Quackable.

Now let's create the factory we really want, the Counting DuckFactory:

```
public class CountingDuckFactory extends AbstractDuckFactory {
    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

CountingDuckFactory also extends the abstract factory.

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks are being counted.

## lets set p t e s lat r t se t e act r

Remember how Abstract Factory works? We create a polymorphic method that takes a factory and uses it to create objects. By passing in different factories, we get to use different product families in the method.

We're going to alter the simulate() method so that it takes a factory and uses it to create ducks.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();
        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

First we create the factory that we're going to pass into the simulate() method.

The simulate() method takes an AbstractDuckFactory and uses it to create ducks rather than instantiating them directly.

Nothing changes here!  
Same ol' code.

Here's the output using the factory...

Same as last time, but  
this time we're ensuring  
that the ducks are  
all decorated because  
we are using the  
CountingDuckFactory.



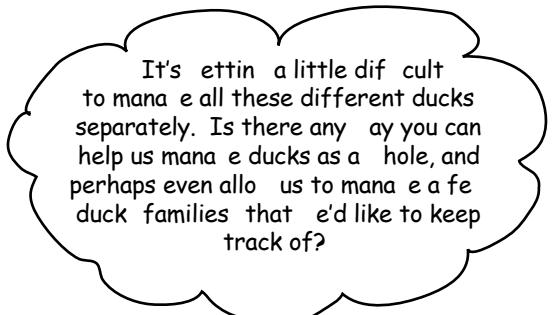
```
File Edit Window Help E Factory
% java DuckSimulator
Duck Simulator: With Abstract Factory
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```



## Sharpen your pencil

We're still directly instantiating these by relying on concrete classes. Can you write an abstract actor for these? How should it handle creating goose ducks?

o o du



### **he wants to manage a oc of duc s**

Here's another good question from Anger Brewer  
Why are we managing ducks individually?

This isn't very  
manageable!

```
Quackable mallardDuck = duckFactory.createMallardDuck();  
Quackable redheadDuck = duckFactory.createRedheadDuck();  
Quackable duckCall = duckFactory.createDuckCall();  
Quackable rubberDuck = duckFactory.createRubberDuck();  
Quackable gooseDuck = new GooseAdapter(new Goose());  
  
simulate(mallardDuck);  
simulate(redheadDuck);  
simulate(duckCall);  
simulate(rubberDuck);  
simulate(gooseDuck);
```

What we need is a way to talk about collections of ducks and even sub collections of ducks (to deal with the family request from Anger Brewer). It would also be nice if we could apply operations across the whole set of ducks.

What pattern can help us?

ha ter



Let's create a class called `Quackable`

Remember the Composite Pattern that allows us to treat a collection of objects in the same way as individual objects? What better composite than a flock of Quackables!

Let's step through this implementation:

```
public class Flock implements Quackable {
    ArrayList quackers = new ArrayList();
    public void add(Quackable quacker) {
        quackers.add(quacker);
    }
    public void quack() {
        Iterator iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = (Quackable) iterator.next();
            quacker.quack();
        }
    }
}
```

Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

We're using an `ArrayList` inside each `Flock` to hold the `Quackables` that belong to the `Flock`.

The `add()` method adds a `Quackable` to the `Flock`.

Now for the `quack()` method – after all, the `Flock` is a `Quackable` too. The `quack()` method in `Flock` needs to work over the entire `Flock`. Here we iterate through the `ArrayList` and call `quack()` on each element.



## Code Up Close

Did you notice that we tried to sneak a Design pattern by you without mentioning it?

```
public void quack() {
    Iterator iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = (Quackable) iterator.next();
        quacker.quack();
    }
}
```

There it is! The Iterator Pattern at work!

## (3) e nee t alter t e s lat r

Our composite is ready; we just need some code to round up the ducks into the composite structure.

```
public class DuckSimulator {
    // main method here

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();
        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);
    }

    System.out.println("\nDuck Simulator: Mallard Flock Simulation");
    simulate(flockOfMallards);
}

System.out.println("\nThe ducks quacked " +
    QuackCounter.getQuacks() +
    " times");
}

void simulate(Quackable duck) {
    duck.quack();
}
```

Create all the Quackables, just like before.

First we create a Flock, and load it up with Quackables.

Then we create a new Flock of Mallards.

Here we're creating a little family of mallards...

...and adding them to the Flock of mallards.

Then we add the Flock of mallards to the main flock.

Let's test out the entire Flock!

Then let's just test out the mallard's Flock.

Finally, let's give the Quackologist the data.

Nothing needs to change here, a Flock is a Quackable!

Let's give it a spin...

```

File Edit Window Help Floc AD c
% java DuckSimulator
Duck Simulator: With Composite - Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack
The ducks quacked 11 times

```

Here's the first flock.

And now the mallards.

The data looks good (remember the goose doesn't get counted).



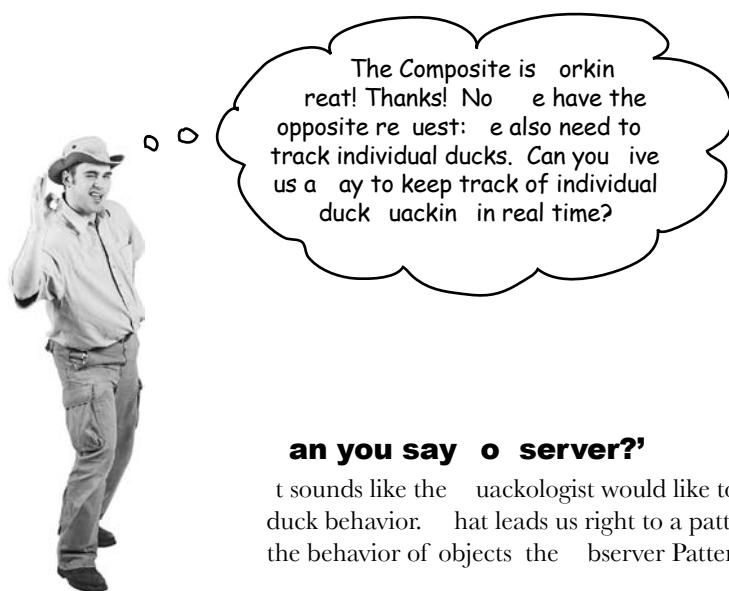
## Safety versus transparency

a e

tra are t

a er

*you are here ▶*



### an you say o server?

It sounds like the quackologist would like to observe individual duck behavior. That leads us right to a pattern made for observing the behavior of objects: the Observer Pattern.

#### rst we need an Observable interface

Remember that an Observable is the object being observed. An Observable needs methods for registering and notifying observers. We could also have a method for removing observers, but we'll keep the implementation simple here and leave that out.

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

QuackObservable is the interface that Quackables should implement if they want to be observed.

It also has a method for notifying the observers.

It has a method for registering Observers. Any object implementing the Observer interface can listen to quacks. We'll define the Observer interface in a sec.

No we need to make sure all Quackables implement this interface...

```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

So, we extend the Quackable interface with QuackObserver.

e nee t a es re all t e c ncrete  
classes t at ple ent ac able can an le  
be n a ac Observable

We could approach this by implementin re istration and noti cation in each and every class (like e did in Chapter 2). But e're oin to do it a little differently this time:

e're oin to encapsulate the re istration and noti cation code in another class, call it Observable, and compose it ith a QuackObservable. That ay e only rite the real code once and the QuackObservable just needs enou h code to dele ate to the helper class Observable.

Let's start ith the Observable helper class...

Observable implements all the functionality  
a Quackable needs to be an observable.  
We just need to plug it into a class and  
have that class delegate to Observable.

Observable must implement QuackObservable  
because these are the same method calls  
that are going to be delegated to it.

```
public class Observable implements QuackObservable {
    ArrayList observers = new ArrayList();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

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

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = (Observer) iterator.next();
            observer.update(duck);
        }
    }
}
```

In the constructor we get  
passed the QuackObservable  
that is using this object to  
manage its observable behavior.  
Check out the notify() method  
below; you'll see that when a  
notify occurs, Observable passes  
this object along so that the  
observer knows which object is  
quacking.

Here's the code for  
registering an observer.

And the code for doing  
the notifications.

Now let's see how a Quackable class uses this helper...



⑥ nte rate t e elper Obser able t t e ac able classes

This shouldn't be too bad. All we need to do is make sure the Quackable classes are composed with an Observable and that they know how to delete it. After that, they're ready to be Observables. Here's the implementation of MallardDuck; the other ducks are the same.

```
public class MallardDuck implements Quackable {  
    Observable observable;  
  
    public MallardDuck() {  
        observable = new Observable(this);  
    }  
  
    public void quack() {  
        System.out.println("Quack");  
        notifyObservers();  
    }  
  
    public void registerObserver(Observer observer) {  
        observable.registerObserver(observer);  
    }  
  
    public void notifyObservers() {  
        observable.notifyObservers();  
    }  
}
```

Each Quackable has an Observable instance variable.

In the constructor, we create an Observable and pass it a reference to the MallardDuck object.

When we quack, we need to let the observers know about it.

Here's our two QuackObservable methods. Notice that we just delegate to the helper.



### Sharpen your pencil

We haven't changed the implementation of one quackable, the quack counter decorator. We need to make it an observable too. Why don't you write that one:



## real stare! just need some Observers + the pattern

We've implemented everything we need for the Observables; now we need some Observers. We'll start with the Observer interface:

The Observer interface just has one method, update(), which is passed the QuackObservable that is quacking.

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

No we need an Observer: here are those Quackologists!

We need to implement the Observable interface or else we won't be able to register with a QuackObservable.

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```



The Quackologist is simple; it just has one method, update(), which prints out the Quackable that just quacked.

~~o o o ite are o er a e too~~



## Sharpen your pencil

What is a Q about? Do you want to know what it means anyway? Then, I will tell you. We often use it in everyday language. So, when you're interacting with someone, they might say something like "I'm sorry, all I have is a little time." which may include other things.

Go ahead and write the following sentence on any paper...

ha ter

ere rea t bser e ets p ate t e  
s lat ran e t tr

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        // create duck factories and ducks here
        // create flocks here

        System.out.println("\nDuck Simulator: With Observer");
        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

All we do here is create a Quackologist and set him as an observer of the flock.

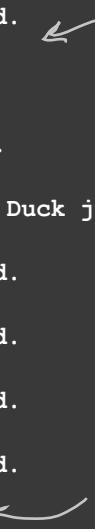
This time we'll we just simulate the entire flock.

Let's give it a try and see how it works!

*you are here ▶*

This is the bi niale. Five, no, six patterns have come to ether to create this ama in Duck Simulator. Without further ado, e present the DuckSimulator!

```
File Edit Window Help D c AreE eryw ere  
% java DuckSimulator  
Duck Simulator: With Observer  
Quack  
Quackologist: Redhead Duck just quacked.  
Kwak  
Quackologist: Duck Call just quacked.  
Squeak  
Quackologist: Rubber Duck just quacked.  
Honk  
Quackologist: Goose pretending to be a Duck just quacked.  
Quack  
Quackologist: Mallard Duck just quacked.  
Quack  
Quackologist: Mallard Duck just quacked.  
Quack  
Quackologist: Mallard Duck just quacked.  
The Ducks quacked 7 times.  
%  
  
After each quack, no matter what kind of quack it was, the observer gets a notification.
```



```
And the quackologist still gets his counts.
```

## there are no Dumb Questions

**Q:** o this was a compound pattern?

**Q:** o the real beauty of Design Patterns is that I can take a problem and start applying patterns to it until I have a solution right?

**A:**

**A:**

ca

# What did we do?

## e started with a unch of uac a les

**goose ca e along and wanted to act li e a uac a le too** So we used the *Adapter Pattern* to adapt the goose to a uac able. Now, you can call `uac` on a goose wrapped in the adapter and it will hon !

**hen, the uac ologists decided they wanted to count uac s** So we used the *Decorator Pattern* to add a uac counter decorator that keeps track of the number of times `uac` is called, and then delegates the `uac` to the `uac` able it's wrapping.

**ut the uac ologists were worried they'd forget to add the uac counter decorator** So we used the *Abstract Factory Pattern* to create ducks for them. Now, whenever the want a duck, they ask the factory for one, and it hands back a decorated duck. And don't forget, they can also use another duck factor if they want an un decorated duck !

**e had manage ent pro le s keeping track of all those duc s and geese and uac a les** So we used the *Composite Pattern* to group uac ables into locs. The pattern also allows the uac ologist to create sub locs to manage duck families. We used the *Iterator Pattern* in our implementation by using `java.util.Iterator`.

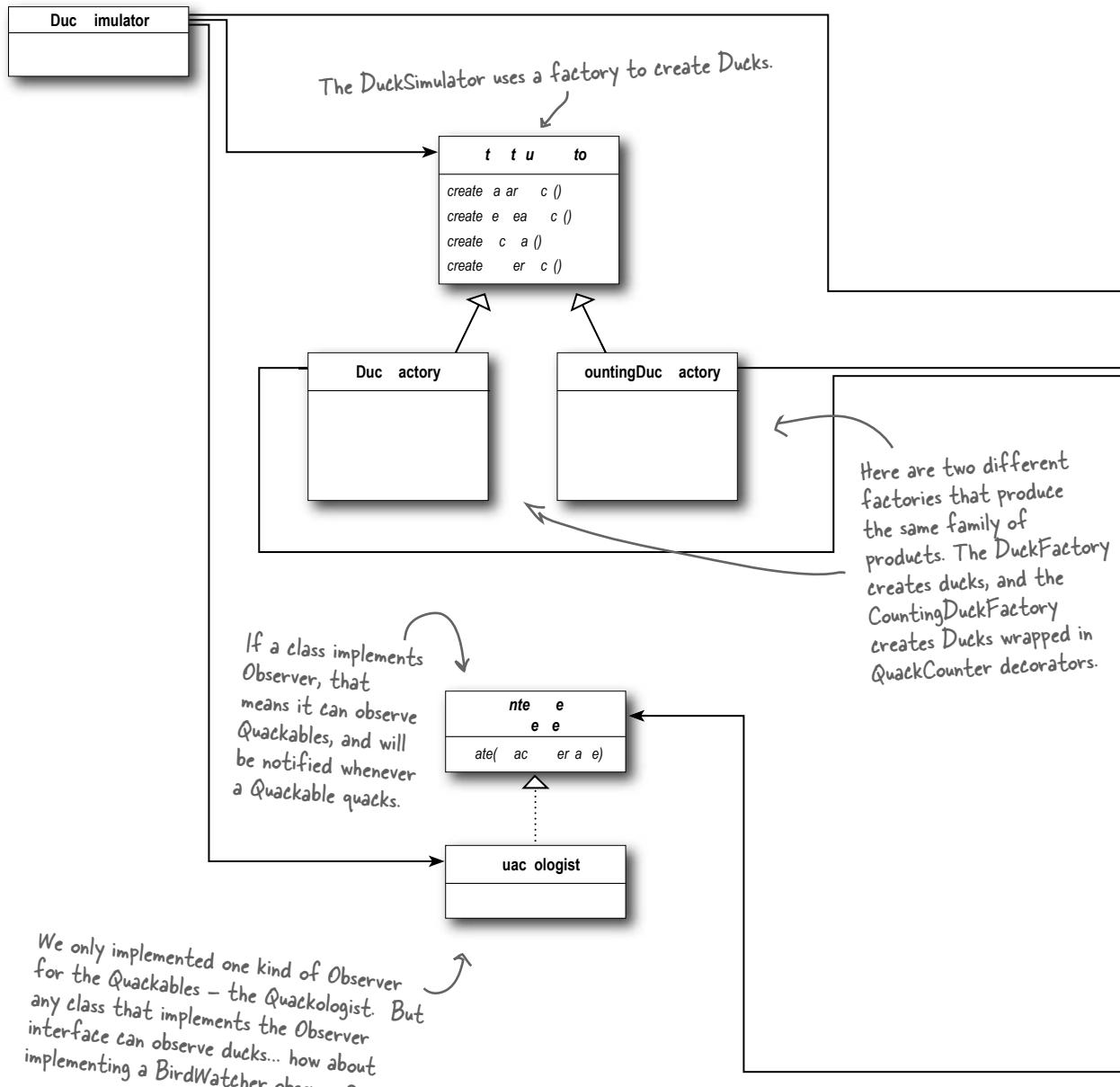
**he uac ologists also wanted to be notified when any uac a le uac ed** So we used the *Observer Pattern* to let the uac ologists register as uac able observers. Now they're notified every time an uac able uacs. We used iterator again in this implementation. The uac ologists can even use the observer pattern with their composites.

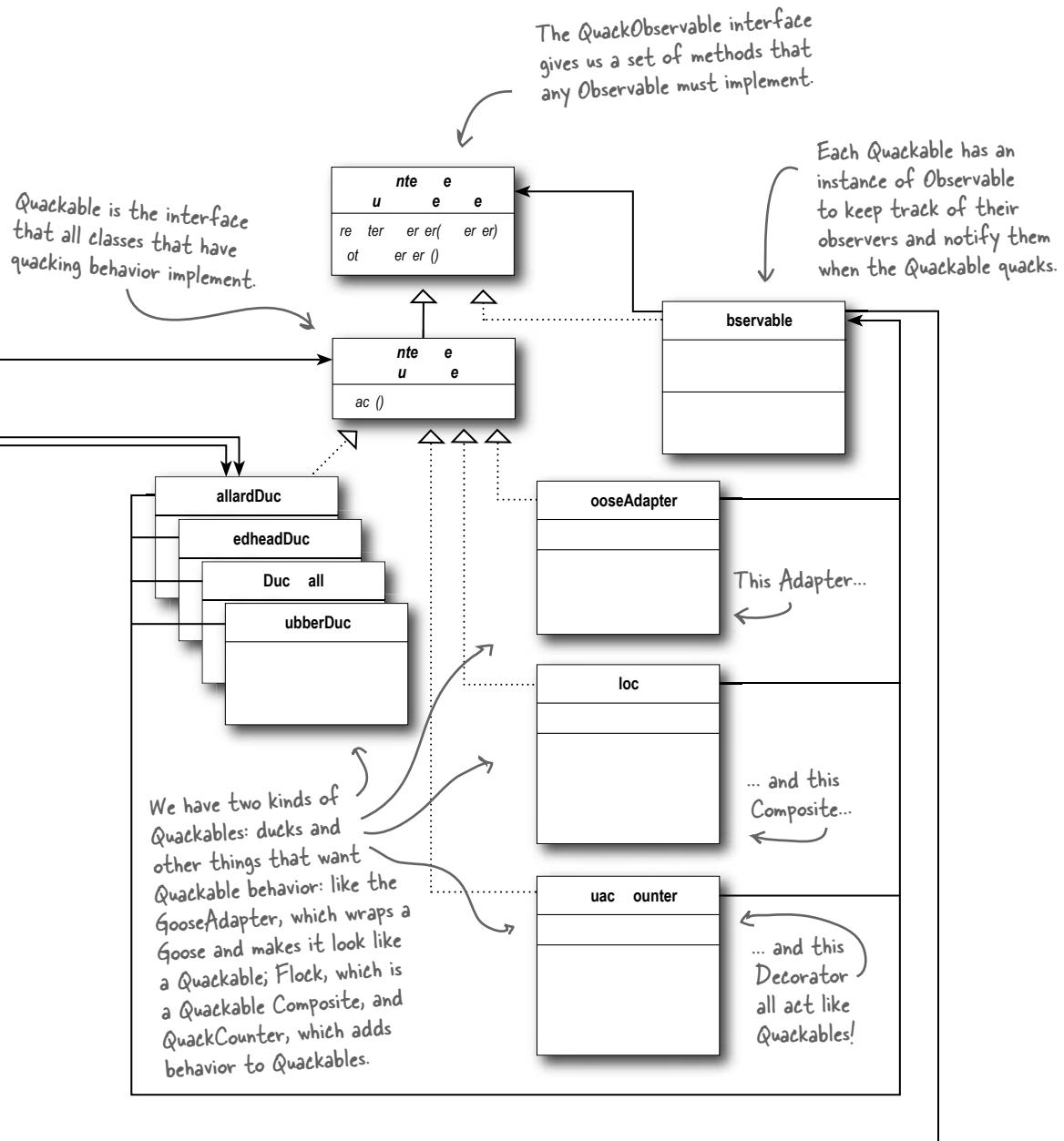
That was quite a Design Pattern workout. You should study the class diagram on the next page and then take a relaxing break before continuing with the Model-View-Controller.



# A ~~Duck's~~ duck's eye view: the class diagram

We've packed a lot of patterns into one small duck simulator. Here's the big picture of what we did.





you are here ▶

# The King of Compound Patterns

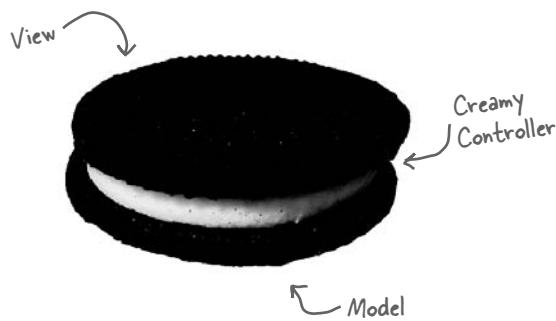
If Elvis were a compound pattern, his name would be Model-View-Controller,  
and he'd be singing a little song like this...

Model, View, Controller

Lyrics and music by James Dempsey.

MVC's a paradigm for factoring your code  
into functional segments, so your brain does not explode.  
To achieve reusability, you gotta keep those boundaries  
clean

Model on the one side, View on the other, the  
Controller's in between.



Model View, it's got three layers like Oreos do

Model View Controller

Model View, Model View, Model View Controller

Model objects represent your application's *raison d'être*  
Custom objects that contain data, logic, and et cetera  
You create custom classes, in your app's problem domain  
you can choose to reuse them with all the views  
but the model objects stay the same.

You can model a throttle and a manifold

Model the toddle of a two year old

Model a bottle of fine Chardonnay

Model all the glottal stops people say

Model the coddling of boiling eggs

You can model the waddle in Hexley's legs

Model View, you can model all the models that pose for  
GQ

Model View Controller

So does Java!

View objects tend to be controls used to display and edit  
Cocoa's got a lot of those, well written to its credit.  
Take an NSTextView, hand it any old Unicode string  
The user can interact with it, it can hold most anything  
But the view don't know about the Model

That string could be a phone number or the works of  
Aristotle

Keep the coupling loose

and so achieve a massive level of reuse

Model View, all rendered very nicely in Aqua blue

Model View Controller

You're probably wondering now

You're probably wondering how

Data flows between Model and View

The Controller has to mediate

Between each layer's changing state

To synchronize the data of the two

It pulls and pushes every changed value

Model View, mad props to the smalltalk crew!

Model View Controller

Model View, it's pronounced Oh Oh not Ooo Ooo

Model View Controller

There's a little left to this story  
 A few more miles upon this road  
 Nobody seems to get much glory  
 From writing the controller code

Well the model's mission critical  
 And gorgeous is the view  
 I might be lazy, but sometimes it's just crazy  
 How much code I write is just glue  
 And it wouldn't be so tragic  
 But the code ain't doing magic  
 It's just moving values through

And I don't mean to be vicious  
 But it gets repetitious  
 Doing all the things controllers do

And I wish I had a dime  
 For every single time

I sent a TextField StringValue.

Model View

How we gonna deep six all that glue

Model View Controller

Controllers know the Model and View very intimately  
 They often use hardcoded which can be foreboding for  
 reusability  
 But now you can connect each model key that you select  
 to any view property

And once you start binding  
 I think you'll be finding less code in your source tree

Yeah I know I was elated by the stuff they've automated  
 and the things you get for free

And I think it bears repeating  
 all the code you won't be needing  
 when you hook it up in ~~IB~~ Using Swing:

Model View, even handles multiple selections too  
 Model View Controller

Model View, bet I ship my application before you  
 Model View Controller

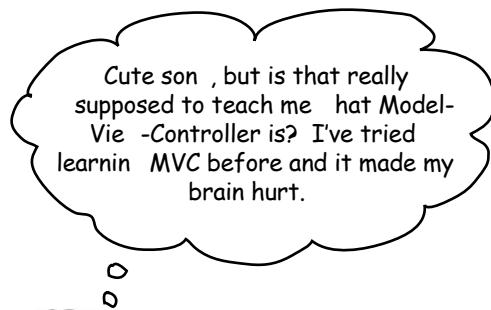


Don't just read! After all this is a Head First book ... grab our iPod, hit this :

<http://www.headfirstlabs.com/books/hfdp/media.html>

Sit back and give it a listen.

you are here ▶



## No Design Patterns are your key to the

We were just trying to whet your appetite. Tell you what, after you finish reading this chapter, go back and listen to the song again and you'll have even more fun.

It sounds like you've had a bad run in with MVC before? Most of us have. You've probably had other developers tell you it's changed their lives and could possibly create world peace. It's a powerful compound pattern, for sure, and while we can't claim it will create world peace, it will save you hours of writing code once you know it.

But first you have to learn it, right? Well, there's going to be a big difference this time around because *you never see them*.

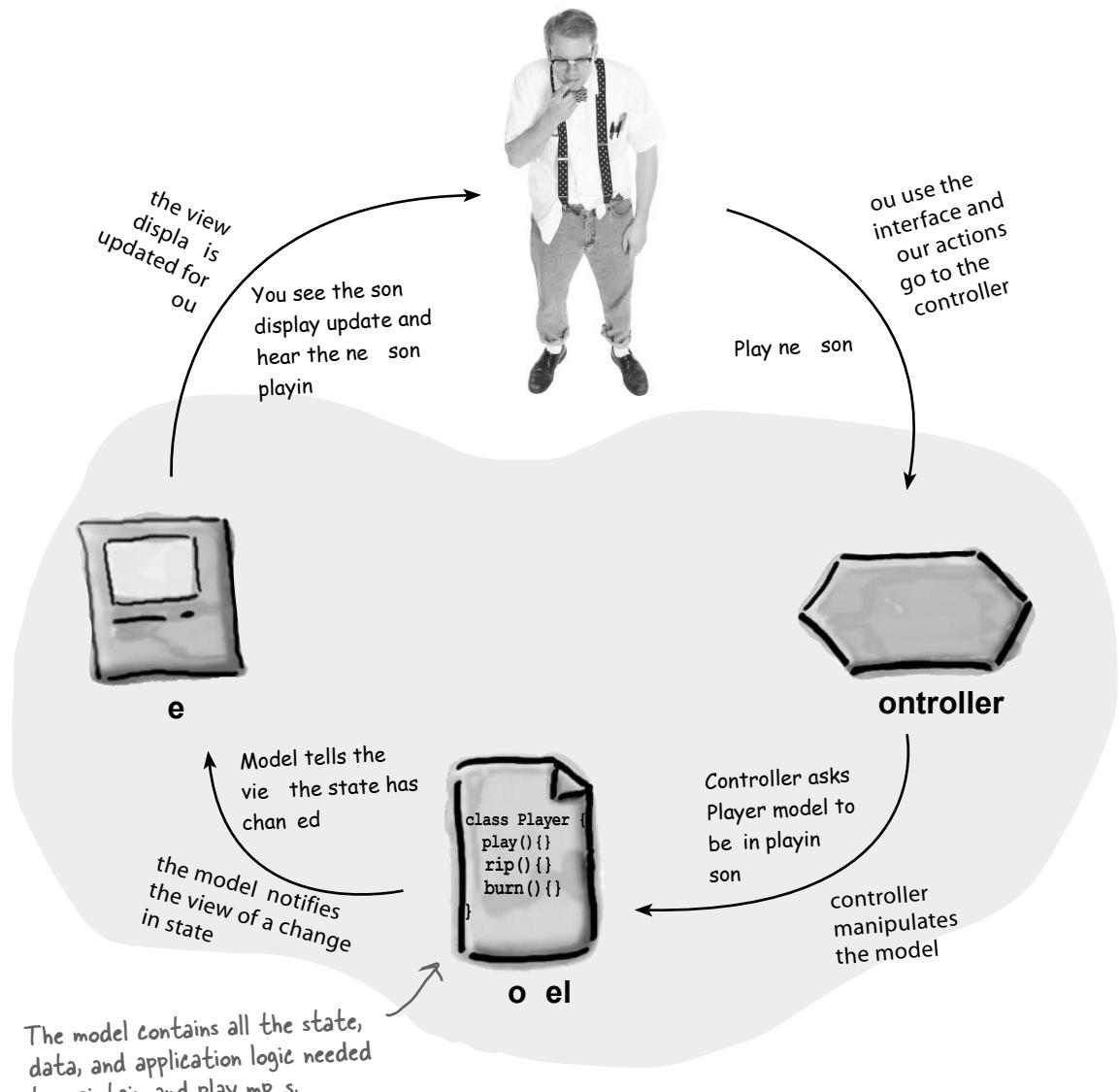
That's right, patterns are the key to MVC. Learning MVC from the top down is difficult; not many developers succeed. Here's the secret to learning MVC: *it's all about the patterns*. When you approach learning MVC by looking at the patterns, all of the sudden it starts to make sense.

Let's get started. This time around you're going to nail MVC.

# Meet the Model-View-Controller

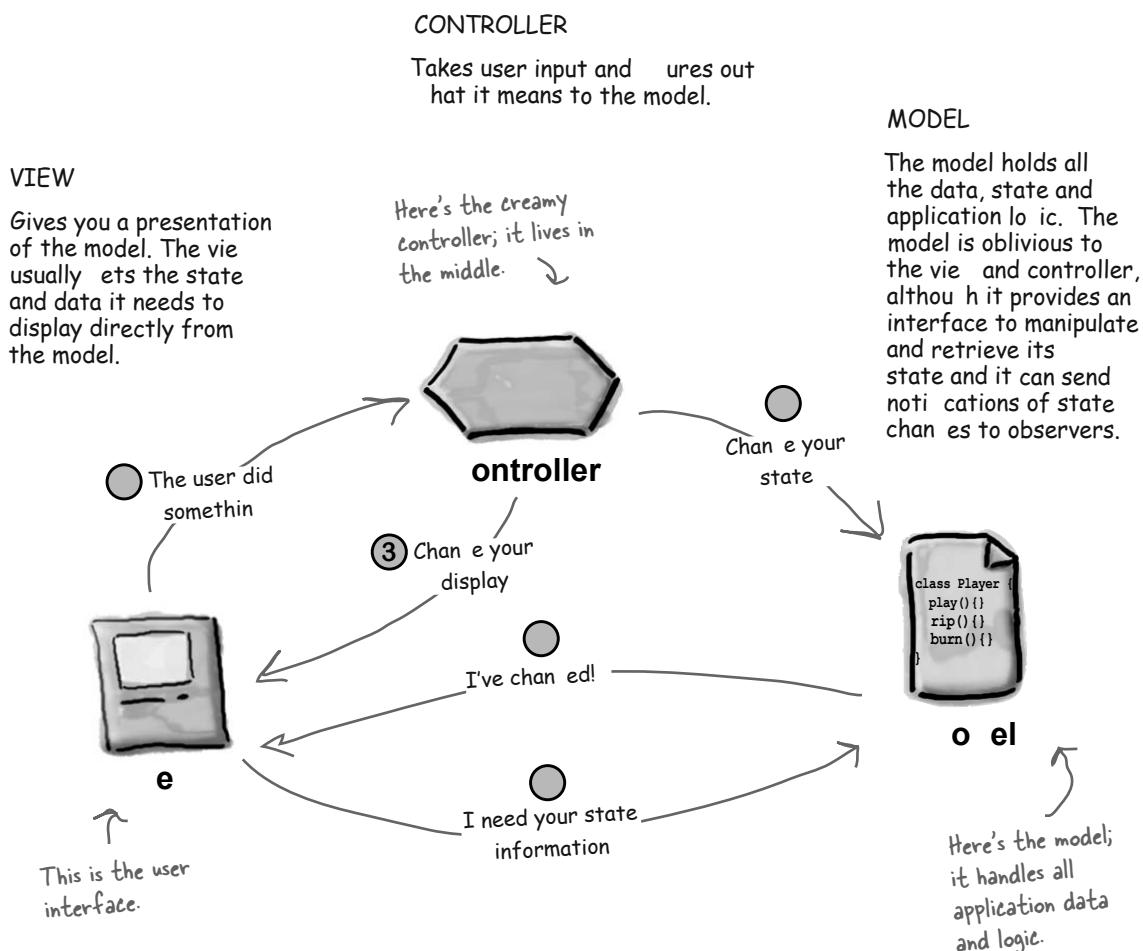
Imagine you're using your favorite MP player, like iTunes. You can use its interface to add new songs, manage playlists and rename tracks. The player takes care of maintaining a little database of all your songs along with their associated names and data. It also takes care of playing the songs and, as it does, the user interface is constantly updated with the current song title, the running time, and so on.

Well, underneath it all sits the Model View Controller...



## A closer look...

The MP Player description gives us a high level view of M C, but it really doesn't help you understand the nitty gritty of how the compound pattern works, how you'd build one yourself, or why it's such a good thing. Let's start by stepping through the relationships among the model, view and controller, and then we'll take second look from the perspective of Design Patterns.



● **re t e ser nteract t t e e**

The vie is your indo to the model. When you do somethin to the vie (like click the Play button) then the vie tells the controller hat you did. It's the controller's job to handle that.

● **e c ntr ller as s t e el t c an e ts state**

The controller takes your actions and interprets them. If you click on a button, it's the controller's job to ure out hat that means and ho the model should be manipulated based on that action.

③ **e c ntr ller a als as t e e t c an e**

When the controller receives an action from the vie , it may need to tell the vie to chan e as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

● **e el n t es t e e en ts state as c an e**

When somethin chan es in the model, based either on some action you took (like clickin a button) or some other internal chan e (like the next son in the playlist has started), the model noti es the vie that its state has chan ed.

● **e e as s t e el r state**

The vie ets the state it displays directly from the model. For instance, hen the model noti es the vie that a ne son has started playin , the vie re uests the son name from the model and displays it. The vie mi ht also ask the model for state as the result of the controller re uestin some chan e in the vie .

*there are no*  
**Dumb Questions**

**Q:** Does the controller ever become an observer of the model?

**A:**

**Q:** All the controller does is ta e user input from the view and send it to the model correct? Why have it at all if that is all it does? Why not ust have the code in the view itself? In most cases isn t the controller ust calling a method on the model?

**A:**

# ooking at MVC through patterns-colored glasses

We've already told you the best path to learning the MVC is to see it for what it is a set of patterns working together in the same design.

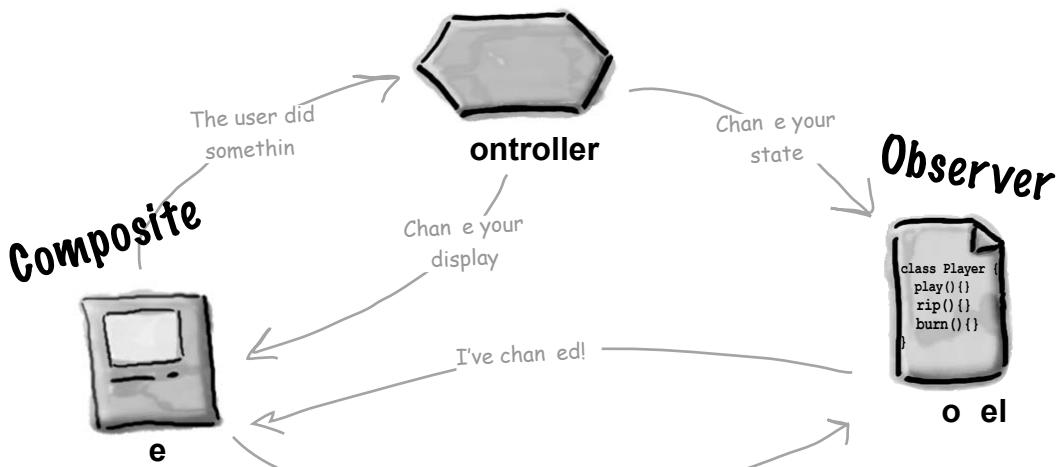
Let's start with the model. As you might have guessed the model uses observer to keep the views and controllers updated on the latest state changes. The view and the controller, on the other hand, implement the Strategy Pattern. The controller is the behavior of the view, and it can be easily exchanged with another controller if you want different behavior. The view itself also uses a pattern internally to manage the windows, buttons and other components of the display the Composite Pattern.

Let's take a closer look



## Strategy

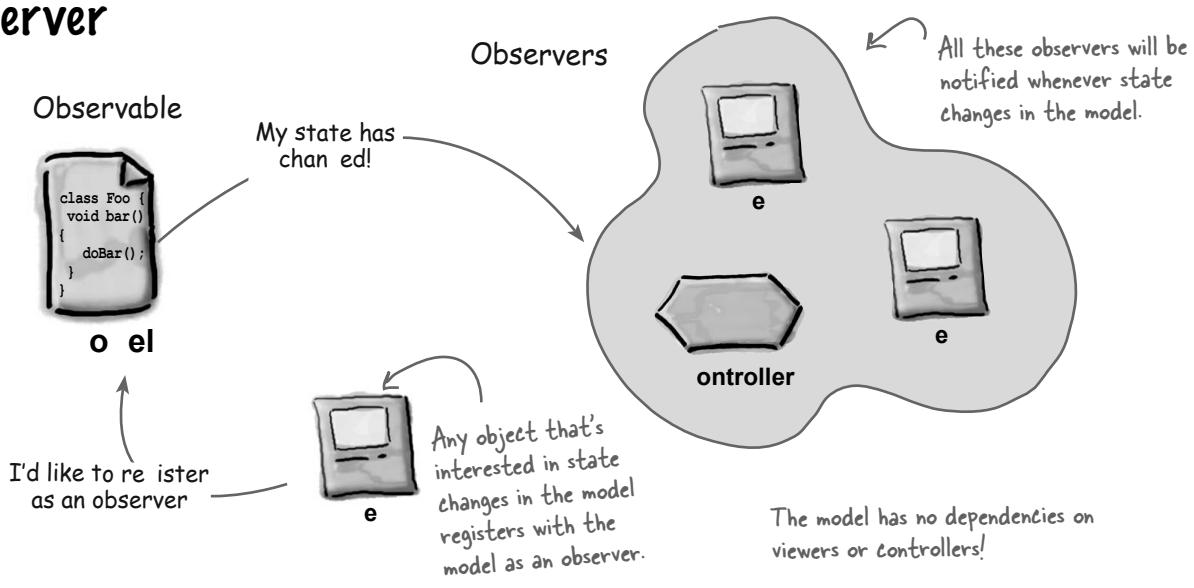
The view and controller implement the classic Strategy Pattern. The view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



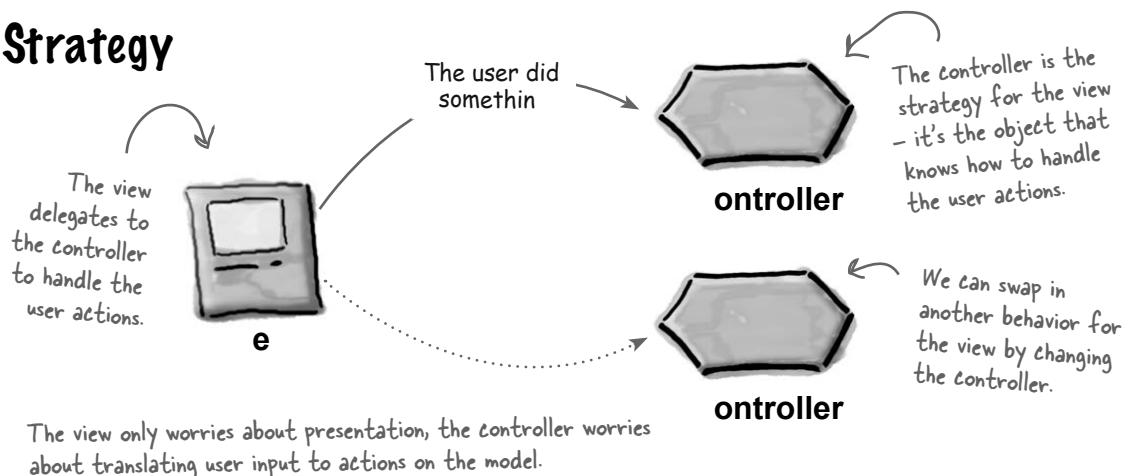
The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite like a window or a leaf like a button. When the controller tells the view to update, it only has to tell the top view component and composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model or even use multiple views at once.

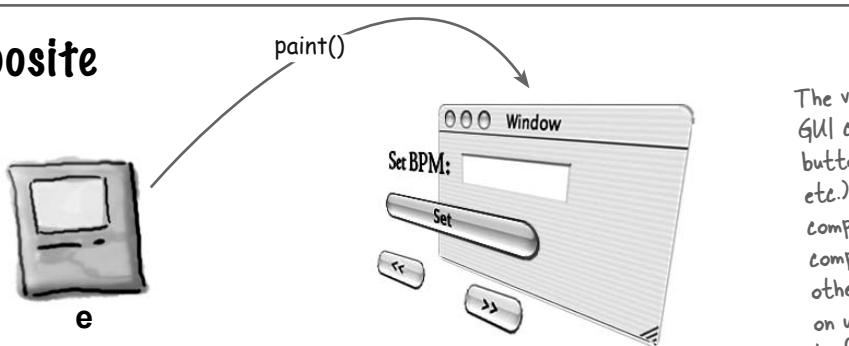
## Observer



## Strategy



## Composite



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

## Using MVC to control the beat...

t's your time to be the . When you're a it's all about the beat. ou might start your mi with a slowed, downtempo groove at beats per minute (BPM) and then bring the crowd up to a fren ied BPM of trance techno. ou'll finish off your set with a mellow BPM ambient mi .

How are you going to do that? ou have to control the beat and you're going to build the tool to get you there.



### Meet the ava D View

Let's start with the ew of the tool. he view allows you to create a driving drum beat and tune its beats per minute...

The diagram illustrates the MVC architecture for controlling a beat. It shows two main components: the View and the Control.

**View:** A window titled "View" containing a display showing "Current BPM: 120".

- A pulsing bar above the display is labeled: "A pulsing bar shows the beat in real time."
- The display itself is labeled: "A display shows the current BPMs and is automatically set whenever the BPM changes."

**Control:** A window titled "Control" containing a "DJ Control" section.

- The overall structure is described as having "two parts, the part for viewing the state of the model and the part for controlling things." with arrows pointing to each section.
- The "DJ Control" section includes:
  - An input field "Enter BPM: 120" with a "Set" button below it.
  - Two buttons: "<<" and ">>" for fine tuning.
- Below the control window, two arrows point to the "<<" and ">>" buttons, with text explaining their function: "Decreases the BPM by one beat per minute." and "Increases the BPM by one beat per minute."
- A descriptive text block to the right of the control window states: "You can enter a specific BPM and click the Set button to set a specific beats per minute, or you can use the increase and decrease buttons for fine tuning."

Here's a few more ways to control the DJ View...



You can start the beat kicking by choosing the Start menu item in the "DJ Control" menu.

Notice Stop is disabled until you start the beat.

You use the Stop button to shut down the beat generation.



Notice Start is disabled after the beat has started.

All user actions are sent to the controller.



controller

## The controller is in the middle...

The **controller** sits between the view and model. It takes your input, like selecting Start from the Control menu, and turns it into an action on the model to start the beat generation.

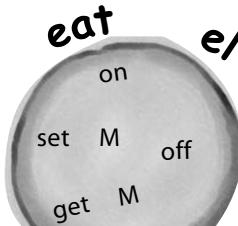
The controller takes input from the user and figures out how to translate that into requests on the model.

## Let's not forget about the model underneath it all...

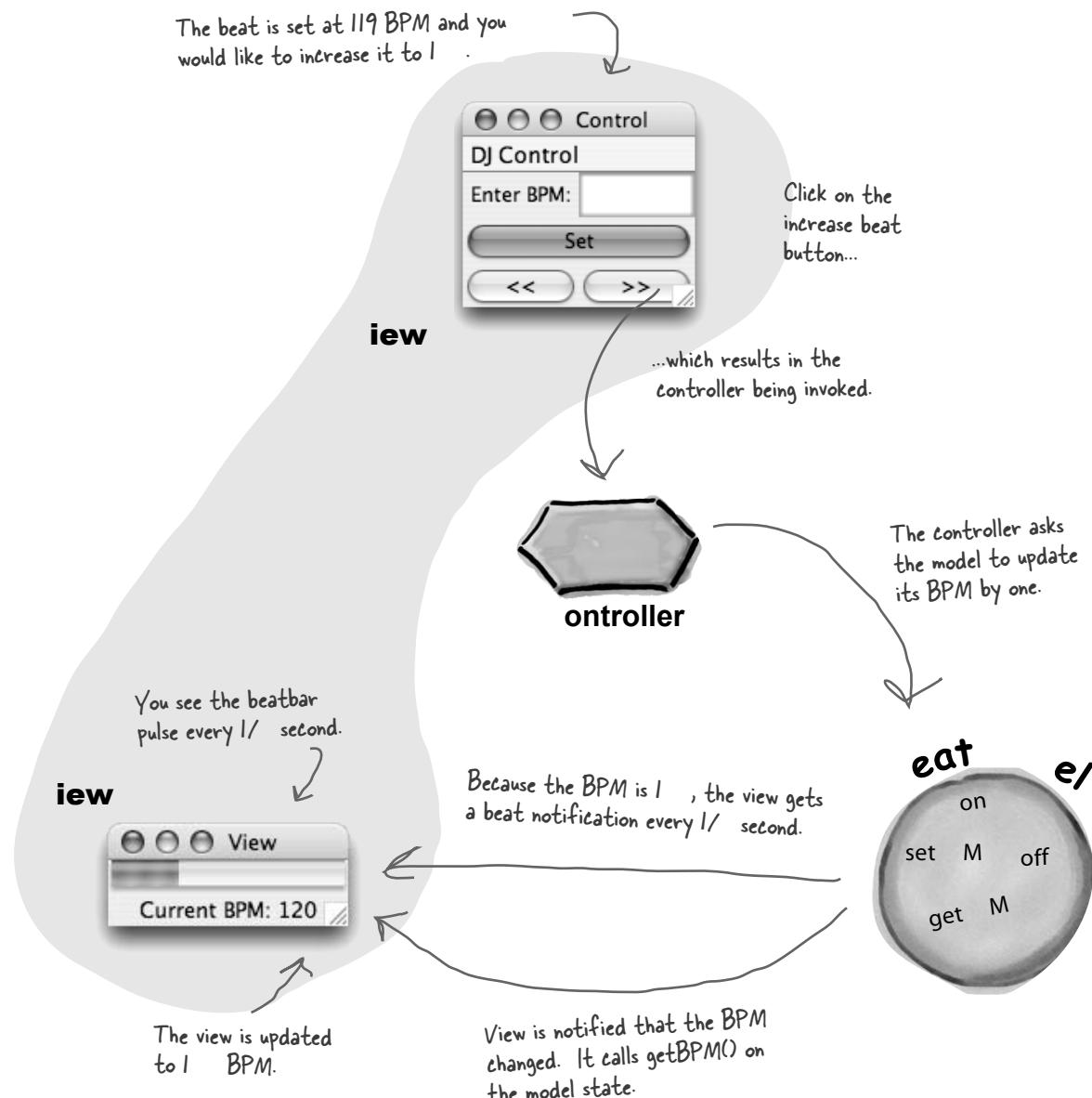
You can't see the **model**, but you can hear it. The model sits underneath everything else, managing the beat and driving the speakers with M.

The BeatModel is the heart of the application. It implements the logic to start and stop the beat, set the beats per minute (BPM), and generate the sound.

The model also allows us to obtain its current state through the getBPM() method.



## Putting the pieces together

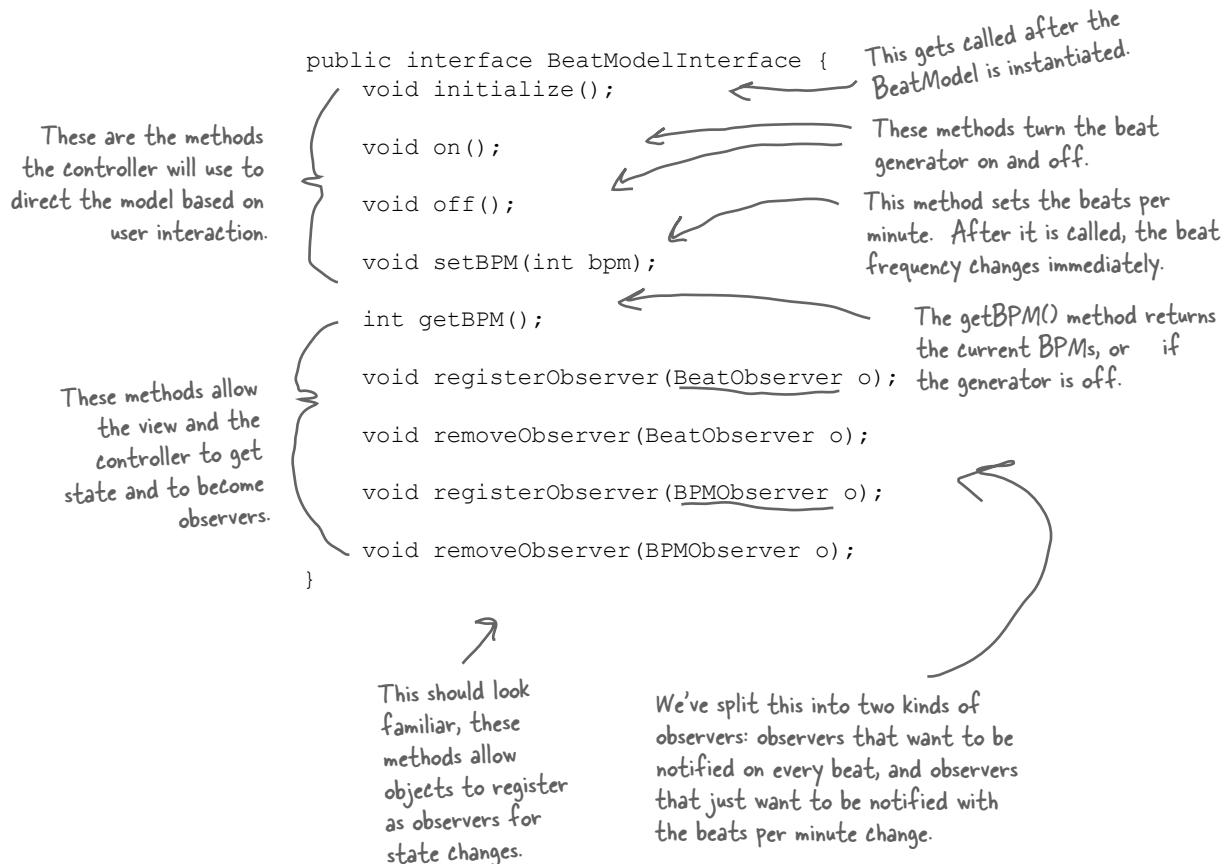


## Building the pieces

Okay, you know the model is responsible for maintaining all the data, state and any application logic. So what's the BeatModel got in it? Its main job is managing the beat, so it has state that maintains the current beats per minute and lots of code that generates M events to create the beat that we hear. It also exposes an interface that lets the controller manipulate the beat and lets the view and controller obtain the model's state.

Also, don't forget that the model uses the Observer Pattern, so we also need some methods to let objects register as observers and send out notifications.

**Let's check out the BeatModelInterface before looking at the implementation:**



## lets a eal att ec cete eat el class

We implement the BeatModelInterface.

```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // other instance variables here

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // Code to register and notify observers
    // Lots of MIDI code to handle the beat
}
```

This is needed for the MIDI code.

The sequencer is the object that knows how to generate real beats (that you can hear!).

These ArrayLists hold the two kinds of observers (Beat and BPM observers).

The bpm instance variable holds the frequency of beats – by default, 9 BPM.

This method does setup on the sequencer and sets up the beat tracks for us.

The on() method starts the sequencer and sets the BPMs to the default: 9 BPM.

And off() shuts it down by setting BPMs to 0 and stopping the sequencer.

The setBPM() method is the way the controller manipulates the beat. It does three things:

(1) Sets the bpm instance variable

(2) Asks the sequencer to change its BPMs.

(3) Notifies all BPM Observers that the BPM has changed.

The getBPM() method just returns the bpm instance variable, which indicates the current beats per minute.

The beatEvent() method, which is not in the BeatModelInterface, is called by the MIDI code whenever a new beat starts. This method notifies all BeatObservers that a new beat has just occurred.



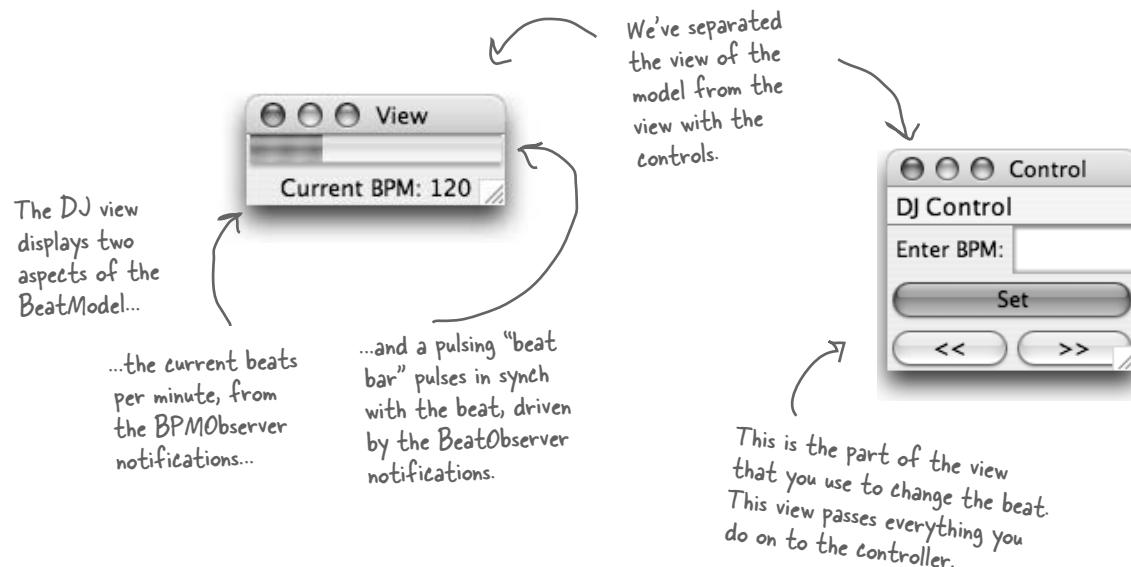
### Ready-bake Code

This model uses Java's MIDI support to generate beats. You can check out the complete implementation of all the DJ classes in the Java source files available on the [headfirstlabs.com](http://headfirstlabs.com) site, or look at the code at the end of the chapter.

# The View

Now the fun starts; we get to hook up a view and visualize the BeatModel.

The first thing to notice about the view is that we've implemented it so that it is displayed in two separate windows. One window contains the current BPM and the pulse; the other contains the interface controls. Why? We wanted to emphasize the difference between the interface that contains the view of the model and the rest of the interface that contains the set of user controls. Let's take a closer look at the two parts of the view.



Our BeatModel makes no assumption about the view. The model is implemented in the Observer Pattern, so it notifies any view registered another whenever its state changes. The view gets the model API to get access to the state. We implemented one type of view, can you think of other views that could make use of notifications and state in the BeatModel?

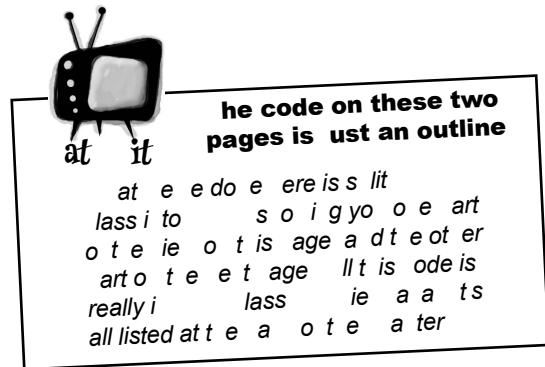
A lightshow that is based on the real-time beat.

A textual view that displays a music genre based on the BPM (ambient, downbeat, techno, etc.).

## Implementing the View

he two parts of the view—the view of the model, and the view with the user interface controls—are displayed in two windows, but live together in one ava class. We'll first show you just the code that creates the view of the model, which displays the current BPM and the beat bar.

hen we'll come back on the ne t page and show you just the code that creates the user interface controls, which displays the BPM te t entry field, and the buttons.



DJView is an observer for both real-time beats and BPM changes.

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller; ← The view holds a reference to both the model and
    JFrame viewFrame; ← the controller. The controller is only used by the
    JPanel viewPanel; ← control interface, which we'll go over in a sec...
    BeatBar beatBar; ← Here, we create a few
    JLabel bpmOutputLabel; ← components for the display.
```

```
    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    } ← The constructor gets a reference
          to the controller and the model,
          and we store references to those in
          the instance variables.

    public void createView() {
        // Create all Swing components here
    }

    public void updateBPM() { ← We also register as a BeatObserver and a
        int bpm = model.getBPM(); ← BPMObserver of the model.
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }
```

```
    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

The updateBPM() method is called when a state change occurs in the model. When that happens we update the display with the current BPM. We can get this value by requesting it directly from the model.

Likewise, the updateBeat() method is called when the model starts a new beat. When that happens, we need to pulse our "beat bar." We do this by setting it to its maximum value (1 ) and letting it handle the animation of the pulse.

## Implementing the View, continued...

Now, we'll look at the code for the user interface controls part of the view. This view lets you control the model by telling the controller what to do, which in turn, tells the model what to do. Remember, this code is in the same class file as the other view code.

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void createControls() {
        // Create all Swing components here
    }
    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }
}
```



This method creates all the controls and places them in the interface. It also takes care of the menu. When the stop or start items are chosen, the corresponding methods are called on the controller.

All these methods allow the start and stop items in the menu to be enabled and disabled. We'll see that the controller uses these to change the interface.

This method is called when a button is clicked.

If the Set button is clicked then it is passed on to the controller along with the new bpm.

Likewise, if the increase or decrease buttons are clicked, this information is passed on to the controller.

## Now for the Controller

It's time to write the missing piece—the controller. Remember the controller is the strategy that we plug into the view to give it some smarts.

Because we are implementing the Strategy Pattern, we need to start with an interface for any strategy that might be plugged into the view. We're going to call it Controller interface.

```
public interface ControllerInterface {  
    void start();  
    void stop();  
    void increaseBPM();  
    void decreaseBPM();  
    void setBPM(int bpm);  
}
```

Here are all the methods the view can call on the controller.

These should look familiar after seeing the model's interface. You can stop and start the beat generation and change the BPM. This interface is "richer" than the BeatModel interface because you can adjust the BPMs with increase and decrease.



## Design Puzzles

One event attaches view and controller together making the Strategy Pattern. Can you draw a class diagram of the two parts representing this pattern?

## And here's the implementation of the controller:

```

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}

```

The controller implements the ControllerInterface.

The controller is the creamy stuff in the middle of the MVC oreo cookie, so it is the object that gets to hold on to the view and the model and glues it all together.

The controller is passed the model in the constructor and then creates the view.

When you choose Start from the user interface menu, the controller turns the model on and then alters the user interface so that the start menu item is disabled and the stop menu item is enabled.

Likewise, when you choose Stop from the menu, the controller turns the model off and alters the user interface so that the stop menu item is disabled and the start menu item is enabled.

If the increase button is clicked, the controller gets the current BPM from the model, adds one, and then sets a new BPM.

Same thing here, only we subtract one from the current BPM.

Finally, if the user interface is used to set an arbitrary BPM, the controller instructs the model to set its BPM.

NOTE: the controller is making the intelligent decisions for the view. The view just knows how to turn menu items on and off; it doesn't know the situations in which it should disable them.

## Putting it all together...

We've got everything we need - a model, a view, and a controller. Now it's time to put them all together into a MVC. We're going to see and hear how well they work together.

All we need is a little code to get things started; it won't take much:

```
public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

A handwritten-style diagram with arrows. An arrow points from the text "First create a model..." to the line "BeatModelInterface model = new BeatModel();". Another arrow points from the text "...then create a controller and pass it the model. Remember, the controller creates the view, so we don't have to do that." to the line "ControllerInterface controller = new BeatController(model);". A final arrow points from the text "Run this..." to the command "% java DJTestDrive".



## And now for a test run...

```
File Edit Window Help LetT eBa Kic
% java DJTestDrive
%
```

## Things to do

- ➊ Start the eat generation with the start menu item. Notice the controller disables the item afterwards.
- ➋ Set the tempo entry along with the increase and decrease buttons to change the BPM. Notice how the view displays reflects the changes despite the fact that it has no logical link to the controls.
- ➌ Notice how the eat bar always keeps up with the eat since it's an observer of the model.
- ➍ Put on your favorite song and see if you can eat through the song by using the increase and decrease controls.
- ➎ Stop the generator. Notice how the controller disables the stop menu item and enables the start menu item.

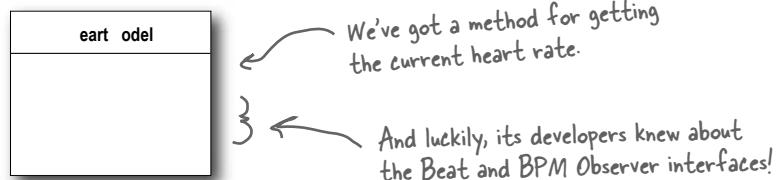
...and you'll see this.



## Exploring Strategy

Let's take the strategy Pattern just a little further to get a better feel for how it is used in M C. We're going to see another friendly pattern pop up too - a pattern you'll often see hanging around the M C trio - the Adapter Pattern.

Think for a second about what the view does. It displays a beat rate and a pulse. Does that sound like something else? How about a heartbeat? It just so happens we happen to have a heart monitor class; here's the class diagram



It certainly would be nice to re-use our current view with the HeartModel, but we need a controller that works with this model. Although the interface of the HeatModel doesn't match what the view expects because it already has a `getHeartRate()` method rather than a `getBPM()`. How would you design a new class to allow the view to be reused with the new model?

## Adapting the Model

For starters, we're going to need to adapt the HeartModel to a BeatModel. If we don't, the view won't be able to work with the model, because the view only knows how to getBPM(), and the equivalent heart model method is getHeartRate(). How are we going to do this? We're going to use the Adapter Pattern, of course. It turns out that this is a common technique when working with the MVC to use an adapter to adapt a model to work with existing controllers and views.

Here's the code to adapt a HeartModel to a BeatModel

```
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }
    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

The code is annotated with several hand-drawn arrows and text boxes:

- An arrow points from the first line to the text: "We need to implement the target interface, in this case, BeatModelInterface."
- An arrow points from the assignment of 'heart' to the text: "Here, we store a reference to the heart model."
- Arrows point from the 'on()' and 'off()' methods to the text: "We don't know what these would do to a heart, but it sounds scary. So we'll just leave them as 'no ops.'"
- An arrow points from the 'getBPM()' method to the text: "When getBPM() is called, we'll just translate it to a getHeartRate() call on the heart model."
- An arrow points from the 'setBPM()' method to the text: "We don't want to do this on a heart! Again, let's leave it as a 'no op'."
- A curly brace groups the observer methods ('registerObserver', 'removeObserver', 'registerObserver', 'removeObserver') with the text: "Here are our observer methods. We just delegate them to the wrapped heart model."

# Now we're ready for a HeartController

With our Heart chapter in hand we should be ready to create a controller and get the view running with the HeartModel. Talk about reuse.

```
public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

The HeartController implements the ControllerInterface, just like the BeatController did.

Like before, the controller creates the view and gets everything glued together.

There is one change: we are passed a HeartModel, not a BeatModel...

...and we need to wrap that model with an adapter before we hand it to the view.

Finally, the HeartController disables the menu items as they aren't needed.

There's not a lot to do here; after all, we can't really control hearts like we can beat machines.

## And that's it Now it's time for some test code...

```
public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

All we need to do is create the controller and pass it a heart monitor.

*you are here ▶*

te t the heart ode

## And now for a test run...

```
File Edit Window Help C ec MyP I e
% java HeartTestDrive
%
```

←  
Run this...

...and you'll see this.



## Things to do

- ➊ Notice that the display works great with a heart beat. The heart rate is constant because the Heart model also supports P and eat servers we can get eat updates just like with the D eats.
- ➋ Notice that the heart beat has natural variation, notice the display is updated with the new beats per minute.
- ➌ Each time we get a P update the adapter is doing its job of translating get P calls to getHeart rate calls.
- ➍ The start and stop buttons are not enabled because the controller disabled them.
- ➎ The other buttons still work but have no effect because the controller implements no ops for them. The view could be changed to support the disabling of these items.



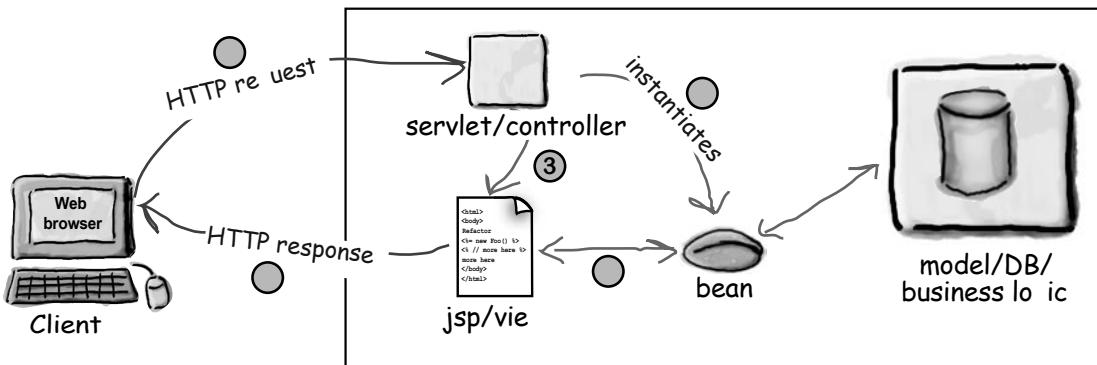
↑  
Nice healthy heart rate.

ha ter

# MVC and the Web

t wasn't long after the Web was spun that developers started adapting the M C to fit the browser server model. The prevailing adaptation is known simply as Model and uses a combination of servlet and JSP technology to achieve the same separation of model, view and controller that we see in conventional G s.

Let's check out how Model works



### a e an re est c srece e b a ser let

Using your web browser you make an HTTP request. This typically involves sending along some form data, like your username and password. A servlet receives this form data and parses it.



### e ser let acts as t e c ntr iler

The servlet plays the role of the controller and processes your request, most likely making requests on the model (usually a database). The result of processing the request is usually bundled up in the form of a JavaBean.



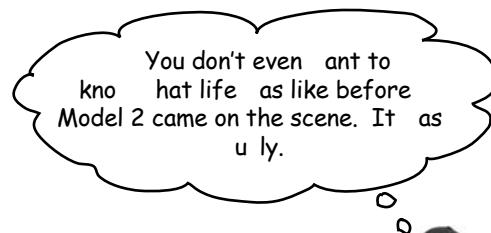
### e c ntr iler r ar s c ntr l t t e e

The View is represented by a JSP. The JSP's only job is to generate the page representing the view of model (which it obtains via the JavaBean) along with any controls needed for further actions.



### e e ret rns a pa e t t e br ser a

A page is returned to the browser, here it is displayed as the view. The user submits further requests, which are processed in the same fashion.



## **odel is more than just a clean design**

The benefits of the separation of the view, model and controller are pretty clear to you now. But you need to know the rest of the story with Model 2 that it saved many web shops from sinking into chaos.

How? Well, Model 2 not only provides a separation of components in terms of design, it also provides a separation in *permissions*. Let's face it, in the old days, anyone with access to your JSPs could get in and write any Java code they wanted, right? And that included a lot of people who didn't know a jar file from a jar of peanut butter. The reality is that most web producers were terrible at their job.

Luckily Model 2 came to the rescue. With Model 2 we can leave the developer jobs to the guys and girls who know their servlets and let the web producers loose on simple Model 2 style JSPs where all the producers have access to is HTML and simple JavaBeans.



former DOT COM'er

## Model 2: D 'ing from a cell phone

You didn't think we'd try to skip out without moving that great BeatModel over to the Web did you? Just think, you can control your entire session through a web page on your cellular phone. Now you can get out of that booth and get down in the crowd. What are you waiting for? Let's write that code.

### The plan



#### ① p t e el

Well, actually, we don't have to fix the model, it's fine just like it is!

#### ② Create a servlet controller

We need a simple servlet that can receive our HTTP requests and perform a few operations on the model. All it needs to do is stop, start and change the beats per minute.

#### ③ Create a view

We'll create a simple view with a JSP. It's going to receive a JavaBean from the controller that will tell it everything it needs to display. The JSP will then generate an HTML interface.



ee S

### Setting up your Servlet environment

It's aarta a aace or to cat

ea r t er et



you are here ▶

## Step one: the model

member that in M-C, the model doesn't know anything about the views or controllers. In other words it is totally decoupled. All it knows is that it may have observers it needs to notify. That's the beauty of the Observer Pattern. It also provides an interface the views and controllers can use to get and set its state.

Now all we need to do is adapt it to work in the web environment, but, given that it doesn't depend on any outside classes, there is really no work to be done. We can use our BeatModel off the shelf without changes. So, let's be productive and move on to step two.

## Step two: the controller servlet

Remember, the servlet is going to act as our controller; it will receive Web browser input in a HTTP request and translate it into actions that can be applied to the model.

Then, given the way the Web works, we need to return a view to the browser. To do this we'll pass control to the view, which takes the form of a JSP. We'll get to that in step three.

Here's the outline of the servlet; on the next page, we'll look at the full implementation.

```
public class DJView extends HttpServlet {  
  
    public void init() throws ServletException {  
        BeatModel beatModel = new BeatModel();  
        beatModel.initialize();  
        getServletContext().setAttribute("beatModel", beatModel);  
    }  
  
    // doPost method here  
  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException, ServletException  
    {  
        // implementation here  
    }  
}
```

We extend the HttpServlet class so that we can do servlet kinds of things, like receive HTTP requests.

Here's the init method; this is called when the servlet is first created.

We first create a BeatModel object...

...and place a reference to it in the servlet's context so that it's easily accessed.

Here's the doGet() method. This is where the real work happens. We've got its implementation on the next page.

Here's the implementation of the doGet() method from the page before

```

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
throws IOException, ServletException
{
    BeatModel beatModel =
        (BeatModel) getServletContext().getAttribute("beatModel");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = beatModel.getBPM() + "";
    }

    String set = request.getParameter("set");
    if (set != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        beatModel.setBPM(bpmNumber);
    }

    String decrease = request.getParameter("decrease");
    if (decrease != null) {
        beatModel.setBPM(beatModel.getBPM() - 1);
    }
    String increase = request.getParameter("increase");
    if (increase != null) {
        beatModel.setBPM(beatModel.getBPM() + 1);
    }
    String on = request.getParameter("on");
    if (on != null) {
        beatModel.start();
    }
    String off = request.getParameter("off");
    if (off != null) {
        beatModel.stop();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/DJView.jsp");
    dispatcher.forward(request, response);
}

```

First we grab the model from the servlet context. We can't manipulate the model without a reference to it.

Next we grab all the HTTP commands/parameters...

If we get a set command, then we get the value of the set, and tell the model.

To increase or decrease, we get the current BPMs from the model, and adjust up or down by one.

If we get an on or off command, we tell the model to start or stop.

Finally, our job as a controller is done. All we need to do is ask the view to take over and create an HTML view.

Following the Model definition, we pass the JSP a bean with the model state in it. In this case, we pass it the actual model, since it happens to be a bean.

## Now we need a view...

All we need is a view and we've got our browser based beat generator ready to go. In Model P, the view is just a P. All the P knows about is the bean it receives from the controller. In our case, that bean is just the model and the P is only going to use its BPM property to extract the current beats per minute. With that data in hand, it creates the view and also the user interface controls.

```
<jsp:useBean id="beatModel" scope="request" class="headfirst.combined.djview.BeatModel" />

<html>
    <head>
        <title>DJ View</title>
    </head>
    <body>

        <h1>DJ View</h1>
        Beats per minutes = <jsp:getProperty name="beatModel" property="BPM" />
        <br />
        <hr>
        <br />

        <form method="post" action="/djview/servlet/DJView">
            BPM: <input type="text" name="bpm"
                value="
```

Beginning of the HTML.

Here we use the model bean to extract the BPM property.

Now we generate the view, which prints out the current beats per minute.

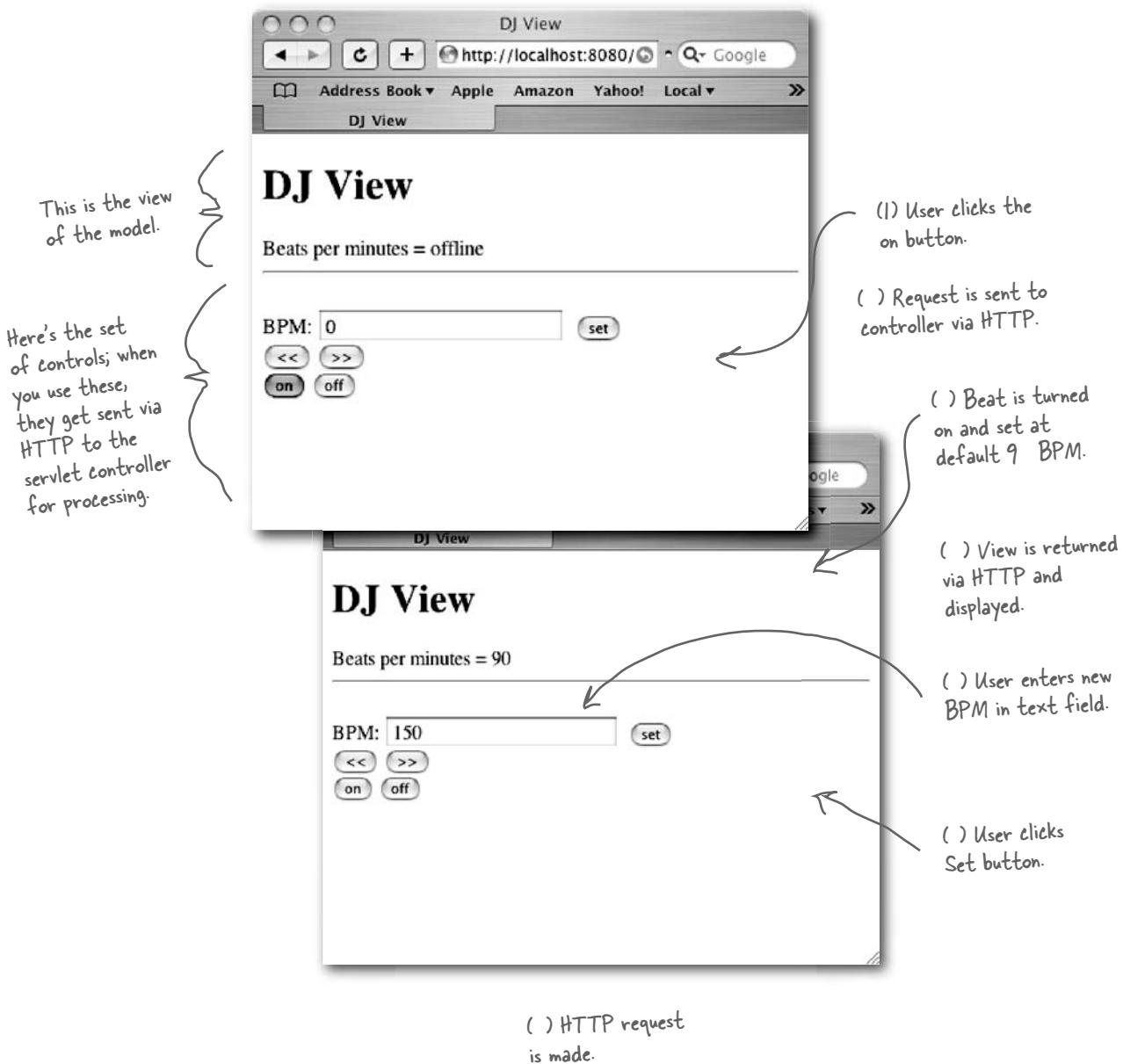
And here's the control part of the view. We have a text entry for entering a BPM along with increase/decrease and on/off buttons.

And here's the end of the HTML.

NOTICE that just like MVC, in Model the view doesn't alter the model (that's the controller's job); all it does is use its state!

## Putting Model 2 to the test...

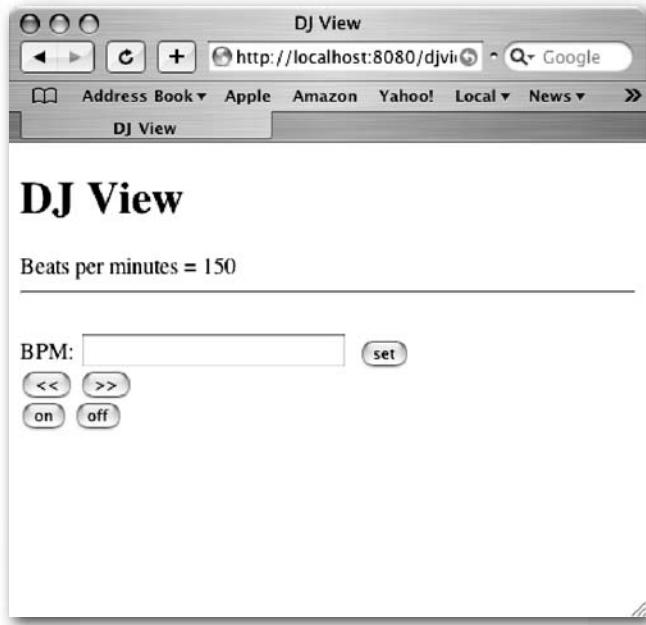
It's time to start your web browser, hit the view servlet and give the system a spin...



you are here ▶

(8) Controller  
changes model to  
1 BPMs

(9) View returns  
HTML reflecting  
the current model.



## Things to do

- ➊ First, hit the **w** page you'll see the **eats per inute at o ahead and clic the on utton**
- ➋ Now you should see the **eats per inute at the default setting P ou** should also hear a **eat on the machine the server is running on**
- ➌ nter a speci c eat, say, , and clic the set utton the page should refresh with a **eats per inute of and you should hear the eat increase**
- ➍ Now play with the increase decrease uttons to ad ust the eat up and down
- ➎ hin a out how each step of the syste wor s he H interface a es a re uest to the servlet the controller the servlet parses the user input and then a es re uests to the odel he servlet then passes control to the P the view , which creates the H view that is returned and displayed

# Design Patterns and Model 2

fter implementing the Control for the Web using Model , you might be wondering where the patterns went. We have a view created in HML from a P but the view is no longer a listener of the model. We have a controller that's a servlet that receives HTTP requests, but are we still using the Strategy Pattern? And what about Composite? We have a view that is made from HML and displayed in a web browser. Is that still the Composite Pattern?

## Model is an adaptation of to the Web

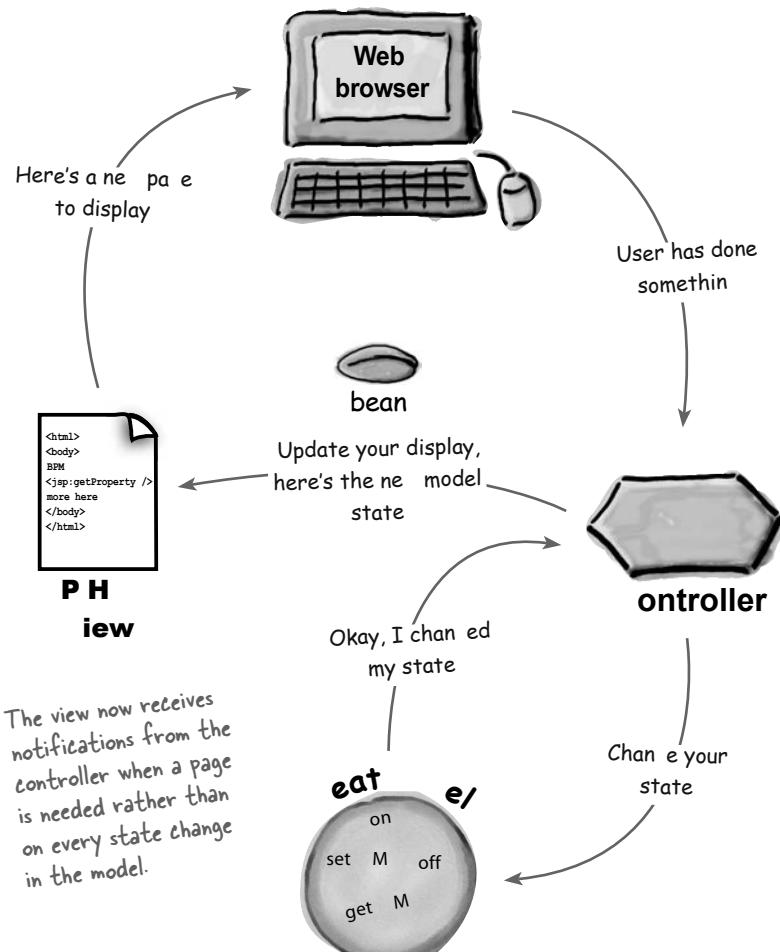
Even though Model doesn't look exactly like the book MVC, all the parts are still there; they've just been adapted to reflect the idiosyncrasies of the web browser model. Let's take another look...

## Observer

The view is no longer an observer of the model in the classic sense; that is, it doesn't register with the model to receive state changes notifications.

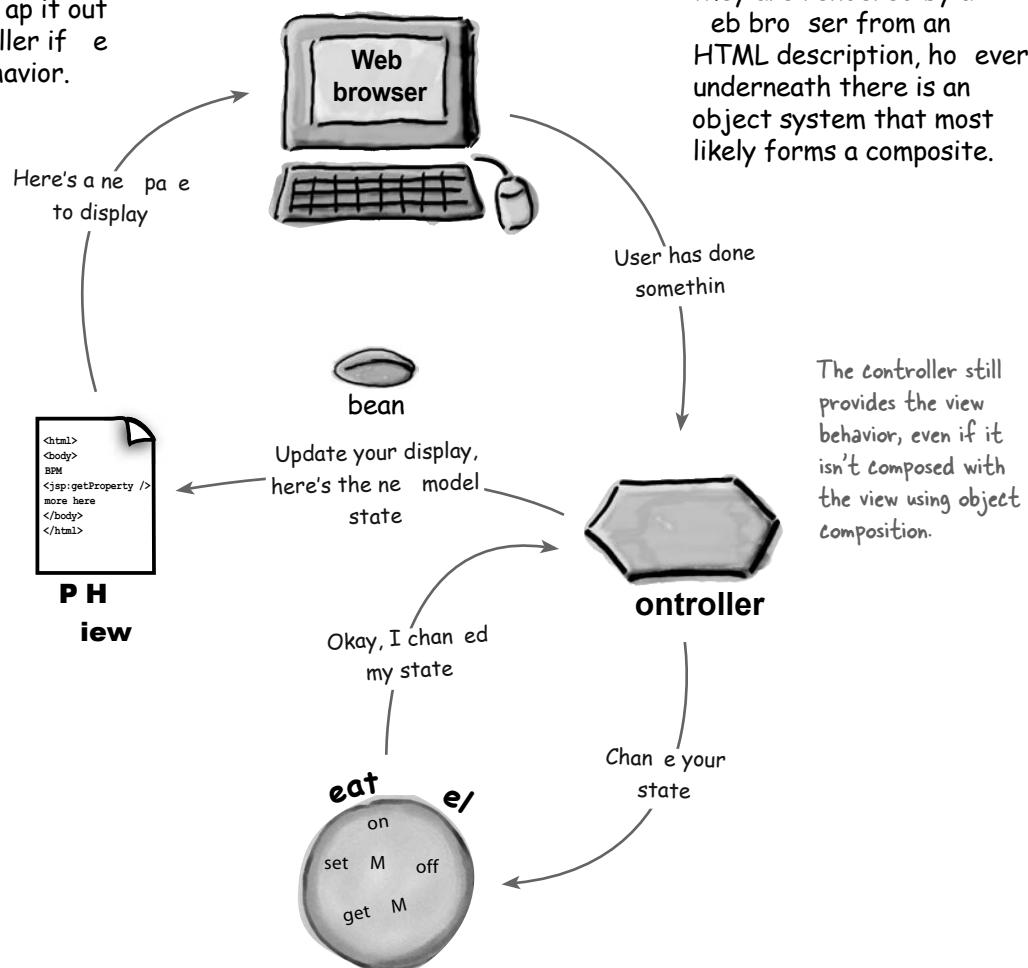
However, the view does receive the equivalent of notifications indirectly from the controller when the model has been changed. The controller even passes the view a bean that allows the view to retrieve the model's state.

If you think about the browser model, the view only needs an update of state information when an HTTP response is returned to the browser; notifications at any other time would be pointless. Only when a page is being created and returned does it make sense to create the view and incorporate the model's state.



## Strategy

In Model 2, the Strategy object is still the controller servlet; however, it's not directly composed with the view in the classic manner. That said, it is an object that implements behavior for the view, and we can swap it out for another controller if we want different behavior.



## Composite

Like our S in GUI, the view is ultimately made up of a nested set of graphical components. In this case, they are rendered by a web browser from an HTML description, however underneath there is an object system that most likely forms a composite.

The controller still provides the view behavior, even if it isn't composed with the view using object composition.

**Q:** It seems like you are really hand waving the fact that the Composite Pattern is really in there? Is it really there?

**A:**

## there are no Dumb Questions

**Q:** Does the view always have to ask the model for its state? Couldn't we use the push model and send the model's state with the update notification?

**A:**

**Q:** You've talked a lot about the state of the model. Does this mean it has the State Pattern in it?

**A:**

**Q:** Does the controller ever implement any application logic?

**A:**

**Q:** I've seen descriptions of the where the controller is described as a mediator between the view and the model. Is the controller implementing the Mediator Pattern?

**A:**

**Q:** If I have more than one view do I always need more than one controller?

**A:**

**Q:** The view is not supposed to manipulate the model, however I noticed in your implementation that the view has full access to the methods that change the model's state. Is this dangerous?

**A:**

**Q:** I've always found the word model hard to wrap my head around. I now get that it's the guts of the application but why was such a vague, hard to understand word used to describe this aspect of the ?



## Tools for your Design Toolbox

You could impress anyone with your design tool now, look at all those principles, patterns and now, compound patterns

### OO Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions. Do not depend on concrete classes.
- Only talk to your friends.
- Don't call us, we'll call you.
- A class should have only one reason to change.

### OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

### OO Patterns

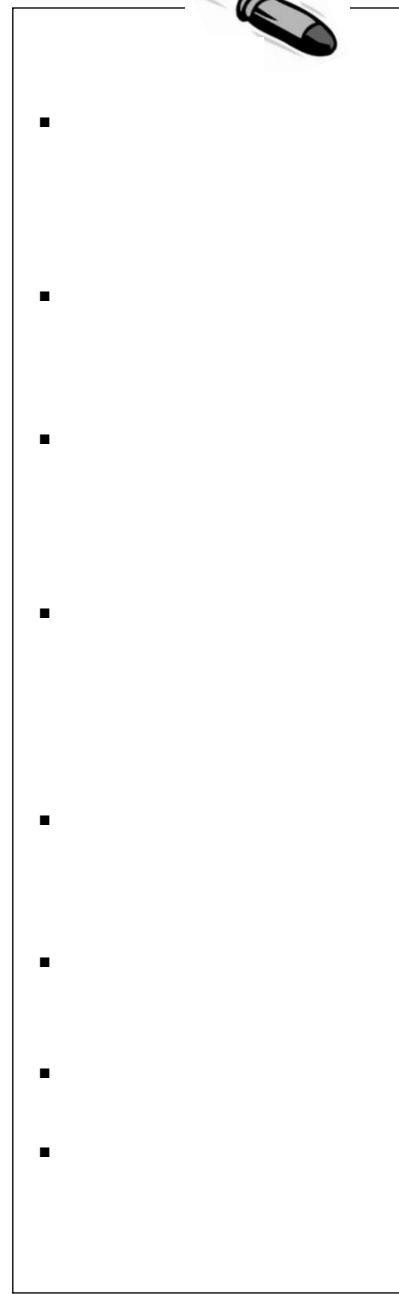
See in vi

Proxy - Provide a surrogate or placeholder for another object to control access to it.

#### Compound Patterns

A Compound Pattern combines two or more patterns into a solution that solves a recurring or general problem.

We have a new category! MVC and Model are compound patterns.





# er ise solutions



## Sharpen your pencil

The QuackCounter is a Quackable too. When we change Quackable to extend QuackObservable, we have to change every class that implements Quackable, including QuackCounter:

QuackCounter is a Quackable, so now it's a QuackObservable too.

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter(Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }

    public void registerObserver(Observer observer) {
        duck.registerObserver(observer);
    }

    public void notifyObservers() {
        duck.notifyObservers();
    }
}
```

Here's the duck that the QuackCounter is decorating. It's this duck that really needs to handle the observable methods.

All of this code is the same as the previous version of QuackCounter.

Here are the two QuackObservable methods. Notice that we just delegate both calls to the duck that we're decorating.



## Sharpen your pencil

What or Q ac olo i t want to o er e an entire oc ? W at doe t at mean anyway? T in a o titli et i :i we o er e a compo ite, t en we re o er in e eryt in i t e compo ite. So, w en yo re i terwit a oc , t e oc compo ite ma e reyo etre i tered wit all it c ildren, w ic may incl de ot er oc .

```
public class Flock implements Quackable {  
    ArrayList ducks = new ArrayList();  
  
    public void add(Quackable duck) {  
        ducks.add(duck);  
    }  
  
    public void quack() {  
        Iterator iterator = ducks.iterator();  
        while (iterator.hasNext()) {  
            Quackable duck = (Quackable) iterator.next();  
            duck.quack();  
        }  
    }  
  
    public void registerObserver(Observer observer) {  
        Iterator iterator = ducks.iterator();  
        while (iterator.hasNext()) {  
            Quackable duck = (Quackable) iterator.next();  
            duck.registerObserver(observer);  
        }  
    }  
  
    public void notifyObservers() { }  
}
```

Flock is a Quackable, so now  
it's a QuackObservable too.

Here's the Quackables that  
are in the Flock.

When you register as an Observer  
with the Flock, you actually  
get registered with everything  
that's IN the flock, which is  
every Quackable, whether it's a  
duck or another Flock.

We iterate through all the  
Quackables in the Flock and  
delegate the call to each  
Quackable. If the Quackable  
is another Flock, it will do  
the same.

Each Quackable does its own notification,  
so Flock doesn't have to worry about it.  
This happens when Flock delegates quack()  
to each Quackable in the Flock.



## Sharpen your pencil

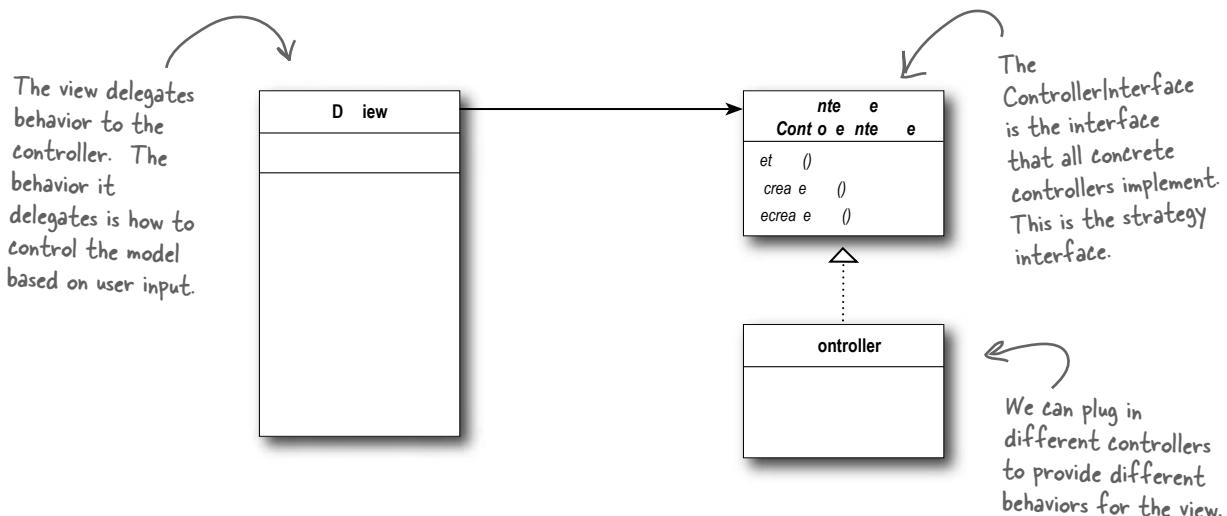
You're still directly instantiating these by relying on concrete classes. Can you write an abstract actor for these? How should it handle creating goose ducks?

You could add a `createGooseDuck()` method to the existing Duck Factories. Or, you could create a completely separate Factory for creating families of Geese.



## Design Class

One common pattern is View and Controller to either make use of the Strategy Pattern. Can you draw a class diagram of the two patterns?



*ready a e ode the d a i ation*



## Ready-bake Code

Here's the complete implementation of the DJView. It shows all the MIDI code to generate the sound, and all the Swing components to create the view. You can also download this code at <http://www.headfirstlabs.com>. Have fun!

```
package headfirst.combined.djview;

public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel ();
        ControllerInterface controller = new BeatController (model);
    }
}
```

*e eat el*

```
package headfirst.combined.djview;

public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```

package headfirst.combined.djview;

import javax.sound.midi.*;
import java.util.*;
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // other instance variables here
    Sequence sequence;
    Track track;

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {

```

**ready a e ode ode**



## Ready-bake Code

```
        BeatObserver observer = (BeatObserver)beatObservers.get(i);
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

public void notifyBPMObservers() {
    for(int i = 0; i < bpmObservers.size(); i++) {
        BPMObserver observer = (BPMObserver)bpmObservers.get(i);
        observer.updateBPM();
    }
}

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

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

public void meta(MetaMessage message) {
    if (message.getType() == 47) {
        beatEvent();
        sequencer.start();
        setBPM(getBPM());
    }
}

public void setUpMidi() {
    try {
        sequencer = MidiSystem.getSequencer();
```

```

sequencer.open();
sequencer.addMetaEventListener(this);
sequence = new Sequence(Sequence.PPQ, 4);
track = sequence.createTrack();
sequencer.setTempoInBPM(getBPM());
} catch(Exception e) {
    e.printStackTrace();
}
}

public void buildTrackAndStart() {
    int[] trackList = {35, 0, 46, 0};

    sequence.deleteTrack(null);
    track = sequence.createTrack();

    makeTracks(trackList);
    track.add(makeEvent(192, 9, 1, 0, 4));
    try {
        sequencer.setSequence(sequence);
    } catch(Exception e) {
        e.printStackTrace();
    }
}

public void makeTracks(int[] list) {

    for (int i = 0; i < list.length; i++) {
        int key = list[i];

        if (key != 0) {
            track.add(makeEvent(144, 9, key, 100, i));
            track.add(makeEvent(128, 9, key, 100, i+1));
        }
    }
}

public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

    } catch(Exception e) {
        e.printStackTrace();
    }
    return event;
}
}

```

*ready a e ode ie*

## The View

## Ready-bake Code



```
package headfirst.combined.djview;

public interface BeatObserver {
    void updateBeat();
}

package headfirst.combined.djview;

public interface BPMObserver {
    void updateBPM();
}

package headfirst.combined.djview;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DJView implements ActionListener, BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;
    JFrame controlFrame;
    JPanel controlPanel;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
```

```

// Create all Swing components here
viewPanel = new JPanel(new GridLayout(1, 2));
viewFrame = new JFrame("View");
viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
viewFrame.setSize(new Dimension(100, 80));
bpmOutputLabel = new JLabel("offline", SwingConstants.CENTER);
beatBar = new BeatBar();
beatBar.setValue(0);
JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
bpmPanel.add(beatBar);
bpmPanel.add(bpmOutputLabel);
viewPanel.add(bpmPanel);
viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
viewFrame.pack();
viewFrame.setVisible(true);
}

public void createControls() {
    // Create all Swing components here
    JFrame.setDefaultLookAndFeelDecorated(true);
    controlFrame = new JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ Control");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.stop();
            //bpmOutputLabel.setText("offline");
        }
    });
    JMenuItem exit = new JMenuItem("Quit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
}

```

*ready a e ode ie*



## Ready-bake Code

```
menu.add(exit);
menuBar.add(menu);
controlFrame.setJMenuBar(menuBar);

bpmTextField = new JTextField(2);
bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
setBPMButton = new JButton("Set");
setBPMButton.setSize(new Dimension(10, 40));
increaseBPMButton = new JButton(">>");
decreaseBPMButton = new JButton("<<");
setBPMButton.addActionListener(this);
increaseBPMButton.addActionListener(this);
decreaseBPMButton.addActionListener(this);

 JPanel buttonPanel = new JPanel(new GridLayout(1, 2));

buttonPanel.add(decreaseBPMButton);
buttonPanel.add(increaseBPMButton);

 JPanel enterPanel = new JPanel(new GridLayout(1, 2));
enterPanel.add(bpmLabel);
enterPanel.add(bpmTextField);
 JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
insideControlPanel.add(enterPanel);
insideControlPanel.add(setBPMButton);
insideControlPanel.add(buttonPanel);
controlPanel.add(insideControlPanel);

bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

controlFrame.getRootPane().setDefaultButton(setBPMButton);
controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

controlFrame.pack();
controlFrame.setVisible(true);
}

public void enableStopMenuItem() {
    stopMenuItem.setEnabled(true);
}

public void disableStopMenuItem() {
    stopMenuItem.setEnabled(false);
}
```

*ha ter*

```

}

public void enableStartMenuItem() {
    startMenuItem.setEnabled(true);
}

public void disableStartMenuItem() {
    startMenuItem.setEnabled(false);
}

public void actionPerformed(ActionEvent event) {
    if (event.getSource() == setBPMButton) {
        int bpm = Integer.parseInt(bpmTextField.getText());
        controller.setBPM(bpm);
    } else if (event.getSource() == increaseBPMButton) {
        controller.increaseBPM();
    } else if (event.getSource() == decreaseBPMButton) {
        controller.decreaseBPM();
    }
}

public void updateBPM() {
    int bpm = model.getBPM();
    if (bpm == 0) {
        bpmOutputLabel.setText("offline");
    } else {
        bpmOutputLabel.setText("Current BPM: " + model.getBPM());
    }
}

public void updateBeat() {
    beatBar.setValue(100);
}
}

```

## The Controller

```

package headfirst.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}

```

*you are here ▶*

*ready a e o de ontro er*

## Ready-bake Code



```
package headfirst.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

*ha ter*

## The Heart Model

```

package headfirst.combined.djview;

public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}

package headfirst.combined.djview;
public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}

package headfirst.combined.djview;
import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
            if (rate < 120 && rate > 50) {
                time += change;
            }
            for(BeatObserver o : beatObservers) {
                o.update();
            }
            for(BPMObserver o : bpmObservers) {
                o.update();
            }
        }
    }
}

```

*ready a e ode heart eat ode*

```
        notifyBeatObservers();
        if (rate != lastrate) {
            lastrate = rate;
            notifyBPMObservers();
        }
    }
    try {
        Thread.sleep(time);
    } catch (Exception e) {}
}
}

public int getHeartRate() {
    return 60000/time;
}

public void registerObserver(BeatObserver o) {
    beatObservers.add(o);
}

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

public void notifyBeatObservers() {
    for(int i = 0; i < beatObservers.size(); i++) {
        BeatObserver observer = (BeatObserver)beatObservers.get(i);
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

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

public void notifyBPMObservers() {
    for(int i = 0; i < bpmObservers.size(); i++) {
        BPMObserver observer = (BPMObserver)bpmObservers.get(i);
        observer.updateBPM();
    }
}
}
```

## Ready-bake Code



*ha ter*

## The Heart Adapter

```
package headfirst.combined.djview;
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

*you are here ▶*

*ready a e o de heart eat ontro er*

## The Controller

## Ready-bake Code



```
package headfirst.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

etter i in ith atterns

# Patterns in the real world



B t, e ore yo o openin all t o e new door o opport nity, we need to co er a ew detail t at yo ll enco nter o t in t e real world t at ri t,t in et a little more comple t an t ey are ere in O ect ille. Come alon , we e ot a nice ide to elp yo t ro t e tran ition on t e ne tpa e...

# The Objectville Guide to Better Living with Design Patterns



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

*Learn the all too common misconceptions about the definition of a "Design Pattern."*

*Discover those nifty Design Pattern Catalogs and why you just have to get one.*

*Avoid the embarrassment of using a Design Pattern at the wrong time.*

*Learn how to keep patterns in classifications where they belong.*

*See that discovering patterns isn't just for the gurus; read our quick HowTo and become a patterns writer too.*

*Be there when the true identity of the mysterious Gang of Four is revealed.*

*Keep up with the neighbors – the coffee table books any patterns user must own.*

*Learn to train your mind like a Zen master.*

*Win friends and influence developers by improving your patterns vocabulary.*

# Design Pattern defined

We bet you've got a pretty good idea of what a pattern is after reading this book. But we've never really given a definition for a Design Pattern. Well, you might be a bit surprised by the definition that is in common use

**atter** is a solution to a problem in a context.

hat's not the most revealing definition is it? But don't worry, we're going to step through each of these parts, context, problem and solution

he **co** **te** **t** is the situation in which the pattern applies. This should be a recurring situation.

he **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

he **solutio**n is what you're after a general design that anyone can apply which resolves the goal and set of constraints.

Example: You have a collection of objects.

You need to step through the objects without exposing the collection's implementation.

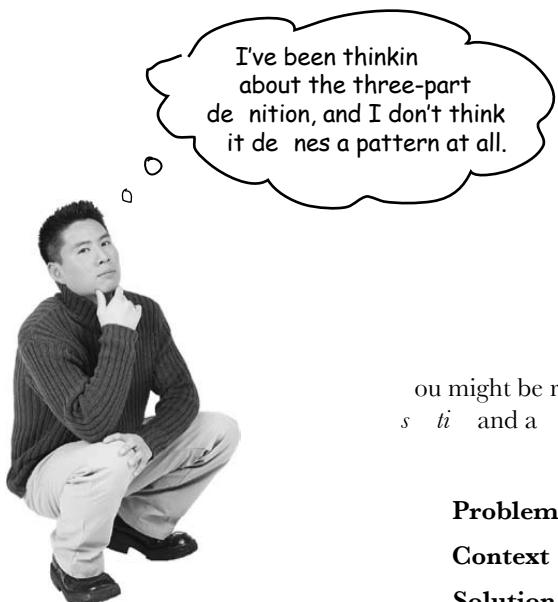
Encapsulate the iteration into a separate class.

This is one of those definitions that takes a while to sink in, but take it one step at a time. Here's a little mnemonic you can repeat to yourself to remember it

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

ow, this seems like a lot of work just to figure out what a Design Pattern is. After all, you already know that a Design Pattern gives you a solution to a common recurring design problem. What is all this formality getting you? Well, you're going to see that by having a formal way of describing patterns we can create a lot of patterns, which has all kinds of benefits.

*de i n attern de ned*



You might be right; let's think about this a bit... We need a *rem*, a *st* and a *te t*

**Problem** How do get to work on time?

**Context** I've locked my keys in the car.

**Solution** Break the window, get in the car, start the engine and drive to work.

We have all the components of the definition we have a problem, which includes the goal of getting to work, and the constraints of time, distance and probably some other factors. We also have a context in which the keys to the car are inaccessible. And we have a solution that gets us to the keys and resolves both the time and distance constraints. We must have a pattern now right?



We followed the Definition and defined a problem, a context, and a solution (which worked!). I tried a pattern? I not, how did it fail? Could we fail the same way when defining an OO Design Pattern?

*ha ter*

## ooking more closely at the Design Pattern definition

ur example does seem to match the Design Pattern definition, but it isn't a true pattern. Why? For starters, we know that a pattern needs to apply to a recurring problem. While an absent minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint cost).

t also fails in a couple of other ways first, it isn't easy to take this description, hand it to someone and have him apply it to his own unique problem.

second, we've violated an important but simple aspect of a pattern we haven't even given it a name. Without a name, the pattern doesn't become part of a vocabulary that can be shared with other developers.

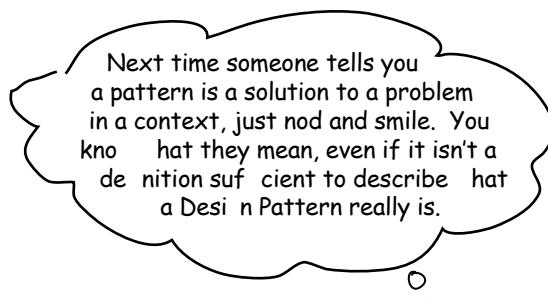
Luckily, patterns are not described and documented as a simple problem, context and solution; we have much better ways of describing patterns and collecting them together into *tter s t gs*.

**Q:** Am I going to see pattern descriptions that are stated as a problem a context and a solution?

**A:**

**Q:** Is it okay to slightly alter a pattern's structure to fit my design? Or am I going to have to go by the strict definition?

**A:**



**Q:** Where can I get a patterns catalog?

**A:**

e e after  
e e to e a e ect re te  
o t are

you are here ▶

*or e    oa    on taint*



ee      S

May the force be with you

*ro e  
oa      eto co tra t*

—

*ha ter*



Fill us in, Jim. I've just been learning patterns by reading a few articles here and there.

sure, each pattern catalog takes a set of patterns and describes each in detail along with its relationship to the other patterns.

Are you saying there is more than one patterns catalog?

Of course; there are catalogs for fundamental design Patterns and there are also catalogs on domain specific patterns, like B patterns.

Which catalog are you looking at?

This is the classic GoF catalog; it contains fundamental design Patterns.

GoF?

Right, that stands for the Gang of Four. The Gang of Four are the guys that put together the first patterns catalog.

What's in the catalog?

There is a set of related patterns. For each pattern there is a description that follows a template and spells out a lot of details of the pattern. For instance, each pattern has a *me*.

Wow, that's earth shattering a name imagine that.

Hold on Frank; actually, the name is really important. When we have a name for a pattern, it gives us a way to talk about the pattern; you know, that whole shared vocabulary thing.

Okay, was just kidding. Go on, what else is there?

Well, like was saying, every pattern follows a template. For each pattern we have a name and a few sections that tell us more about the pattern. For instance, there is an intent section that describes what the pattern is, kind of like a definition. Then there are Motivation and applicability sections that describe when and where the pattern might be used.

What about the design itself?

here are several sections that describe the class design along with all the classes that make it up and what their roles are. here is also a section that describes how to implement the pattern and often sample code to show you how.

It sounds like they've thought of everything.

here's more. here are also examples of where the pattern has been used in real systems as well as what think is one of the most useful sections how the pattern relates to other patterns.

h, you mean they tell you things like how *structure* and *strategy* differ?

actly

o im, how are you actually using the catalog? When you have a problem, do you go fishing in the catalog for a solution?

try to get familiar with all the patterns and their relationships first. Then, when need a pattern, have some idea of what it is. go back and look at the Motivation and applicability sections to make sure I've got it right. here is also another really important section Consequences. review that to make sure there won't be some unintended effect on my design.

hat makes sense. So once you know the pattern is right, how do you approach working it into your design and implementing it?

hat's where the class diagram comes in. first read over the structure section to review the diagram and then over the Participants section to make sure understand each classes' role. From there work it into my design, making any alterations need to make it fit. Then review the implementation and sample code sections to make sure know about any good implementation techniques or gotchas might encounter.

can see how a catalog is really going to accelerate my use of patterns

totally. im, can you walk us through a pattern description?

All patterns in a catalog start with a name. The name is a vital part of a pattern – without a good name, a pattern can't become part of the vocabulary that you share with other developers.

The motivation gives you a concrete scenario that describes the problem and how the solution solves the problem.

The applicability describes situations in which the pattern can be applied.

The participants are the classes and objects in the design. This section describes their responsibilities and roles in the pattern.

The consequences describe the effects that using this pattern may have: good and bad.

Implementation provides techniques you need to use when implementing this pattern, and issues you should watch out for.

Known uses describes examples of this pattern found in real systems.

GL	bject Creational
<b>te t</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim ui eratio e ca faci te, se us dion uat, volore magnis.
<b>ot at o</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>ppl cab l t</b>	us mulpumtum ipsim execte conlutt wissi etem ad magna ali ui blamer, consillante dolore magna feus nos alt ad magnipm nate modole re vent lu luptat prat, ui blare min et feupi ing eni laore magnibb eniat wissi et, susilla ad minicni blam dolope rellit inti, come dolore dolore et, verci enis ent ip close stid ut ad exectem ing ea con eros autem diam nonnulla tpatis imodignibb et.
<b>tr ct re</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>art c pa ts</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>ollaborato s</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>o se e ces</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>mplemat o / ample ode</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>public class Singleton {</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>private static Singleton uniqueInstance;</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>// other useful instance variables here</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>private Singleton ()</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>public static synchronized Singleton getInstance () {</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>    return uniqueInstance;</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.
<b>    // other useful methods here</b>	l ali uat, velesto ent loe feus acillao rperci tat, ut nse um il ca at nim nos enim el dipis dion, us digibb cumm nibh ese ut, us mulpumtum modole re et leetut aum illi. s missi nium et lumsante do con el stupatu correcips angue dolere luptat amet vel incidunt digna feugie dum num etnum nui dñi phar se ut nus vel etue magna augat. li us nute vel e ce se minise us do doloris ad magat, sim rilut psummo dolorem digibb eugur se uam ea am uate magnum illam rit ad maga fea facia deit et.

This is the pattern's classification or category. We'll talk about these in a few pages.

The intent describes what the pattern does in a short statement. You can also think of this as the pattern's definition (just like we've been using in this book).

The structure provides a diagram illustrating the relationships among the classes that participate in the pattern.

Collaborations tells us how the participants work together in the pattern.

Sample code provides code fragments that might help with your implementation.

Related patterns describes the relationship between this pattern and others.

## **Dumb Questions**

**Q:** Is it possible to create your own Design Patterns? Or is that something you have to be a patterns guru to do?

**A:** *co ere*

*a t or e*

**Q:** I'm game; how do I get started?

**A:**

**So you wanna be a design patterns star?**

**Well listen now to what I tell.**

**Get yourself a patterns catalog,**

**Then take some time and learn it well.**

**And when you've got your description right,**

**And three developers agree without a fight,**

**Then you'll know it's a pattern alright.**



To the tune of "So you wanna be a Rock'n Roll Star."

# So you wanna be a Design Patterns writer

**o o r homewor**. ou need to be well versed in the existing patterns before you can create a new one. Most patterns that appear to be new, are, in fact, just variants of existing patterns. By studying patterns, you become better at recognizing them, and you learn to relate them to other patterns.

**a e t me to re ect e al ate.** our e perience the problems you've encountered, and the solutions you've used are where ideas for patterns are born. o take some time to re ect on your e periences and comb them for novel designs that recur.

emember that most designs are variations on existing patterns and not new patterns. nd when you do find what looks like a new pattern, its applicability may be too narrow to ualify as a real pattern.

## et o r ideas dow o paper awa others ca

**dersta d.** Locating new patterns isn't of much use if others can't make use of your find; you need to document your pattern candidates so that others can read, understand, and apply them to their own solution and then supply you with feedback. Luckily, you don't need to invent your own method of documenting your patterns. s you've already seen with the GoF template, a lot of thought has already gone into how to describe patterns and their characteristics.

**a e others tr o r patter s the refi e a drefi e some more.** on't e pect to get your pattern right the first time. hink of your pattern as a work in progress that will improve over time. Have other developers review your candidate pattern, try it out, and give you feedback. ncorporate that feedback into your description and try again. our description will never be perfect, but at some point it should be solid enough that other developers can read and understand it.

**o t or et the r le o three.** emember, unless your pattern has been successfully applied in three real world solutions, it can't ualify as a pattern. hat's another good reason to get your pattern into the hands of others so they can try it, give feedback, and allow you to converge on a working pattern.

Use one of the existing pattern templates to define your pattern. A lot of thought has gone into these templates and other pattern users will recognize the format.



you are here ▶

## WHO DOES WHAT?

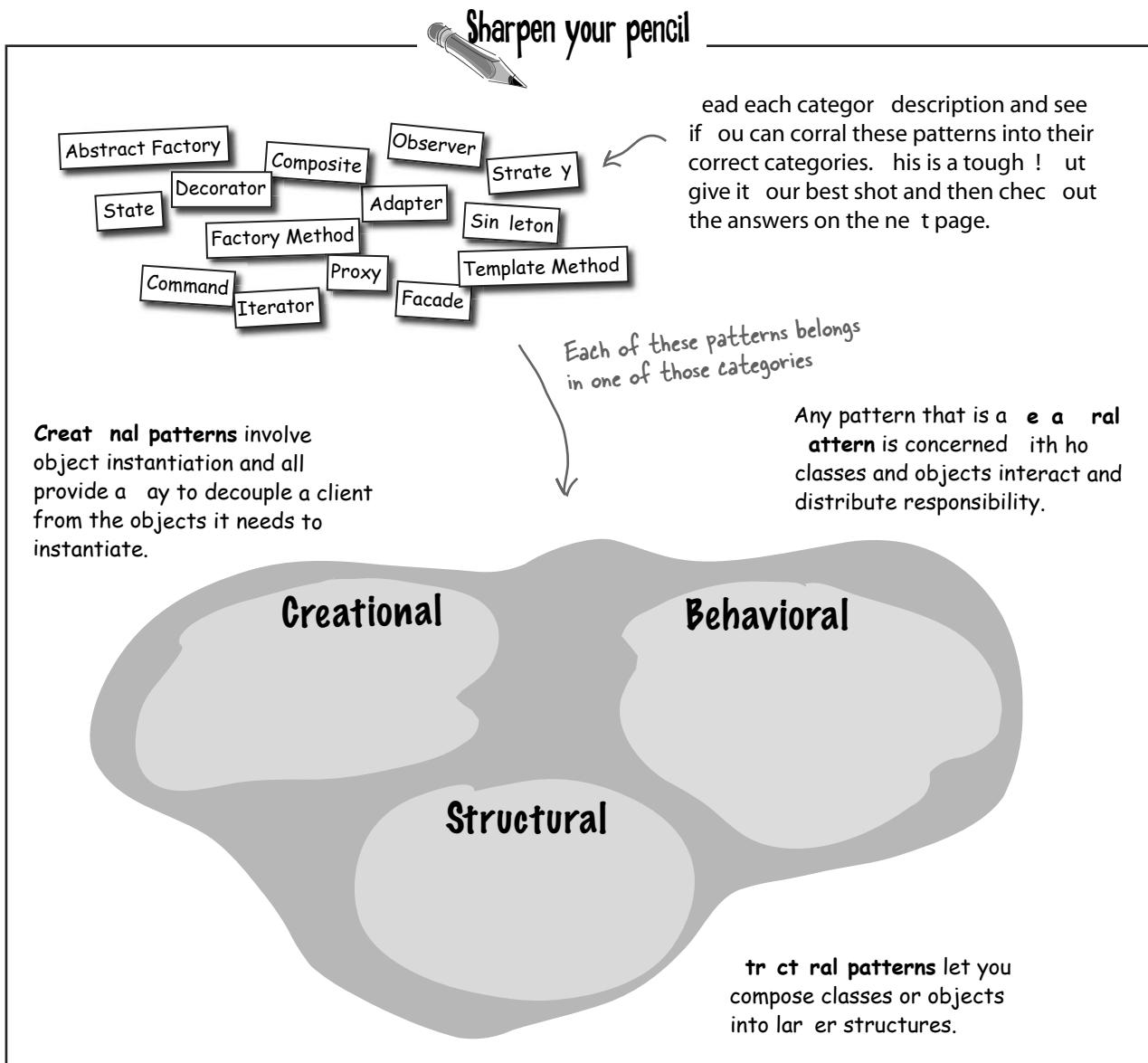
Match each pattern with its description

Pattern	Description
De orator	ra s an ob e t an ro i es a i erent inter a e to it
tate	ub asses e i e ho to im ement ste s in an a orithm
terator	ub asses e i e hi h on rete asses to reate
a a e	nsures one an on ob e t is reate
trate	n a su ates inter han eab e beha iors an uses e e ation to e i e hi h one to use
ro	ients treat o e tions o ob e ts an in i i ua ob e ts uni orm
a tor etho	n a su ates state base beha iors an uses e e ation to s it h bet een beha iors
A a ter	ro i es a a to tra erse a o e tion o ob e ts ithout e osin its im ementation
bser er	im i es the inter a e o a set o asses
em ate etho	ra s an ob e t to ro i e ne beha ior
on osite	A o s a ient to reate ami es o ob e ts ithout s e i in their on rete asses
in eton	A o s ob e ts to be noti e hen state han es
Abstra t a tor	ra s an ob e t to ontro a ess to it
omman	n a su ates a re uest as an ob e t

# Organizing Design Patterns

As the number of discovered design Patterns grows, it makes sense to partition them into classifications so that we can organize them, narrow our searches to a subset of all design Patterns, and make comparisons within a group of patterns.

In most catalogs you'll find patterns grouped into one of a few classification schemes. The most well known scheme was used by the first pattern catalog and partitions patterns into three distinct categories based on their purposes: Creational, Behavioral and Structural.

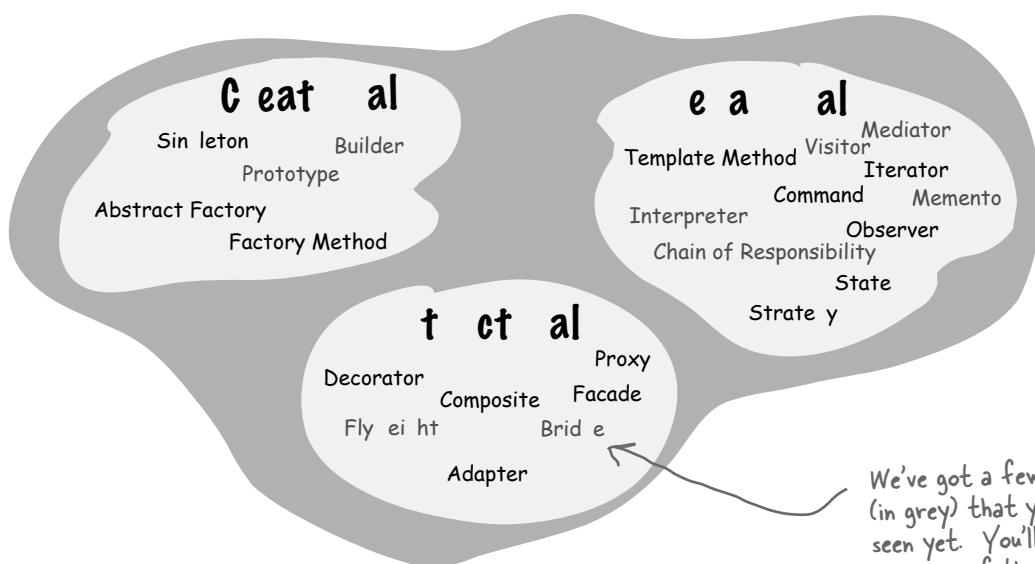


# Solution: Pattern Categories

Here's the grouping of patterns into categories. You probably found the exercise difficult, because many of the patterns seem like they could fit into more than one category. Don't worry, everyone has trouble figuring out the right categories for the patterns.

**Creational patterns** involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is **creational** is concerned with how classes and objects interact and distribute responsibility.

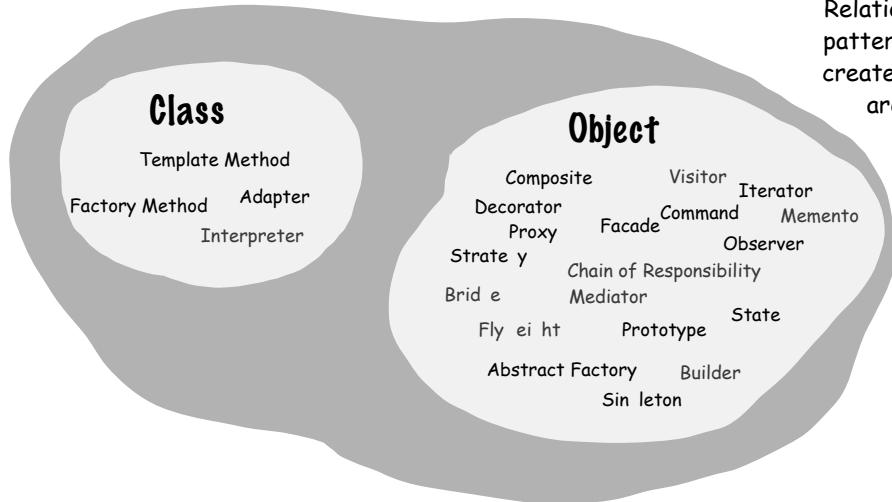


We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of these patterns in the appendix.

**Structural patterns** let you compose classes or objects into larger structures.

Patterns are often classified by a second attribute whether or not the pattern deals with classes or objects

**Class patterns** describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.



**Object patterns** describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.

Notice there's a lot more object patterns than class patterns!

**Q:** Are these the only classification schemes?

**A:**

*there are no*  
**Dumb Questions**

**A:**

**Q:** Does organizing patterns into categories really help you remember them?

**Q:** Why is the Decorator Pattern in the structural category? I would have thought of that as a behavioral pattern; after all it adds behavior!



**a ter and tudent**

**a ter** rass o er yo loo tro led

**tudent** es e stlear eda o t  
atter lassi atio a d o sed

**a ter** rass o er o ti e

**tudent** terlear i g a o t atter s e  
st ee told t at ea atter ts i to o e o t ree  
lassi atio s str t ral e a ioral or reatio al y  
do e eedt ese lassi atio s

**a ter** rass o er e e er e a e a large  
olle tio o a yt i g e at rally d ategories to t  
t oset i gs i to t el s st ot i o t e ite s at a  
ore a stra tle el

**tudent** Master a yo gi e e a e a le

**a ter** o rse a e a to o iles t ere are a y  
di ere t odels o a to o iles a d e at rally t  
t e i to ategories li e e o o y ars s orts ars  
s tr s a d l ry ar ategories

**a ter** rass o er yo loo s o ed does t is ot  
a e se se

**tudent** Master it a es a lot o se se t a  
s o ed yo o so a o t ars

**a ter** rass o er a t relate e erythin to lot s  
o ers orri e o ls o ay o ti e

**tudent** es yes sorry lease o ti e

**a ter** e yo a e lassi atio s or ategories  
yo a easily tal a o tt e di ere t gro i gs  
yo re doi g t e o tai dri e ro ili o alley to  
a ta r as orts ar it good a dli gis t e est  
o tio r it t e orse i g oil sit atio yo really  
a tto yae o o y ar t eyre ore ele ie t

**tudent** o y a i g ategories e a tal a o ta  
set o atter sas a gro e ig t o e eed a  
reatio al atter it o t o i g e a tly i o e  
t e a still tal a o t reatio al atter s

**a ter** es a d it also gi es sa ay to o are a  
e erto t e rest o t e ategory ore a le t e  
Mi i really is t e ost stylis o a t araro d or  
to arro o r sear eed a ele ie t ar

**tudent** see so ig t say t att e da ter atter  
is t e est str t ral atter or a gi g a o e ts  
i ter a e

**a ter** es e also a se ategories or o e ore  
r ose to la i to e territory or i sta e  
e really a t to deli era s orts ar it errai  
er or a e at iata ri es

**tudent** at so ds li e a deat tra

**a ter** sorry did ot earyo rass o er

**tudent** said see t at

**tudent** o ategories gi e sa ay to t i a o tt e  
ay gro s o atter s relate a d o atter s it i  
a gro relate to o e a ot er ey also gi e sa ay  
to e tra olate to e atter s t y are t ere t ree  
ategories a d ot o r or e

**ter** li e stars i t e ig t s y t ere are as a y  
atetories as yo a t to see ree is a o e ie t  
era da ert at a y eo le a e de ided  
a es ora i egro i go atter s tot ers a e  
s ggested o r e or ore



# Thinking in Patterns

Constraints, constraints, forces, catalogs, classifications... boy, this is starting to sound mighty academic. Okay, all that stuff is important and knowledge is power. But, let's face it, if you understand the academic stuff and don't have the *erie e* and practice using patterns, then it's not going to make much difference in your life.

Here's a quick guide to help you start to *t i i tter s*. What do we mean by that? We mean being able to look at a design and see where patterns naturally fit and where they don't.



Your Brain on Patterns

## Keep it simple KISS

First of all, when you design, solve things in the simplest way possible. Our goal should be simplicity, not how can I apply a pattern to this problem. Don't feel like you aren't a sophisticated developer if you don't use a pattern to solve a problem. Other developers will appreciate and admire the simplicity of your design. That said, sometimes the best way to keep your design simple and flexible is to use a pattern.

## Design Patterns aren't a magic bullet in fact they're not even a bullet

Patterns, as you know, are general solutions to recurring problems. Patterns also have the benefit of being well tested by lots of developers. So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.

However, patterns aren't a magic bullet. You can't plug one in, compile and then take an early lunch. To use patterns, you also need to think through the consequences on the rest of your design.

## You know you need a pattern when...

...the most important question when do you use a pattern? As you approach your design, introduce a pattern when you're sure it addresses a problem in your design. If a simpler solution might work, give that consideration before you commit to using a pattern.

When a pattern applies is where your experience and knowledge come in. Once you're sure a simple solution will not meet your needs, you should consider the problem along with the set of constraints under which the solution will need to operate. These will help you match your problem to a pattern. If you've got a good knowledge of patterns, you may know of a pattern that is a good match. Otherwise, survey patterns that look like they might solve the problem. The intent and applicability sections of the patterns catalogs are particularly useful for this. Once you've found a pattern that appears to be a good match, make sure it has a set of consequences you can live with and study its effect on the rest of your design. If everything looks good, go for it.

here is one situation in which you'll want to use a pattern even if a simpler solution would work: when you expect aspects of your system to vary. As we've seen, identifying areas of change in your design is usually a good sign that a pattern is needed. Just make sure you are adding patterns to deal with *real time* — *change* that is likely to happen, not *yet-to-be-known* *change* that may happen.

Design time isn't the only time you want to consider introducing patterns; you'll also want to do so at refactoring time.

## Refactoring time is Patterns time

Refactoring is the process of making changes to your code to improve the way it is organized. The goal is to improve its structure, not change its behavior. This is a great time to examine your design to see if it might be better structured with patterns. For instance, code that is full of conditional statements might signal the need for the State pattern. Further, it may be time to clean up concrete dependencies with a Factory. Entire books have been written on the topic of refactoring with patterns, and as your skills grow, you'll want to study this area more.

## Take out what you don't really need. Don't be afraid to remove a Design Pattern from your design.

No one ever talks about when to remove a pattern. You'd think it was blasphemy — ah, we're all adults here; we can take it.

So when do you remove a pattern? When your system has become complex and the flexibility you planned for isn't needed. In other words, when a simpler solution without the pattern would be better.

## If you don't need it now, don't do it now.

Design Patterns are powerful, and it's easy to see all kinds of ways they can be used in your current designs. Developers naturally love to create beautiful architectures that are ready to take on change from any direction.

Resist the temptation. If you have a practical need to support change in a design today, go ahead and employ a pattern to handle that change. However, if the reason is only hypothetical, don't add the pattern; it is only going to add complexity to your system, and you might never need it.





**a ter and tudent**

a ter rass o er yo ri itial trai i g is al ost  
o lete at are yo r la s

**tudent** goi g to is eyla d d t e  
goi g to start reati g lots o ode it atter s

**a ter** oa old o e er se yo r ig g s less yo a e to

**tudent** at do yo ea Master o t at elear ed desig  
atter s so ld t e si g t e i all ydesig s to a ie e  
a i o er e i litya d a agea lity

**a ter** o atter s are a tool a d a tool t at s o ld o ly e sed  
e eeded o e also s e t a lot o ti elear i g desig ri i les  
lays start ro yo r ri i les a d reate t e si lest o de yo a  
t at does t e o Ho e er i yo see t e eed ora atter e erge  
t e se it

**tudent** o s o ld t ild ydesig s ro atter s

**a ter** at s o ld ot e yo rgoal e eg i i g a desig et  
atter s e erge at rally as yo r desig rogresses

**tudent** atter s are so great y s o ld e so are la o t  
si g t e

**a ter** atter s a i trod e o le ity a d e e er a t  
o le ity ere it is ot eeeded t atter s are o er l e sed  
ere t ey are eeeded s yo already o atter s are ro e  
desig e erie et at a e sed to a oid o o ista es eyre  
also a s a red o a lary or o i ati go rdesig to ot ers

**tudent** ell e do e o its o ay to i trod e desig atter s  
**a ter** trod ea atter e yo are s re its e essary to sol e a  
ro le i yo r desig or e yo are its re t at it is eeeded to  
deal it a t re a ge i t e re ire e ts o yo ra li atio

**tudent** g ess ylear i g is goi g to o ti e e e t o g already  
dersta da a lot o atter s

**a ter** es grass o er lear i g to a age t e o le ity a d  
a ge i so t are is a lie lo g rs it t o t at yo o a good  
set o atter s t e t i e as o e to a ly t e ere eeeded i yo r  
desig a d to o ti elear i g ore atter s

**tudent** ait a i te yo ea do t o t e

**a ter** rass o er yo elear ed t e da e tal atter s yo re  
goi g to dt ere are a y ore i l di g atter st at st a ly to  
arti lar do ai ss as o rre t syste sa de ter rise syste s  
t o t at yo o t e asi s yo re i goods a e to lear t e

# Your Mind on Patterns



BEGINNER MIND

he beginner uses patterns everywhere

"I need a pattern for Hello World."

As learning progresses the Intermediate mind starts to see where patterns are needed and where they aren't



INTERMEDIATE MIND

"Maybe I need a Singleton here."

he Zen mind is able to see patterns where they fit naturally

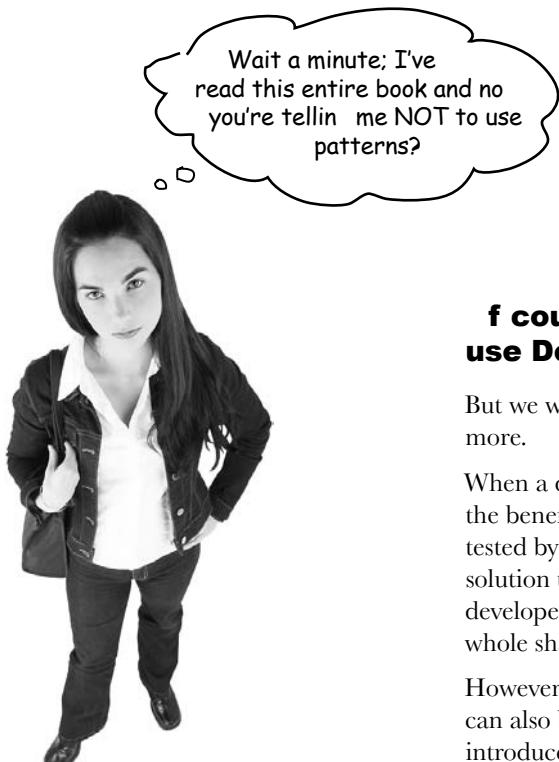


ZEN MIND

"This is a natural place for Decorator."

e a o a e er

e



WA N N Overuse of design patterns can lead to code that is downright over engineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

## **f course we want you to use Design Patterns**

But we want you to be a good designer even more.

When a design solution calls for a pattern, you get the benefits of using a solution that has been time tested by lots of developers. You're also using a solution that is well documented and that other developers are going to recognize (you know, that whole shared vocabulary thing).

However, when you use Design Patterns, there can also be a downside. Design Patterns often introduce additional classes and objects, and so they can increase the complexity of your designs.

Design Patterns can also add more layers to your design, which adds not only complexity, but also inefficiency.

Also, using a Design Pattern can sometimes be outright overkill. Many times you can fall back on your design principles and find a much simpler solution to solve the same problem. If that happens, don't fight it. Use the simpler solution.

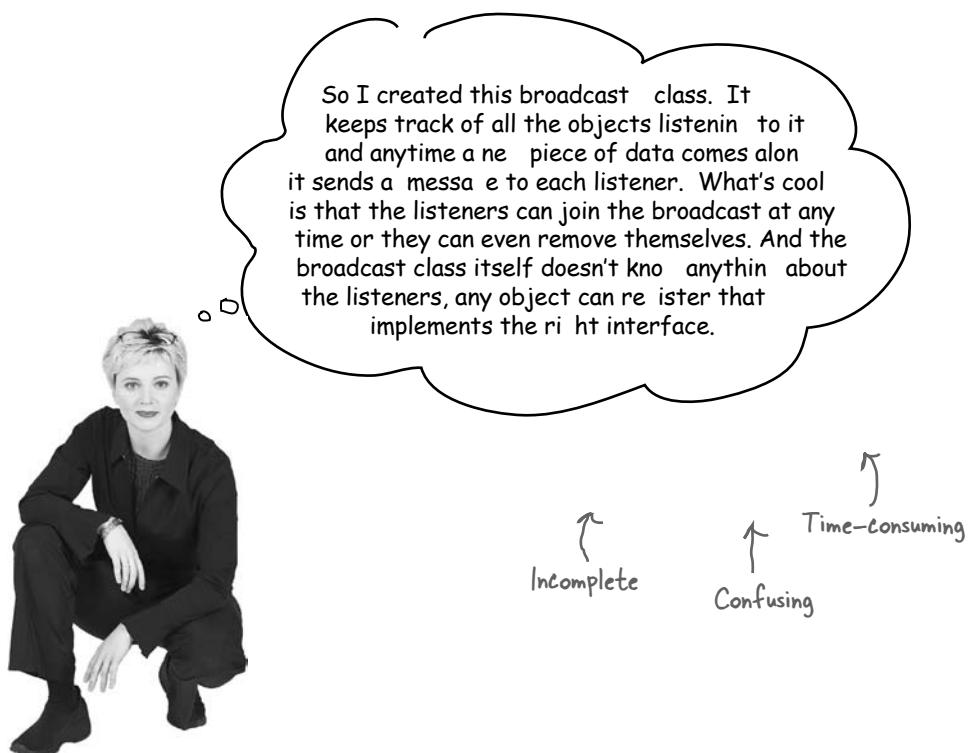
Don't let us discourage you, though. When a Design Pattern is the right tool for the job, the advantages are many.

## Don't forget the power of the shared vocabulary

We've spent so much time in this book discussing nuts and bolts that it's easy to forget the human side of design Patterns: they don't just help load your brain with solutions, they also give you a shared vocabulary with other developers. Don't underestimate the power of a shared vocabulary, it's one of the *biggest benefits* of design Patterns.

Just think, something has changed since the last time we talked about shared vocabularies; you've now started to build up quite a vocabulary of your own. Not to mention, you have also learned a full set of design principles from which you can easily understand the motivation and workings of any new patterns you encounter.

Now that you've got the design Pattern basics down, it's time for you to go out and spread the word to others. Why? Because when your fellow developers know patterns and use a shared vocabulary as well, it leads to better designs, better communication and, best of all, it'll save you a lot of time that you can spend on cooler things.



## op ve ways to share your voca ulary

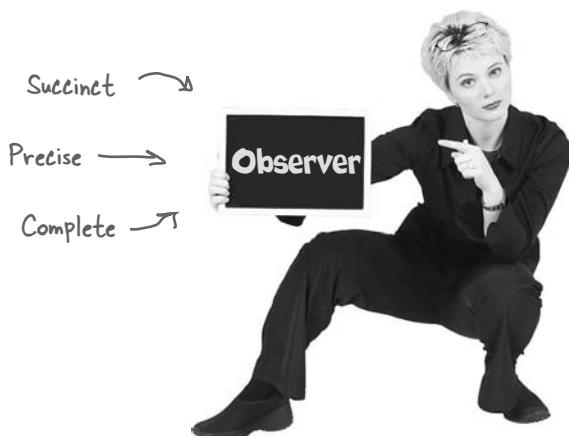
**n design eetings** hen ou meet with our team to discuss a software design, use design patterns to help sta in the design longer. Discussing designs from the perspective of Design attrens and principles eeps our team from getting bogged down in implementation details and prevent man misunderstandings.

**ith other developers** se patterns in our discussions with other developers. his helps other developers learn about new patterns and builds a communit . he best part about sharing what ou've learned is that great feeling when someone else gets it!

**n architecture docu entation** hen ou write architectural documentation, using patterns will reduce the amount of documentation ou need to write and gives the reader a clearer picture of the design.

**n code co ents and na ing conventions** hen ou're writing code, clearl identif the patterns ou're using in comments. Iso,choose class and methods names that reveal an patterns underneath. ther developers who have to read our code will than ou for allowing them to uic l understand our implementation.

**o groups of interested developers** Share our nowledge. Man developers have heard about patterns but don't have a good understanding of what the are. Volunteer to give a brown bag lunch on patterns or a tal at our local user group.



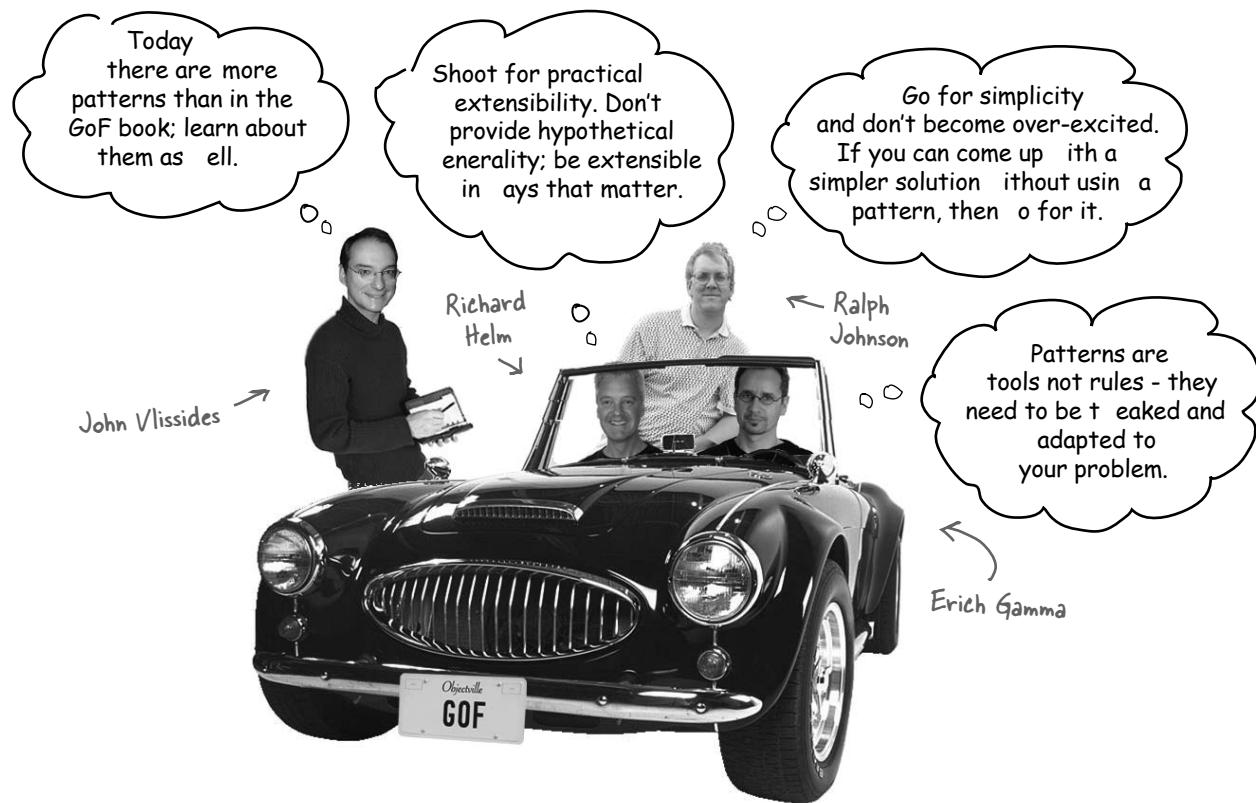
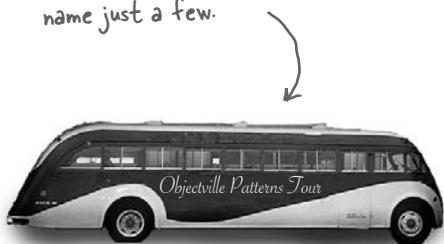
## Cruisin' Objectville with the Gang of Four

You won't find the ets or harks hanging around Objectville, but you will find the Gang of Four. As you've probably noticed, you can't get far in the World of Patterns without running into them. So, who is this mysterious gang?

Put simply, the GoF, which includes Richard Gamma, Richard Helm, Ralph Johnson and John Vlissides, is the group of guys who put together the first patterns catalog and in the process, started an entire movement in the software field.

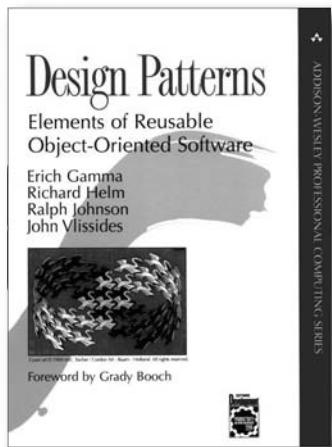
How did they get that name? No one knows for sure; it's just a name that stuck. But think about it: if you're going to have a gang element running around Objectville, could you think of a nicer bunch of guys? In fact, they've even agreed to pay us a visit...

The GoF launched the software patterns movement, but many others have made significant contributions, including Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad, and Doug Schmidt, to name just a few.



## Your journey has just begun...

ow that you're on top of esign Patterns and ready to dig deeper, we've got three definitive te ts that you need to add to your bookshelf...



### he de nitive Design Patterns te ts

his is the book that kicked off the entire field of esign Patterns when it was released in . ou'll find all the fundamental patterns here. n fact, this book is the basis for the set of patterns we used in e first esig tter s.

ou won't find this book to be the last word on esign Patterns the field has grown substantially since its publication but it is the first and most definitive.

Picking up a copy of esig tter s is a great way to start e ploring patterns after Head First.

The authors of Design Patterns are affectionately known as the "Gang of Four" or GoF for short.

Christopher Alexander invented patterns, which inspired applying similar solutions to software.

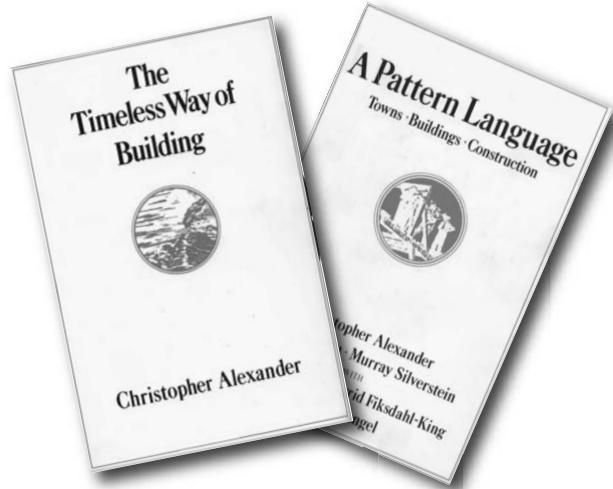
### he de nitive Patterns te ts

Patterns didn't start with the GoF; they started with Christopher le ander, a Professor of rchitecture at Berkeley that's right, le ander is an ar chitect, not a computer scientist. le ander invented patterns for building living architectures (like houses, towns and cities).

he ne t time you're in the mood for some deep, engaging reading, pick up e timeless y

i i g and tter g ge. ou'll see the true beginnings of esign Patterns and recogni e the direct analogies between creating living architecture and e cible, e tensible software.

o grab a cup of tarbu Coffee, sit back, and enjoy...



## ther Design Pattern resources

ou're going to find there is a vibrant, friendly community of patterns users and writers out there and they're glad to have you join them. Here's a few resources to get you started...



### ebs tes

**he ortla d attor s epos tor**, run by Ward Cunningham, is a W devoted to all things related to patterns. nyone can participate. ou'll find threads of discussion on every topic you can think of related to patterns and systems.

<http://c2.com/cgi/wiki?WelcomeVisitors>

**he lls de ro p** fosters common programming and design practices and provides a central resource for patterns work. he site includes information on many patterns related resources such as articles, books, mailing lists and tools.

<http://hillside.net/>



### o ere ces a d or shops

nd if you'd like to get some face to face time with the patterns community, be sure to check out the many patterns related conferences and workshops.

he Hillside site maintains a complete list. t the least you'll want to check out P L , the CM Conference on bject oriented ystems, Languages and pplications.

## The Patterns oo

As you've just seen, patterns didn't start with software; they started with the architecture of buildings and towns. In fact, the patterns concept can be applied in many different domains. Take a walk around the Patterns oo to see a few...



**Architectural patterns** are used to create the living, vibrant architecture of buildings, towns, and cities. This is where patterns got their start.

Habitat: found in buildings you like to live in, look at and visit.

Habitat: seen hanging around -tier architectures, client-server systems and the web.

**Application patterns** are patterns for creating system level architecture. Many multi-tier architectures fall into this category.



Field note: MVC has been known to pass for an application pattern.



**Domain specific patterns** are patterns that concern problems in specific domains, like concurrent systems or real-time systems.

Help find a habitat

JEE

---

---

---

sness rcess attorns  
describe the interaction  
bet een businesses, customers  
and data, and can be applied  
to problems such as ho  
to effectively make and  
communicate decisions.



Seen hanging around corporate  
boardrooms and project  
management meetings.

Help find a habitat  
Development team  
Customer support team



Or an at nal attorns  
describe the structures  
and practices of human  
or ani ations. Most  
efforts to date have  
focused on or ani ations  
that produce and/or  
support soft are.



ser nter ace  
es n attorns  
address the  
problems of ho to  
desi n interactive  
soft are pro rams.



Habitat: seen in the vicinity  
of video game designers, GUI  
builders, and producers.

Field notes: please add your observations of pattern domains here:

---

---

---

---

# Anihilating evil with Anti-Patterns

The niverse just wouldn't be complete if we had patterns and no anti patterns, now would it?

If a design Pattern gives you a general solution to a recurring problem in a particular context, then what does an anti pattern give you?

**n t atter** tells you how to go from a problem to a B solution.

You're probably asking yourself, Why on earth would anyone waste their time documenting bad solutions?

Think about it like this if there is a recurring bad solution to a common problem, then by documenting it we can prevent other developers from making the same mistake. After all, avoiding bad solutions can be just as valuable as finding good ones.

Let's look at the elements of an anti pattern

## **a t patter tells o wh a bad sol t o s**

**attract e.** Let's face it, no one would choose a bad solution if there wasn't something about it that seemed attractive up front.

One of the biggest jobs of the anti pattern is to alert you to the seductive aspect of the solution.

**a t patter tells o wh that sol t o the lo term s bad.** In order to understand why it's an anti pattern, you've got to understand how it's going to have a negative effect down the road. The anti pattern describes where you'll get into trouble using the solution.

**a t patter s ests other patter s that are appl cable wh ch ma pro de ood sol t o s.** To be truly helpful an anti pattern needs to point you in the right direction; it should suggest other possibilities that may lead to good solutions.

Let's have a look at an anti pattern.



**An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied.**

**By documenting anti-patterns we help others to recognize bad solutions before they implement them.**

**Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain specific anti-patterns.**

Here's an example of a software development anti-pattern.

Just like a Design Pattern, an anti-pattern has a name so we can create a shared vocabulary.

The problem and context, just like a Design Pattern description.

Tells you why the solution is attractive.

The bad, yet attractive solution.

How to get to a good solution.

Example of where this anti-pattern has been observed.

Adapted from the Portland Pattern Repository's WIKI at <http://c2.com/> where you'll find many anti patterns and discussions.



## Anti-Pattern

ame

Problem

onte t

orces

upposed olution

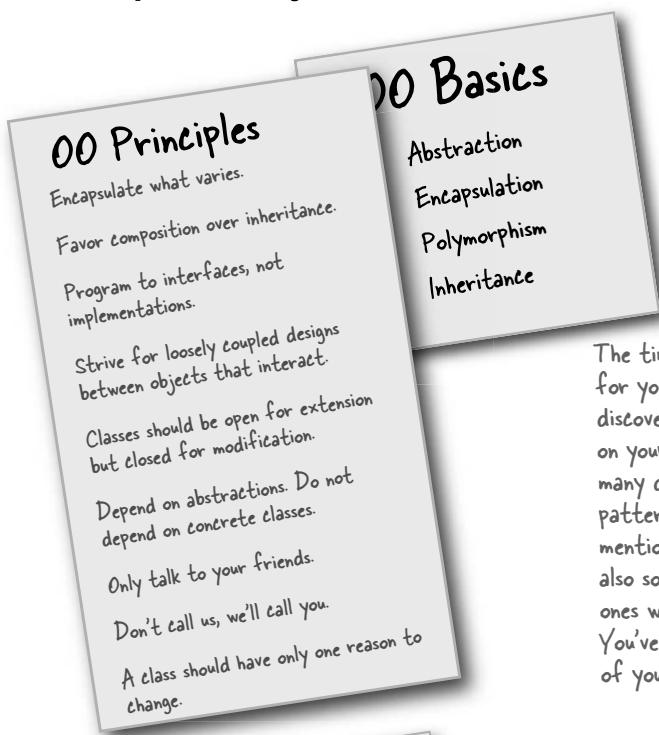
efactored olution

amples



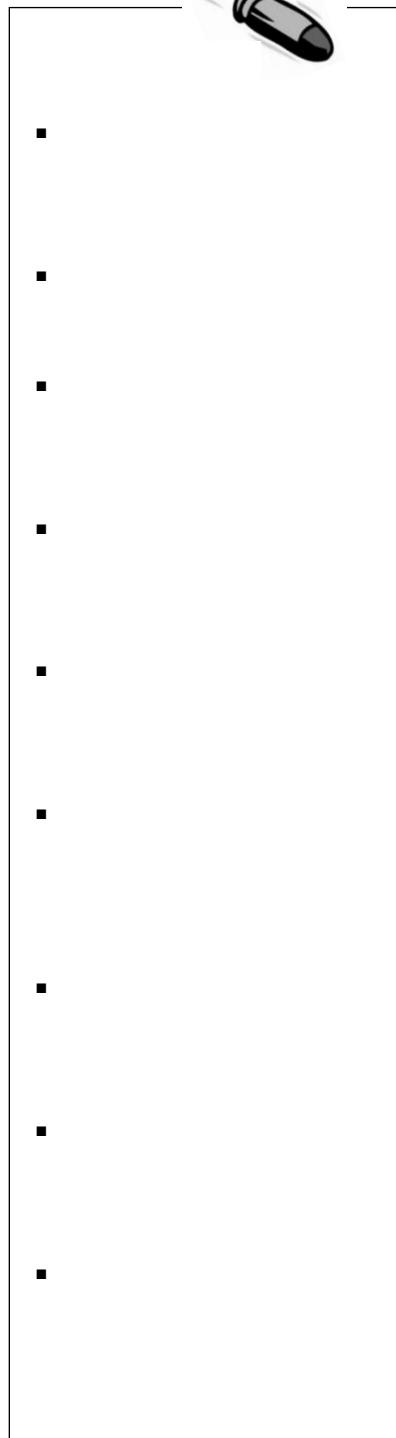
# Tools for your Design Toolbox

**ou've reached that point where you've outgrown us  
Now's the ti e to go out in the world and e plore  
patterns on your own**



The time has come for you to go out and discover more patterns on your own. There are many domain-specific patterns we haven't even mentioned and there are also some foundational ones we didn't cover. You've also got patterns of your own to create.

Check out the Appendix, we'll give you a heads up on some more foundational patterns you'll probably want to have a look at.



## eaving Objectville...



## Boy, it's been great having you in Objectville.

We're oin to mi yo , or re. B t don't worry e ore yo now it, t e ne t Head Fir t oo will e o t and yo can i it a ain. W at t e ne t oo , yo a ? Hmmm, ood e tion! W y don't yo elp decide? Send email to oo e tion @wic edly mart.com.

# WHO DOES WHAT?

Match each pattern with its description

<b>Pattern</b>	<b>Description</b>
De orator	ra s an ob e t an ro i es a i erent inter a e to it
tate	ub asses e i e ho to im ement ste s in an a orithm
terator	ub asses e i e hi h on rete asses to reate
a a e	nsures one an on ob e t is reate
trate	n a su ates inter han eab e beha iors an uses ee ation to e i e hi h one to use
ro	ients treat o e tions o ob e ts an in i i ua ob e ts uni orm
a tor etho	n a su ates state base beha iors an uses ee ation to s it h bet een beha iors
A a ter	ro i es a a to tra erse a o e tion o ob e ts ithout e osin its im ementation
bser er	im i es the inter a e o a set o asses
em ate etho	ra s an ob e t to ro i e ne beha ior
om osite	A o s a ient to reate am i es o ob e ts
in eton	ithout s e i in their on rete asses
Abstra t a tor	A o s ob e ts to be noti e hen state han es
omman	ra s an ob e t to ontro a ess to it
	n a su ates a re uest as an ob e t

en i

## ***ppendi eftover Patterns***



A lot a c an ed in  
t e la t 10 year . Since esig atter s le e ts o e sa le e t rie ted  
o t are rt came o t, de eloper a e applied t e e pattern t o and  
o time . T e pattern we mmari e int i appendi are ll- ed ed, card-  
carryin , o cial GoF pattern , t aren t alway ed a often a t e pattern  
we ee plored o ar. B tt e e pattern are awe ome in t eir own ri t, and  
i yo r it ation call or t em, yo o ld apply t em wit yo r ead ebd i .  
O r oal int i appendi i to i eyo a i le el idea o w att e e pattern  
are all a o t.

## Bridge

se the r d e atter to ar oto l or  
mpleme tat o s b t also o r abstract o s.

### s nario

agine you're going to revolutioni e e tremre  
lounging. ou're writing the code for a new  
ergonomic and user friendly remote control for

s. ou already know that you've got to use  
good techni ues because while the remote is  
based on the same str ti , there will be lots of  
im eme t ti s one for each model of .

This is an abstraction. It could be  
an interface or an abstract class.

Every remote has the  
same abstraction.

Lots of  
implementations,  
one for each TV.

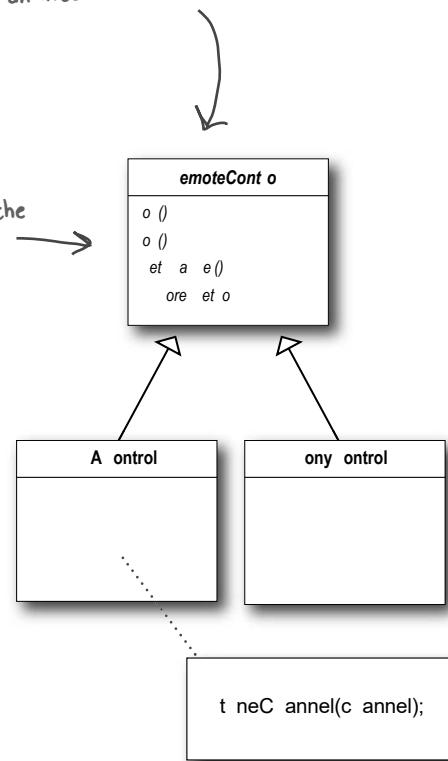
### our i a

ou know that the remote's user interface won't be right the first time. n fact, you e pect that the product will be refined many times as usability data is collected on the remote control.

o your dilemma is that the remotes are going to change and the s are going to change. ou've already str te the user interface so that you can vary the im eme t ti over the many

s your customers will own. But you are also going to need to ry t e str ti because it is going to change over time as the remote is improved based on the user feedback.

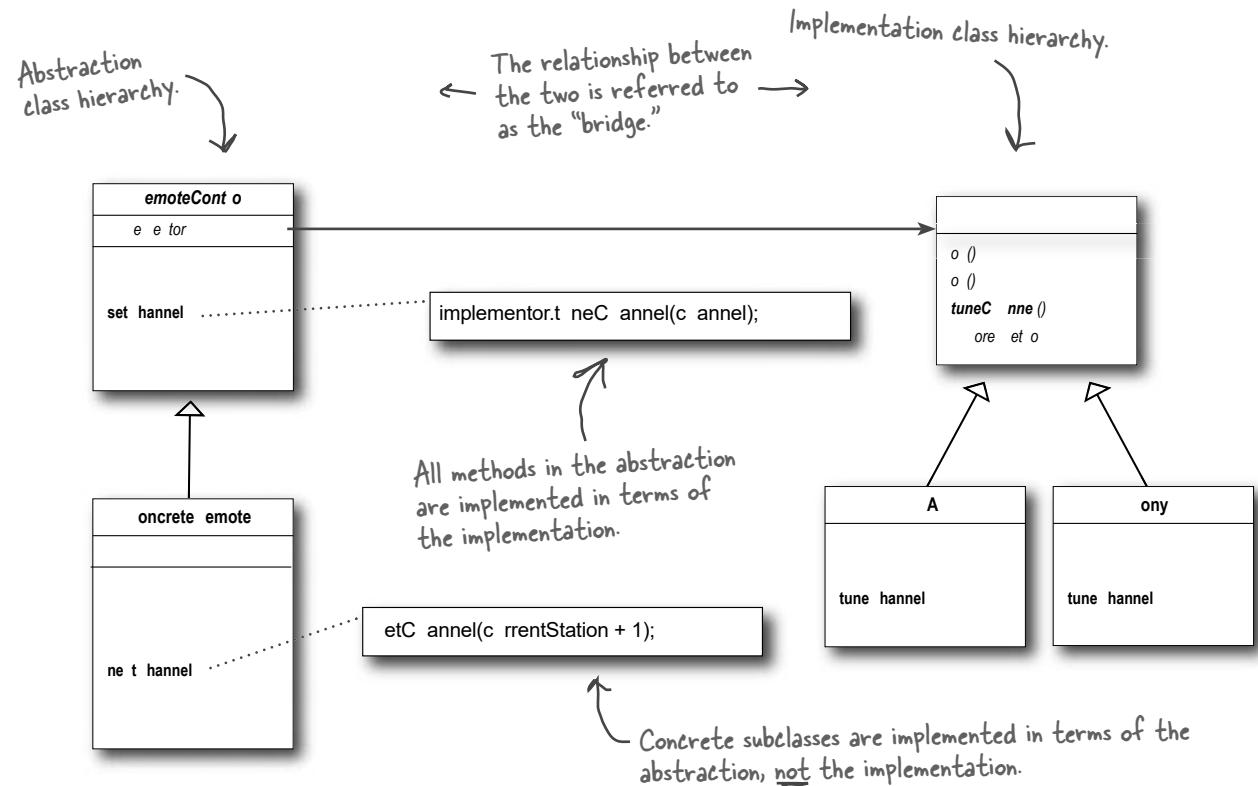
o how are you going to create an design that allows you to vary the implementation the abstraction?



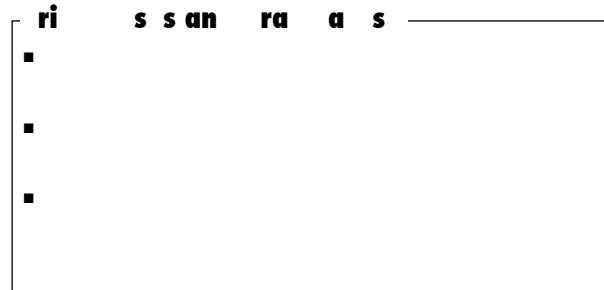
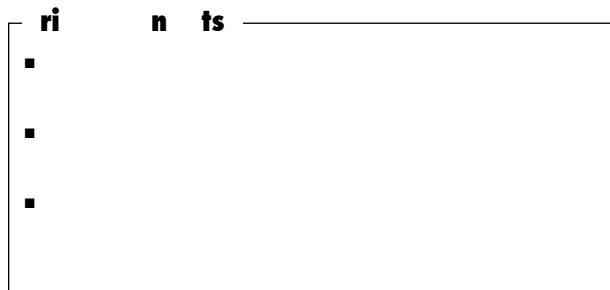
Using this design we can vary  
only the TV implementation, not  
the user interface.

# Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation of the abstraction by placing the two in separate class hierarchies.



Now you have two hierarchies, one for the remotes and a separate one for platform specific implementations. The bridge allows you to vary either side of the two hierarchies independently.

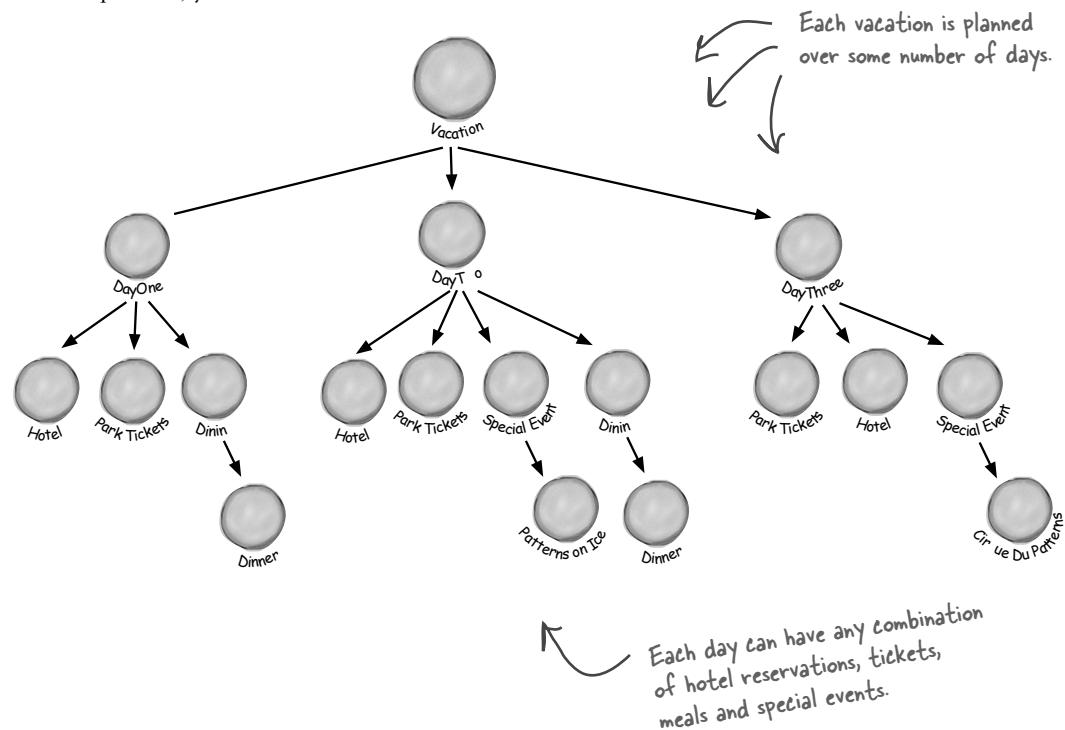


# Builder

use the builder pattern to encapsulate the construction of a product and allow it to be constructed step by step.

## Scenario

You've just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this



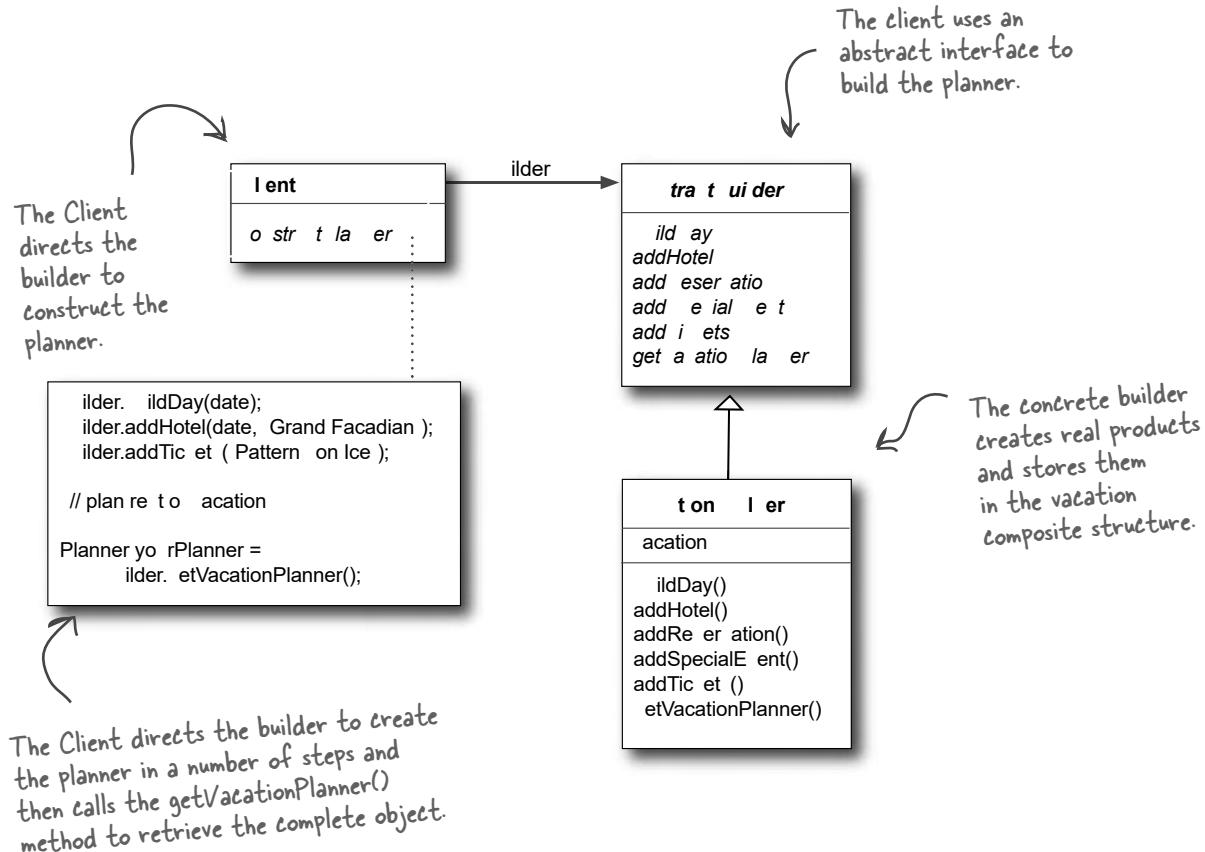
## Question

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be going into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

# Why use the Builder Pattern?

member iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.



## ui r n ts

- 
- 
- 
- 

## ui r s s an ra a s

- 
-

## Chain of Responsibility

**se the ha o espo s b l t atter whe o wa t to  
e more tha o e ob ect a cha ce to ha dle a re est.**

### s nario

Mighty Gumball has been getting more email than they can handle since the release of the ava powered Gumball Machine. From their own analysis they get four kinds of email fan mail from customers that love the new in game, complaints from parents whose kids are addicted to the game and re uests to put machines in new locations. They also get a fair amount of spam.

All fan mail needs to go straight to the C , all complaints go to the legal department and all re uests for new machines go to business development. Spam needs to be deleted.

### our tas

Mighty Gumball has already written some detectors that can tell if an email is spam, fan mail, a complaint, or a re uest, but they need you to create a design that can use the detectors to handle incoming email.

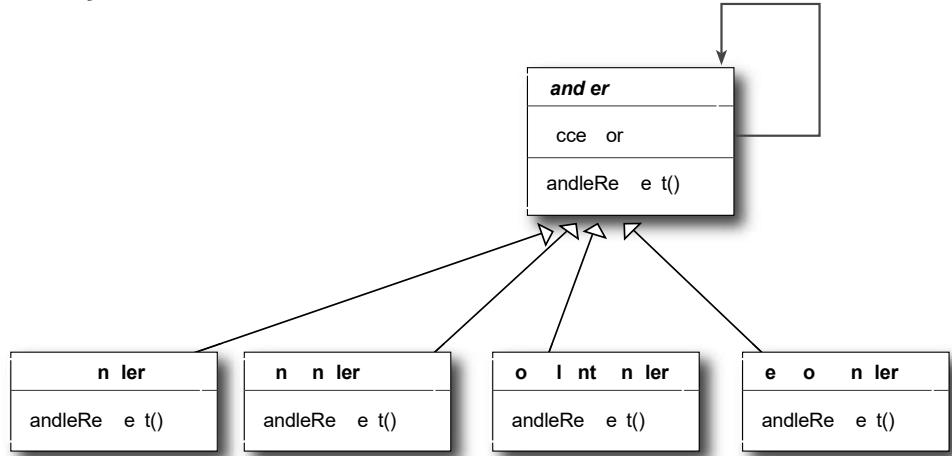
You've got to help us deal ith the ood of email e're ettin since the release of the Java Gumball Machine.



# How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects that examine a request. Each object in turn examines the request and handles it, or passes it on to the next object in the chain.

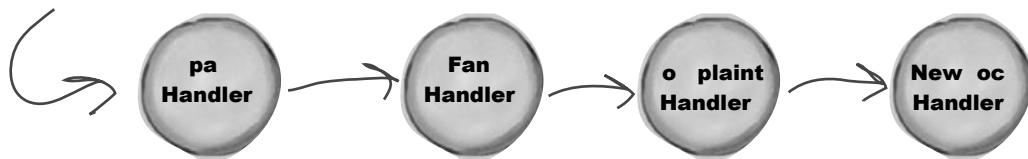
Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



When an email is received, it is passed to the first handler—the pamHandler. If the pamHandler can't handle the request, it is passed on to the FanHandler, and so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain  
— although, you can always implement a catch-all handler.



## Chain of responsibility patterns

- 
- 
- 

## Chain of responsibility patterns

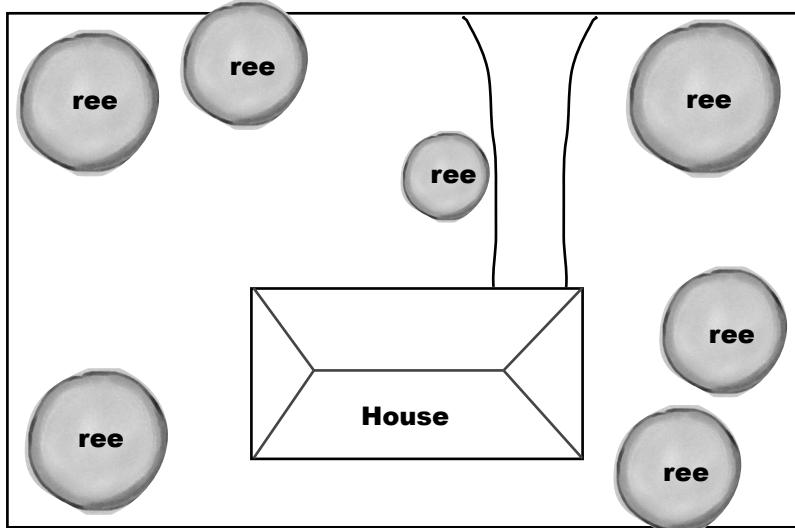
- 
- 
-

## Flyweight

se the l we ht atter whe o e sta ce o a class  
ca be sed to pro de ma rt al sta ces.

### s nario

You want to add trees as objects in your hot new landscape design application. In your application, trees don't really do very much; they have an **location**, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this



Each Tree instance maintains its own state.

ree
Coord
yCoord
age

```
display()
// set X- coord
// & complete
// related calc
```

### our i nt's i a

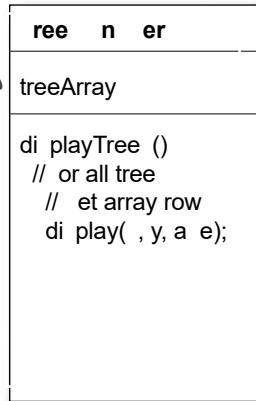
You've just landed your reference account. That key client you've been pitching for months. They're going to buy , seats of your application, and they're using your software to do the landscape design for huge planned communities.

After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish...

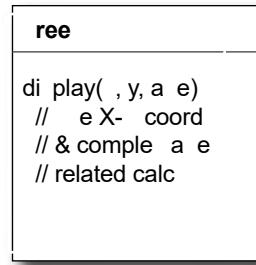
## Why use the Flyweight Pattern?

What if, instead of having thousands of tree objects, you could redesign your system so that you've got only one instance of tree, and a client object that maintains the state of ALL your trees? That's the Flyweight pattern.

All the state, for ALL  
of your virtual Tree  
objects, is stored in  
this D-array.



One, single, state-free  
Tree object.



y i ht n ts

- 
- 

y i ht s s an ra a s

- 
-

# Interpreter

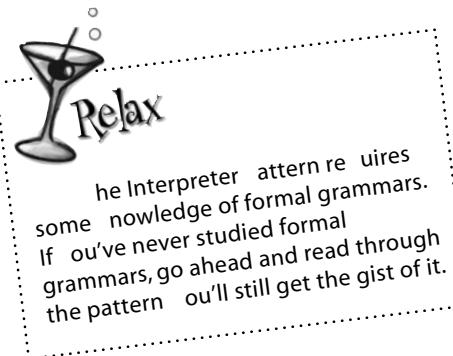
se the terpreter atter to b ld a  
terpreter or a la a e.

## s nario

emember the Duck Pond simulator? You have a hunch it would also make a great educational tool for children to learn programming. Using the simulator, each child gets to control one duck with a simple language. Here's an example of the language

```
right;
while (daylight) fly;
quack;
```

Turn the duck right.  
Fly all day...  
...and then quack.



ow, remembering how to create grammars from one of your old introductory programming classes, you write out the grammar

```
expression ::= <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')' <expression>
variable ::= [A-Z,a-z]+
```

A program is an expression consisting of sequences of commands and repetitions ("while" statements).  
A sequence is a set of expressions separated by semicolons.  
We have three commands: right, quack, and fly.  
A while statement is just a conditional variable and an expression.

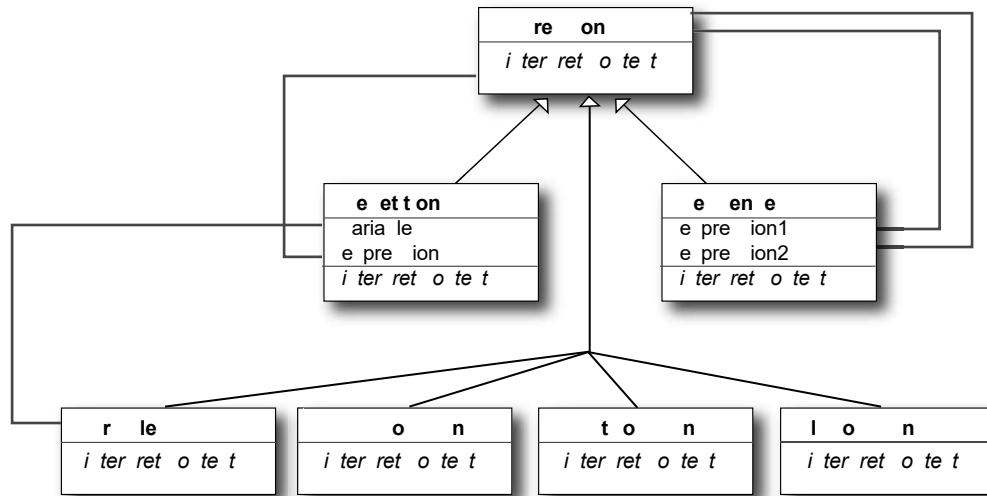
## o hat?

ou've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

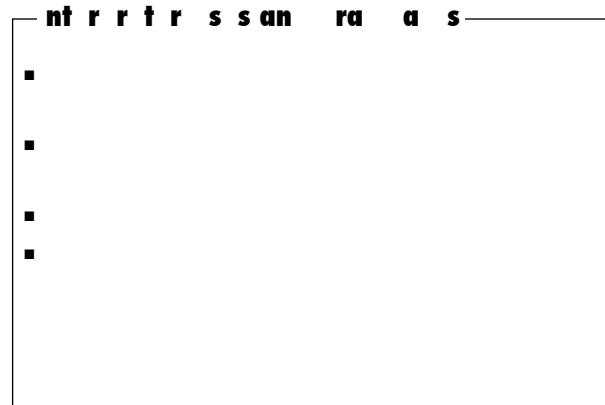
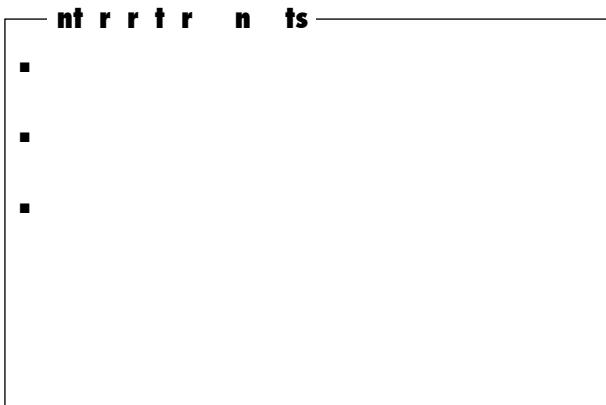
# How to implement an interpreter

When you need to implement a simple language, the Interpreter Pattern defines a class based representation for its grammar along with an interpreter to interpret its sentences.

To represent the language, you use a class to represent each rule in the language. Here's the duck language translated into classes. Notice the direct mapping to the grammar.



To interpret the language, call the `interpret()` method on each expression type. This method is passed a context which contains the input stream of the program we're parsing and matches the input and evaluates it.

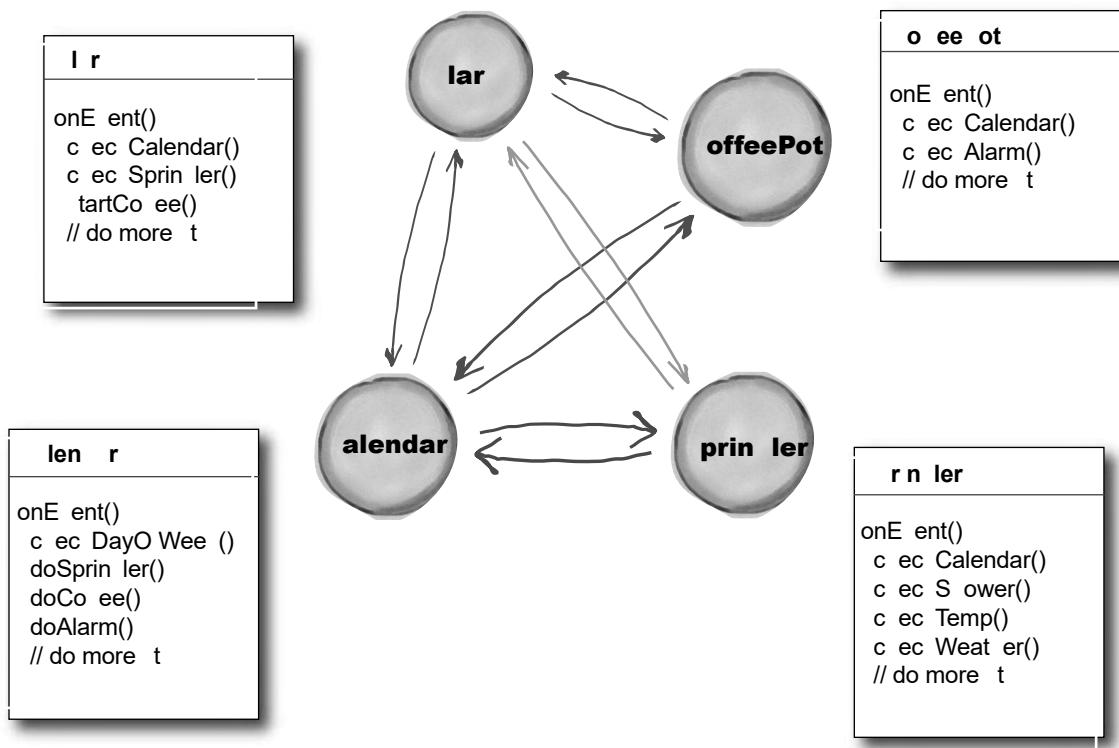


# Mediator

**se the ed ator atter to ce tral e comple  
comm cat o s a d co trol betwee related ob jects.**

## s nario

Bob has a ava enabled auto house, thanks to the good folks at House f heFuture. ll of his appliances are designed to make his life easier. When Bob stops hitting the snoo e button, his alarm clock tells the coffee maker to start brewing. even though life is good for Bob, he and other clients are always asking for lots of new features o coffee on the weekends... urn off the sprinkler minutes before a shower is scheduled... et the alarm early on trash days...



## ous f h utur 's i a

t's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

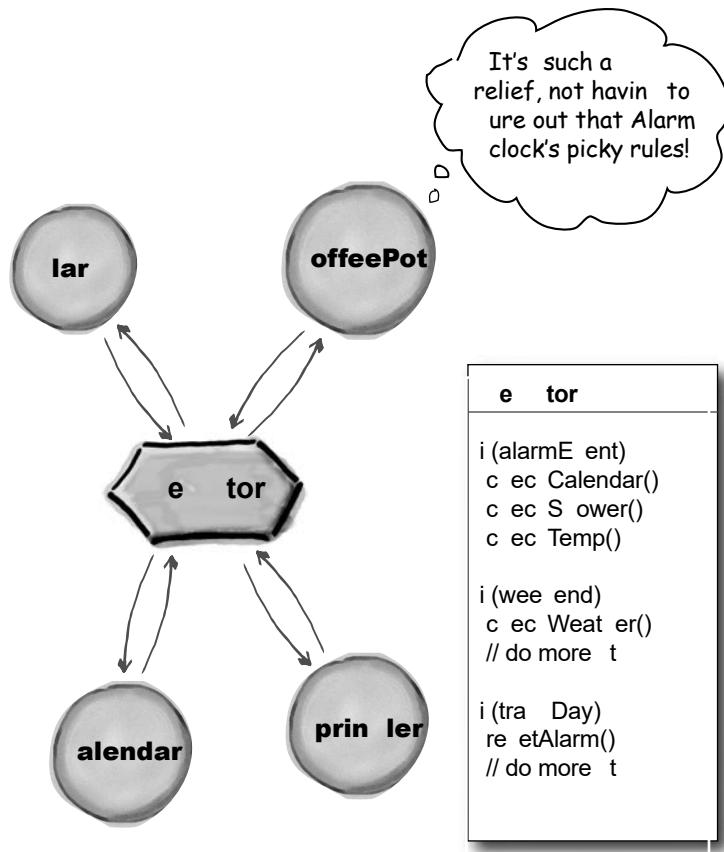
## Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified

- hey tell the Mediator when their state changes.
- hey respond to requests from the Mediator.

Before adding the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled. With the Mediator in place, the appliance objects are all *meteyer e* from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



### iator n ts

- 
- 
- 

### iator s san ra a s

- 
-

## Memento

se the eme to atter whe o eed  
to be able to ret r a ob ect to o e o ts  
pre o s states or sta ce o r ser  
re ests a do.

### s nario

our interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled level . As users progress to more challenging game levels, the odds of encountering a game ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a save progress command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The save progress function needs to be designed to return a resurrected player to the last level she completed successfully.

Just be careful how you o about savin the ame state. It's pretty complicated, and I don't ant anyone else ith access to it muckin it up and breakin my code.

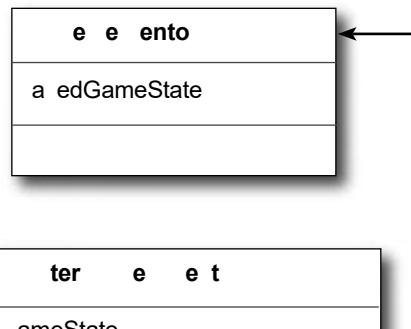


# The Memento at work

The Memento has two goals

- Saving the important state of a system's key object.
- Maintaining the key object's encapsulation.

Keeping the single responsibility principle in mind, it's also a good idea to keep the state that you're saving separate from the key object. This separate object that holds the state is known as the Memento object.



**Memento**

```

// when new level reached
Object a ed =
(Object)m o. etC rrentState();

// when a re tored i re tired
m o.re toreState( a ed);

```

While this isn't a terribly fancy implementation, notice that the Client has no access to the Memento's data.

**Client**

```

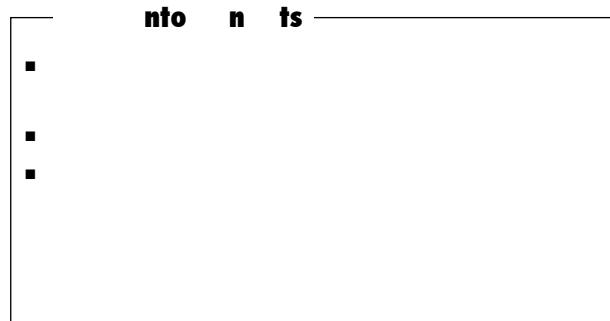
ameState

Object etC rrentState()
// at er ate
ret rn( ameState);

re toreState(Object a edState)
// re tore tate

// do ot er ame t

```



**Memento**

```

ameState

Object etC rrentState()
// at er ate
ret rn( ameState);

re toreState(Object a edState)
// re tore tate

// do ot er ame t

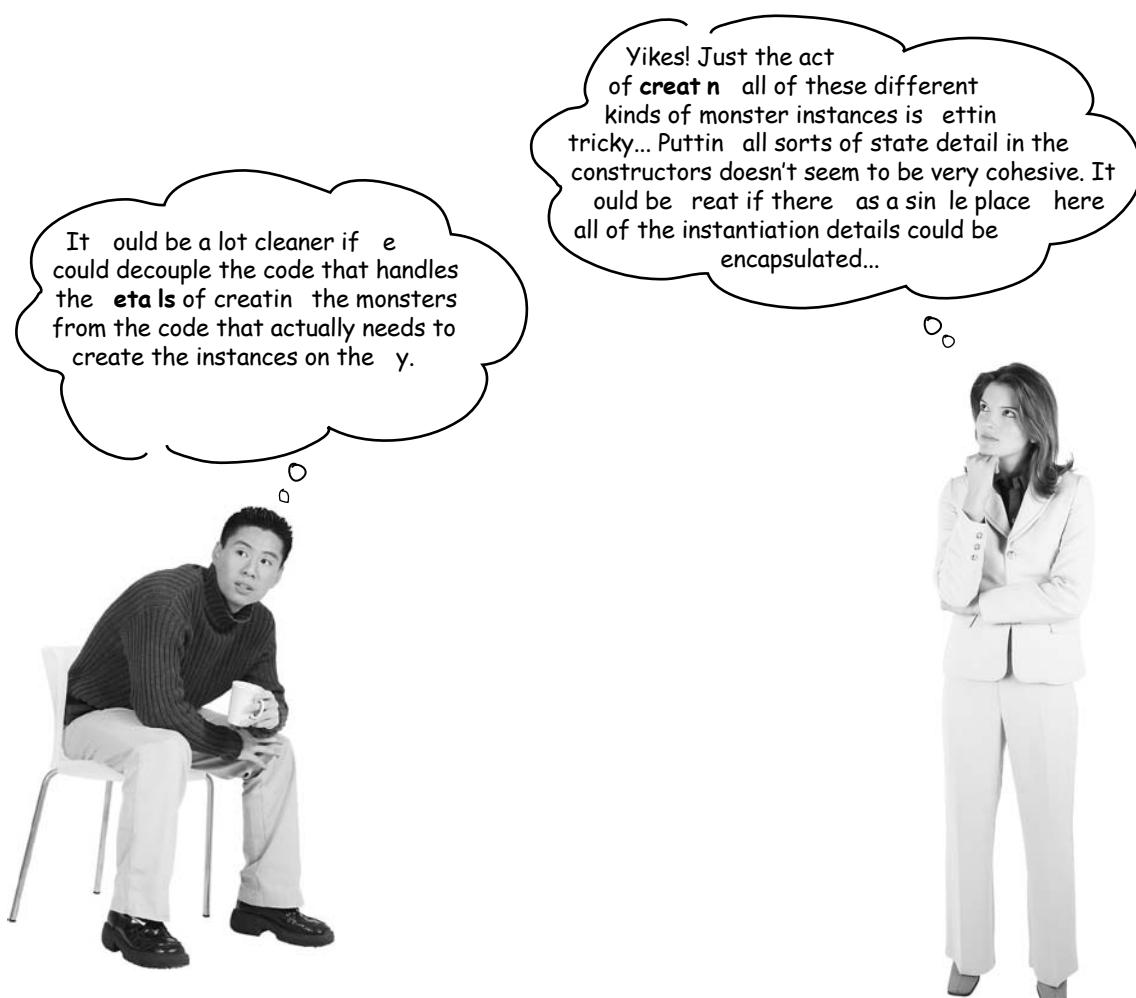
```

## Prototype

se the rotot pe atter whe creat a  
sta ce o a e class s e ther e pe s e or  
compl cated.

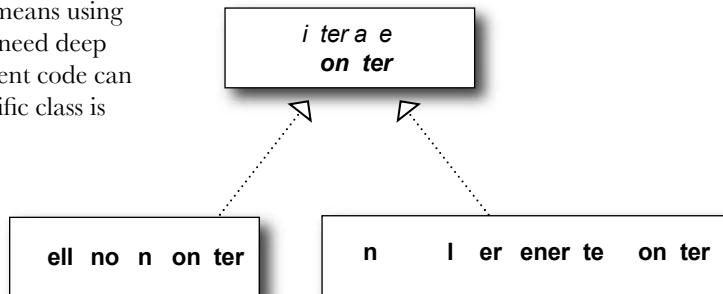
### s nario

our interactive role playing game has an insatiable appetite for monsters. As your heroes make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird like monsters to follow your characters into underseas realms. Finally, you'd like to allow advanced players to create their own custom monsters.



## Prototype to the rescue

The Prototype Pattern allows you to make new instances by copying existing instances. (In Java this typically means using the `clone()` method, or deserialization when you need deep copies.) A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.



```

on ter er

main()
{
    RandomMonster m = 
        Registry.getInstance().getMonster();
}
  
```

The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)

```

on ter e tr

Monster getMonster()
{
    // find the correct monster
    return correctMonster.clone();
}
  
```

The registry finds the appropriate monster, makes a clone of it, and returns the clone.

**rotofy**      n ts

- 
- 
- 

**rotofy**      s s an ra a s

- 
-

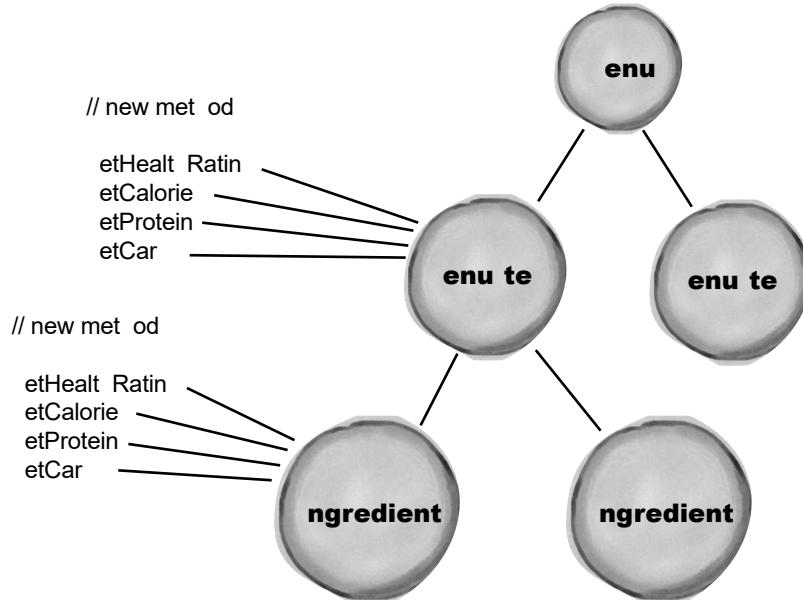
## Visitor

se the stor atter whe o wa t to  
add capab l t es to a compos te o ob ects  
a de caps lat o s ot importa t.

### s nario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per ingredient basis.

### ou's ro os so ution



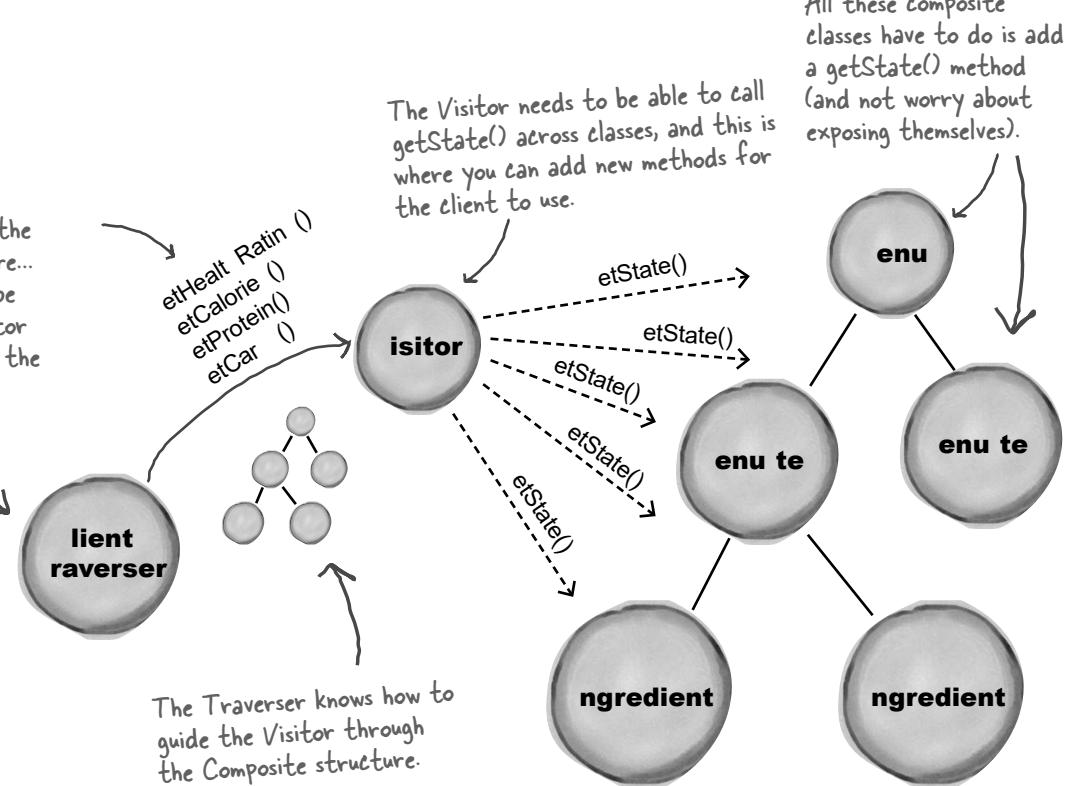
### 's on rns

Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places...

## The Visitor drops by

The visitor must visit each element of the Composite; that functionality is in a traverser object. The visitor is guided by the traverser to gather state from all of the objects in the Composite. Once state has been gathered, the Client can have the visitor perform various operations on the state. When new functionality is required, only the visitor must be enhanced.

The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.



### Visitor Benefits

- 
- 
- 

### Visitor Drawbacks

- 
-





# n e

Abstract Factory Pattern 156. *See also* Factory Pattern

Adapter Pattern

advantages 242

class adapters 244

class diagram 243

combining patterns 504

defined 243

duck magnets 245

Enumeration Iterator Adapter 248

exercise 251

explained 241

fireside chat 247, 252–253

introduction 237

object adapters 244

Alexander, Christopher 602

annihilating evil 606

Anti-Patterns 606–607

Golden Hammer 607

application patterns 604

architectural patterns 604

Bridge Pattern 612–613

Builder Pattern 614–615

bullet points 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608

business process patterns 605

## C

CD Cover Viewer 463

Chain of Responsibility Pattern 616–617

change 339

anticipating 14

constant in software development 8

identifying 53

Choc-O-Holic, Inc. 175

class explosion 81

code magnets 69, 179, 245, 350

cohesion 339–340

Combining Patterns 500

Abstract Factory Pattern 508

Adapter Pattern 504

class diagram 524

Composite Pattern 513

Decorator Pattern 506

Observer Pattern 516

Command Pattern

class diagram 207

command object 203

defined 206–207

introduction 196

loading the Invoker 201

- Command Pattern, continued  
logging requests 229  
macro command 224  
Null Object 214  
queuing requests 228  
undo 216, 220, 227
- Composite Pattern  
and Iterator Pattern 368  
class diagram 358  
combining patterns 513  
composite behavior 363  
default behavior 360  
defined 356  
interview 376–377  
safety 367  
safety versus transparency 515  
transparency 367, 375
- composition 23, 85, 93, 247, 309
- compound pattern 500, 522
- controlling access 460. *See also* Proxy Pattern
- creating objects 134
- crossword puzzle 33, 76, 163, 187, 231, 271, 310, 378, 490
- cubicle conversation 55, 93, 195, 208, 387, 397, 433, 583–584
- Decorator Pattern  
and Proxy Pattern 472–473  
class diagram 91  
combining patterns 506  
cubicle conversation 93  
defined 91  
disadvantages 101, 104
- fireside chat 252–253  
interview 104  
introduction 88  
in Java I/O 100–101  
structural pattern 591
- Dependency Inversion Principle 139–143  
and the Hollywood Principle 298
- Design Patterns  
Abstract Factory Pattern 156  
Adapter Pattern 243  
benefits 599  
Bridge Pattern 612–613  
Builder Pattern 614–615  
categories 589, 592–593  
Chain of Responsibility Pattern 616–617  
class patterns 591  
Command Pattern 206  
Composite Pattern 356  
Decorator Pattern 91  
defined 579, 581  
discover your own 586–587  
Facade Pattern 264  
Factory Method Pattern 134  
Flyweight Pattern 618–619  
Interpreter Pattern 620–621  
Iterator Pattern 336  
Mediator Pattern 622–623  
Memento Pattern 624–625  
Null Object 214  
object patterns 591  
Observer Pattern 51  
organizing 589  
Prototype Pattern 626–627  
Proxy Pattern 460

- Simple Factory 114
- Singleton Pattern 177
- State Pattern 410
- Strategy Pattern 24
- Template Method Pattern 289
- use 29
- versus frameworks 29
- versus libraries 29
- Visitor Pattern 628–629
- Design Principles. *See* Object Oriented Design Principles
- Design Puzzle 25, 133, 279, 395, 468, 542
- Design Toolbox 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608
- DJ View 534
- domain specific patterns 604
  
- Elvis 526
- encapsulate what varies 8–9, 75, 136, 397, 612
- encapsulating algorithms 286, 289
- encapsulating behavior 11
- encapsulating iteration 323
- encapsulating method invocation 206
- encapsulating object construction 614–615
- encapsulating object creation 114, 136
- encapsulating requests 206
- encapsulating state 399
  
- Facade Pattern
  - advantages 260
  - and Principle of Least Knowledge 269
  - class diagram 264
- defined 264
- introduction 258
- Factory Method Pattern 134. *See also* Factory Pattern
- Factory Pattern
  - Abstract Factory
    - and Factory Method 158–159, 160–161
    - class diagram 156–157
    - combining patterns 508
    - defined 156
    - interview 158–159
    - introduction 153
  - Factory Method
    - advantages 135
    - and Abstract Factory 160–161
    - class diagram 134
    - defined 134
    - interview 158–159
    - introduction 120, 131–132
    - up close 125
  - Simple Factory
    - defined 117
    - introduction 114
  - family of algorithms. *See* Strategy Pattern
  - family of products 145
  - favor composition over inheritance 23, 75
  - fireside chat 62, 247, 252, 308, 418, 472–473
  - Five minute drama 48, 478
  - Flyweight Pattern 618–619
  - forces 582
  - Friedman, Dan 171
- Gamma, Erich 601

- Gang of Four 583, 601  
Gamma, Erich 601  
Helm, Richard 601  
Johnson, Ralph 601  
Vlissides, John 601  
global access point 177  
gobble gobble 239  
Golden Hammer 607  
guide to better living with Design Patterns 578  
Gumball Machine Monitor 431
- HAS-A 23  
Head First learning principles xxx  
Helm, Richard 601  
Hillside Group 603  
Hollywood Principle, The 296  
    and the Dependency Inversion Principle 298  
Home Automation or Bust, Inc. 192  
Home Sweet Home Theater 255  
Hot or Not 475
- inheritance  
    disadvantages 5  
    for reuse 5–6  
    versus composition 93  
interface 12  
Interpreter Pattern 620–621  
inversion 141–142  
IS-A 23  
Iterator Pattern  
    advantages 330
- and collections 347–349  
and Composite Pattern 368  
and Enumeration 338  
and Hashtable 343, 348  
class diagram 337  
code magnets 350  
defined 336  
exercise 327  
external iterator 338  
for/in 349  
internal iterator 338  
introduction 325  
java.util.Iterator 332  
Null Iterator 372  
polymorphic iteration 338  
removing objects 332
- Johnson, Ralph 601
- KISS 594
- Law of Demeter. *See* Principle of Least Knowledge  
lazy instantiation 177  
loose coupling 53
- magic bullet 594  
master and student 23, 30, 85, 136, 592, 596  
Matchmaking in Objectville 475

- Mediator Pattern 622–623  
 Memento Pattern 624–625  
 middleman 237  
 Mighty Gumball, Inc. 386  
 Model-View-Controller  
     Adapter Pattern 546  
     and design patterns 532  
     and the Web 549  
     Composite Pattern 532, 559  
     introduction 529  
     Mediator Pattern 559  
     Observer Pattern 532  
     ready-bake code 564–576  
     song 526  
     Strategy Pattern 532, 545  
     up close 530  
 Model 2 549. *See also* Model-View-Controller  
     and design patterns 557–558  
 MVC. *See* Model-View-Controller
- Null Object 214, 372
- Objectville Diner 26, 197, 316, 628  
 Objectville Pancake House 316, 628  
 Object Oriented Design Principles 9, 30–31  
     Dependency Inversion Principle 139–143  
     encapsulate what varies 9, 111  
     favor composition over inheritance 23, 243, 397  
     Hollywood Principle 296  
     one class, one responsibility 185, 336, 339, 367  
     Open-Closed Principle 86–87, 407
- Principle of Least Knowledge 265  
 program to an interface, not an implementation 11, 243, 335  
 strive for loosely coupled designs between objects that interact 53
- Observable 64, 71  
 Observer Pattern  
     class diagram 52  
     code magnets 69  
     combining patterns 516  
     cubicle conversation 55  
     defined 51–52  
     fireside chat 62  
     Five minute drama 48  
     introduction 44  
     in Swing 72–73  
     Java support 64  
     pull 63  
     push 63  
     one-to-many relationship 51–52  
 OOPSLA 603  
 Open-Closed Principle 86–87  
 oreo cookie 526  
 organizational patterns 605
- part-whole hierarchy 356. *See also* Composite Pattern  
 patterns catalog 581, 583, 585  
 Patterns Exposed 104, 158, 174, 377–378  
 patterns in the wild 299, 488–489  
 patterns zoo 604  
 Pattern Honorable Mention 117, 214  
 Pizza shop 112  
 Portland Patterns Repository 603

Principle of Least Knowledge 265–268

disadvantages 267

program to an implementation 12, 17, 71

program to an interface 12

program to an interface, not an implementation 11, 75

Prototype Pattern 626–627

Proxy Pattern

and Adapter Pattern 471

and Decorator Pattern 471, 472–473

Caching Proxy 471

class diagram 461

defined 460

Dynamic Proxy 474, 479, 486

and RMI 486

exercise 482

fireside chat 472–473

`java.lang.reflect.Proxy` 474

Protection Proxy 474, 477

Proxy Zoo 488–489

ready-bake code 494

Remote Proxy 434

variants 471

Virtual Proxy 462

image proxy 464

publisher/subscriber 45

Quality, The. *See* Quality without a name

Quality without a name. *See* Quality, The

refactoring 354, 595

remote control 193, 209

Remote Method Invocation. *See* RMI

remote proxy 434. *See also* Proxy Pattern

reuse 13, 23, 85

RMI 436

shared vocabulary 26–28, 599–600

sharpen your pencil 5, 42, 54, 61, 94, 97, 99, 124, 137, 148, 176, 183, 205, 225, 242, 268, 284, 322, 342, 396, 400, 406, 409, 421, 483, 511, 518, 520, 589

Simple Factory 117

SimUDuck 2, 500

Singleton Pattern

advantages 170, 184

and garbage collection 184

and global variables 185

and multithreading 180–182

class diagram 177

defined 177

disadvantages 184

double-checked locking 182

interview 174

up close 173

Single Responsibility Principle 339. *See also* Object Oriented Design Principles: one class, one responsibility

skeleton 440

Starbuzz Coffee 80, 276

state machines 388–389

State Pattern

and Strategy Pattern 411, 418–419

class diagram 410

defined 410

- disadvantages 412, 417
- introduction 398
- sharing state 412
- static factory 115
- Strategy Pattern 24
  - and State Pattern 411, 418–419
  - and Template Method Pattern 308–309
  - encapsulating behavior 22
  - family of algorithms 22
  - fireside chat 308
  - stub 440
- Template Method Pattern
  - advantages 288
  - and Applet 307
  - and `java.util.Arrays` 300
  - and Strategy Pattern 305, 308–309
  - and Swing 306
  - and the Hollywood Principle 297
  - class diagram 289
  - defined 289
  - fireside chat 308–309
  - hook 292, 295
  - introduction 286
  - up close 290–291
- The Little Lisper 171
- thinking in patterns 594–595
- tightly coupled 53

## U

- undo 216, 227
- user interface design patterns 605

# *olophon*



Kat y and Bert created t e loo & eel o t e Head Fir t erie .  
T e oo wa prod ced in Ado e InDe i n CS (an n elie a ly cool de i n tool t at we can't et  
eno o ) and Ado e P oto op CS. T e oo wa type et in Uncle Stin y, Mi ter Fri y(yo t in  
we re iddin ), Ann Satellite, Ba er ille, Comic San , Myriad Pro, S ippy S arp, Sa oye LET, Jo erman  
LET, Co rier New and Woodrow type ace .

Interior de in and production all happened easily on Apple Macintosh at Head First we're all about T in Different (even identical grammatical). All Java code was created in James Gosling's IDE, i.e., we really should try Eric Gamma Eclipse.

Long day or writing were powered by the caffeine in Honey Tea and Tea-a, the clean Santa Fe air, and the roof in London and the Banco de Gaia, Cocteau Twins, Bad Bar I-VI, Delerium, Enigma, Nine Old Men, Olive Oil, Orange, Italian, LTJ Brem, Magazine Attac, Steel Roach, Saab and Discreet, Tie Ery Corporation, Echo 7 and Neil Finn (in all his incarnations) along with a few other acid trance and more 80s music tracks to now and then.

## And now, a final word from Head First Labs...

Our world class researchers are working day and night in a mad race to uncover the mysteries of Life, the Universe and Everything—before it's too late. Never before has a research team with such noble and daunting goals been assembled. Currently, we are focusing our collective energy and brain power on creating the ultimate learning machine. Once perfected, you and others will join us in our quest!

You're fortunate to be holding one of our first prototypes in your hands. But only through constant refinement can our goal be achieved. We ask you, a pioneer user of the technology, to send us periodic field reports of your progress, at [fieldreports@headfirrlabs.com](mailto:fieldreports@headfirrlabs.com)



## Now that you've applied the Head First approach to Design Patterns, why not apply it to the rest of your life?

Come join us at the Head First Labs Web site, our virtual hangout where you'll find Head First resources including podcasts, forums, code and more.

But you won't just be a spectator; we also encourage you to join the fun by participating in discussions and brainstorming.

### What's in it for you?

- Get the latest news about what's happening in the Head First World.
- Participate in our upcoming books and technologies.
- Learn how to tackle those tough technical topics (say that three times fast) in as little time as possible.
- Look behind the scenes at how Head First books are created.
- Meet the Head First authors and the support team who keep everything running smoothly.
- Why not audition to be a Head First author yourself?



<http://www.headfirstlabs.com>

# Better than e-books

Try it  
**FREE!**  
Sign up today and get your first 14 days free.  
[safari.oreilly.com](http://safari.oreilly.com)

## Search

inside electronic versions  
of thousands of books

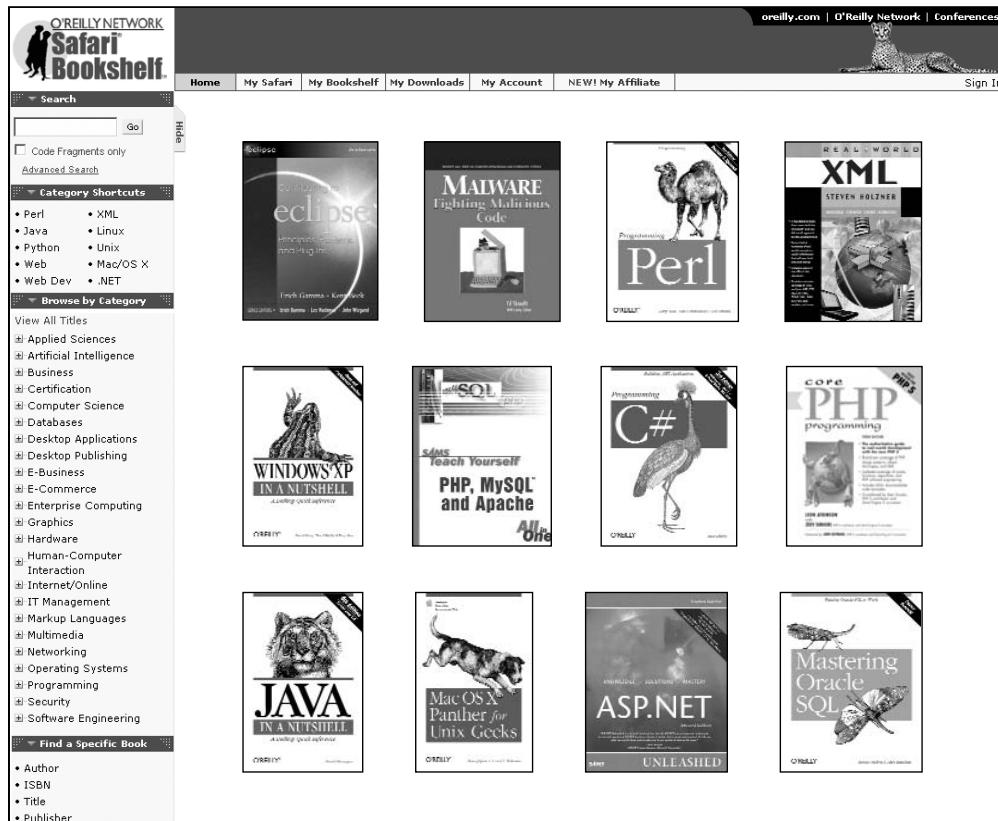
## Browse

books by category.  
With Safari researching  
any topic is a snap

## Find

answers in an instant

Read books from cover  
to cover. Or, simply click  
to the page you need.



**Search Safari! The premier electronic reference  
library for programmers and IT professionals**



Addison  
Wesley  
AdobePress

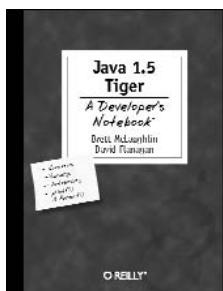
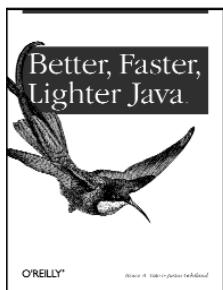
Sum  
microsystems  
O'REILLY  
SAMS  
New Riders

ALPHA  
Java  
Sun  
Microsystems  
QUE

Microsoft  
Press  
Cisco Press

Peachpit  
Press  
macromedia  
PRESS  
PRINTICE  
PTR

# Related Titles Available from O'Reilly



## Java

- Ant: The Definitive Guide  
Better, Faster, Lighter Java  
Eclipse  
Eclipse Cookbook  
Enterprise JavaBeans,  
*4th Edition*  
Hardcore Java  
Head First Java  
Head First Servlets & JSP  
Head First EJB  
Hibernate:  
A Developer's Notebook  
J2EE Design Patterns  
Java 1.5 Tiger:  
A Developer's Notebook  
Java & XML Data Binding  
Java & XML  
Java Cookbook, *2nd Edition*  
Java Data Objects  
Java Database Best Practices  
Java Enterprise Best Practices  
Java Enterprise in a Nutshell,  
*2nd Edition*  
Java Examples in a Nutshell,  
*3rd Edition*  
Java Extreme Programming  
Cookbook  
Java in a Nutshell, *4th Edition*  
Java Management Extensions  
Java Message Service  
Java Network Programming,  
*2nd Edition*  
Java NIO  
Java Performance Tuning,  
*2nd Edition*  
Java RMI  
Java Security, *2nd Edition*  
JavaServer Faces  
Java ServerPages, *2nd Edition*  
Java Servlet & JSP Cookbook  
Java Servlet Programming,  
*2nd Edition*  
Java Swing, *2nd Edition*  
Java Web Services in a Nutshell  
Learning Java, *2nd Edition*  
Mac OS X for Java Geeks  
Programming Jakarta Struts  
*2nd Edition*  
Tomcat: The Definitive Guide  
WebLogic:  
The Definitive Guide

---

O'REILLY®

Our books are available at most retail and online bookstores.  
To order direct: 1-800-998-9938 • [order@oreilly.com](mailto:order@oreilly.com) • [www.oreilly.com](http://www.oreilly.com)  
Online editions of most O'Reilly titles are available by subscription at [safari.oreilly.com](http://safari.oreilly.com)

# Keep in touch with O'Reilly

## 1. Download examples from our books

To find example files for a book, go to:  
[www.oreilly.com/catalog](http://www.oreilly.com/catalog)  
select the book, and follow the "Examples" link.

## 2. Register your O'Reilly books

Register your book at [register.oreilly.com](http://register.oreilly.com)  
Why register your books?  
Once you've registered your O'Reilly books you can:

- Win O'Reilly books, T-shirts or discount coupons in our monthly drawing.
- Get special offers available only to registered O'Reilly customers.
- Get catalogs announcing new books (US and UK only).
- Get email notification of new editions of the O'Reilly books you own.

## 3. Join our email lists

Sign up to get topic-specific email announcements of new books and conferences, special offers, and O'Reilly Network technology newsletters at:  
[elists.oreilly.com](http://elists.oreilly.com)

It's easy to customize your free elists subscription so you'll get exactly the O'Reilly news you want.

## 4. Get the latest news, tips, and tools

[www.oreilly.com](http://www.oreilly.com)

- "Top 100 Sites on the Web"—PC Magazine
- CIO Magazine's Web Business 50 Awards

Our web site contains a library of comprehensive product information (including book excerpts and tables of contents), downloadable software, background articles, interviews with technology leaders, links to relevant sites, book cover art, and more.

## 5. Work for O'Reilly

Check out our web site for current employment opportunities:  
[jobs.oreilly.com](http://jobs.oreilly.com)

## 6. Contact us

O'Reilly & Associates  
1005 Gravenstein Hwy North  
Sebastopol, CA 95472 USA  
TEL: 707-827-7000 or 800-998-9938  
(6am to 5pm PST)  
FAX: 707-829-0104

**order@oreilly.com**  
For answers to problems regarding your order or our products. To place a book order online, visit:

[www.oreilly.com/order\\_new](http://www.oreilly.com/order_new)

**catalog@oreilly.com**  
To request a copy of our latest catalog.

**booktech@oreilly.com**  
For book content technical questions or corrections.

**corporate@oreilly.com**  
For educational, library, government, and corporate sales.

**proposals@oreilly.com**  
To submit new book proposals to our editors and product managers.

**international@oreilly.com**  
For information about our international distributors or translation queries. For a list of our distributors outside of North America check out:

[international.oreilly.com/distributors.html](http://international.oreilly.com/distributors.html)

**adoption@oreilly.com**  
For information about academic use of O'Reilly books, visit:  
[academic.oreilly.com](http://academic.oreilly.com)



Our books are available at most retail and online bookstores.

To order direct: 1-800-998-9938 • [order@oreilly.com](mailto:order@oreilly.com) • [www.oreilly.com](http://www.oreilly.com)

Online editions of most O'Reilly titles are available by subscription at [safari.oreilly.com](http://safari.oreilly.com)