Table of Contents (summary)

Table of Contents (the real thing)

Intro

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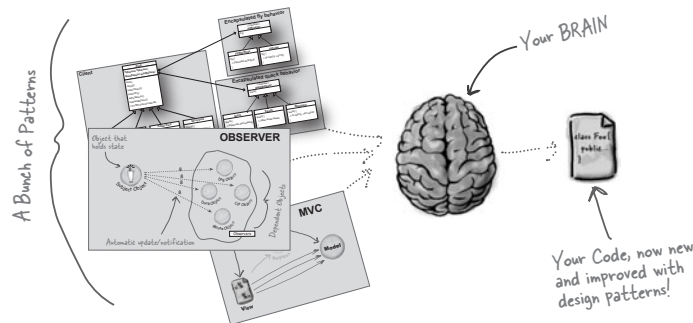
1 Welcome to Design Patterns

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

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

A Bunch of Patterns

OBSERVER

MVC

Your BRAIN

Your Code, now new and improved with design patterns!

the Observer Pattern

2 Keeping your Objects in the Know

Don't miss out when something interesting happens!

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

OO Basics

Abstraction
...tion
...hism
...ce

OO Principles

Encapsulate what varies

Favor Composition over inheri-tance

Program to Interfaces, not implementations

Strive for loosely coupled designs between objects that interact

ONE TO MANY RELATIONSHIP

Object that holds state

Subject Object

int

Dog Object

Duck Object

Cat Object

Mouse Object

Observers

Dependent Objects

Automatic update/notification

3

Decorating Objects

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

I used to think real men subclassed everything. That was until I learned the power of extension at runtime, rather than at compile time. Now look at me!

the Factory Pattern

Baking with OO Goodness

Get ready to cook some loosely coupled OO designs.

4

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

5 One of a Kind Objects

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

Hershey, PA

OO Patterns

Singleton – Ensure a class only has one instance and provide a global point of access to it.

the Command Pattern

6

Encapsulating Invocation

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

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

The Order consists of an order slip and the customer's menu items that are written on it

I'll have a Burger with Cheese and a Malt Shake

createOrder()

takeOrder()

Start Here

The customer knows what he wants and creates an order.

The Waitress takes the Order, and when she gets around to it, she calls its orderUp() method to begin the Order's preparation.

orderUp()

The Order has all the instructions needed to prepare the meal. The Order directs the Short Order Cook with methods like makeBurger().

makeBurger(), makeShake()

The Short Order Cook follows the instructions of the Order and produces the meal.

output

the Adapter and Facade Patterns

7 Being Adaptive

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

European Wall Outlet

AC Power Adapter

Standard AC Plug

Client — request() — translatedRequest() — Adaptee

The Client is implemented against the target interface

target interface

Adapter

The Adapter implements the target interface and holds an instance of the Adaptee

adaptee interface

Turkey was the adaptee interface

the Template Method Pattern

8 Encapsulating Algorithms

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

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

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea
1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee
1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize

generalize

Caffeine Beverage
1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

2. Steep the teabag in the water
4. Add lemon

Coffee subclass

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

9 Well-Managed Collections

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

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

the State Pattern

The State of Things

10

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

the Proxy Pattern

11 Controlling Object Access

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

The proxy now consists of two classes.

Compound Patterns

Patterns of Patterns

Who would have ever guessed that Patterns could work together?

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

The beat is set at 119 BPM and you would like to increase it to 120.

You click on the increase beat button.

View

Which results in the controller being invoked.

Controller

The controller asks the model to update its BPM by one.

BeatModel
on()
setBPM() off()
getBPM()

You see the beatbar pulse every 1/2 second.

View

Because the BPM is 120, the view gets a beat notification every 1/2 second.

The view is updated to 120 BPM.

View is notified that the BPM changed. It calls getBPM() on the model state.

13 Patterns in the Real World

Ahhhh, now you're ready for a bright new world filled with Design Patterns.

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

The Objectville Guide to Better Living with Design Patterns

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

- Learn the oft 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 on and 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 Design Patterns mind like a Zen master.
- Win friends and influence developers by improving your patterns vocabulary.

Richard Helm

Ralph Johnson

Gang of Four

John Vlissides

Erich Gamma

GOF

14 Appendix: Leftover Patterns

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

The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.

The Visitor needs to be able to call getState() across classes, and this is where you can add new methods for the client to use.

All these composite classes have to do is add a getState() method (and not worry about exposing themselves :).

The Traverser knows how to guide the Visitor through the Composite structure.

getHealthRating()
getCalories()
getProtein()
getCarbs()

getState()
getState()
getState()
getState()

Visitor
Client / Traverser
Menu
MenuItem
MenuItem
Ingredient
Ingredient

i Index

Head First
Design Patterns

Avoid those
embarrassing
coupling mistakes

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

Discover the secrets
of the Patterns Guru

Load the patterns
that matter straight
into your brain

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

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

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

O'REILLY®

3 the Decorator Pattern

✳ *Decorating Objects* ✳

> I used to think real men subclassed everything. That was until I learned the power of extension at runtime, rather than at compile time. Now look at me!
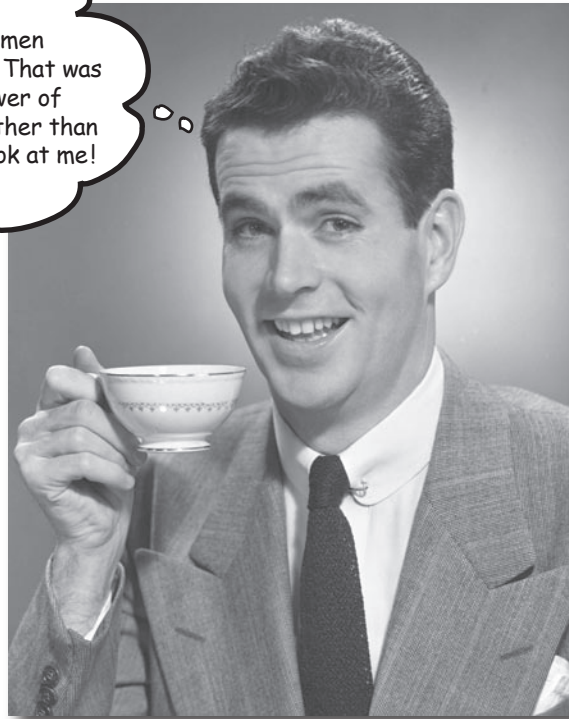
Just call this chapter "Design Eye for the Inheritance Guy."

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

Welcome to Starbuzz Coffee

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

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

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

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost() method is abstract; subclasses need to define their own implementation.

The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.

Beverage

description

getDescription()
cost()

// Other useful methods...

HouseBlend	DarkRoast	Decaf	Espresso
cost()	cost()	cost()	cost()

Each subclass implements cost() to return the cost of the beverage.

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

Here's their first attempt...

Beverage

description

getDescription()
cost()

// Other useful methods...

HouseBlendWithSteamedMilkandMocha — cost()
DarkRoastWithSteamedMilkandMocha — cost()
DecafWithSteamedMilkandMocha — cost()
EspressoWithSteamedMilkandMocha — cost()
HouseBlend... — cost()
DarkRoastWithSteamedMilkandCaramel — cost()
DecafWithSteamedMilkandCaramel — cost()
EspressoWithSteamedMilkandCaramel — cost()
EspressoWithWhipandMocha — cost()
HouseBlendWithSteamed... — cost()
HouseBlendWit... — cost()
DarkRoastWith... — cost()
DecafWith... — cost()
DecafWithSoy — cost()
HouseBlendWith...andS — cost()
DarkRoastWithSteamedMilkandSoy — cost()
DecafWithSteamedMilkandS — cost()
EspressoWithS... — cost()
HouseBlendWith... — cost()
HouseBlendWithWhip — cost()
HouseBl... — cost()
DarkRoastWithSteamedM... — cost()
DarkRoas... — cost()
DecafWithSteamedMilk — cost()
Deca... — cost()
DecafWithSoyandMocha — cost()
HouseBlendWithWhipandSoy — cost()
DarkRoastWith... — cost()
D... — cost()
EspressoWithSteamedMilkandWhip — cost()
DarkRoastWithSteamedMilkandWhip — cost()
DecafWithSteamed... — cost()
EspressoWithWhipandSoy — cost()
DarkRoastWithWhipandSoy — cost()
DecafWithWhipandSoy — cost()

Whoa! Can you say "class explosion?"

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

BRAIN
POWER

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

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

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

This is stupid; why do we need all these classes? Can't we just use instance variables and inheritance in the superclass to keep track of the condiments?

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

Beverage
description milk soy mocha whip
getDescription() cost() hasMilk() setMilk() hasSoy() setSoy() hasMocha() setMocha() hasWhip() setWhip() // Other useful methods..

New boolean values for each condiment.

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

These get and set the boolean values for the condiments.

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

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

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

```
                    Beverage
 ─────────────────────────────────
 description
 milk
 soy
 mocha
 whip
 ─────────────────────────────────
 getDescription()
 cost()

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

 // Other useful methods..
```

HouseBlend	DarkRoast	Decaf	Espresso
cost()	cost()	cost()	cost()
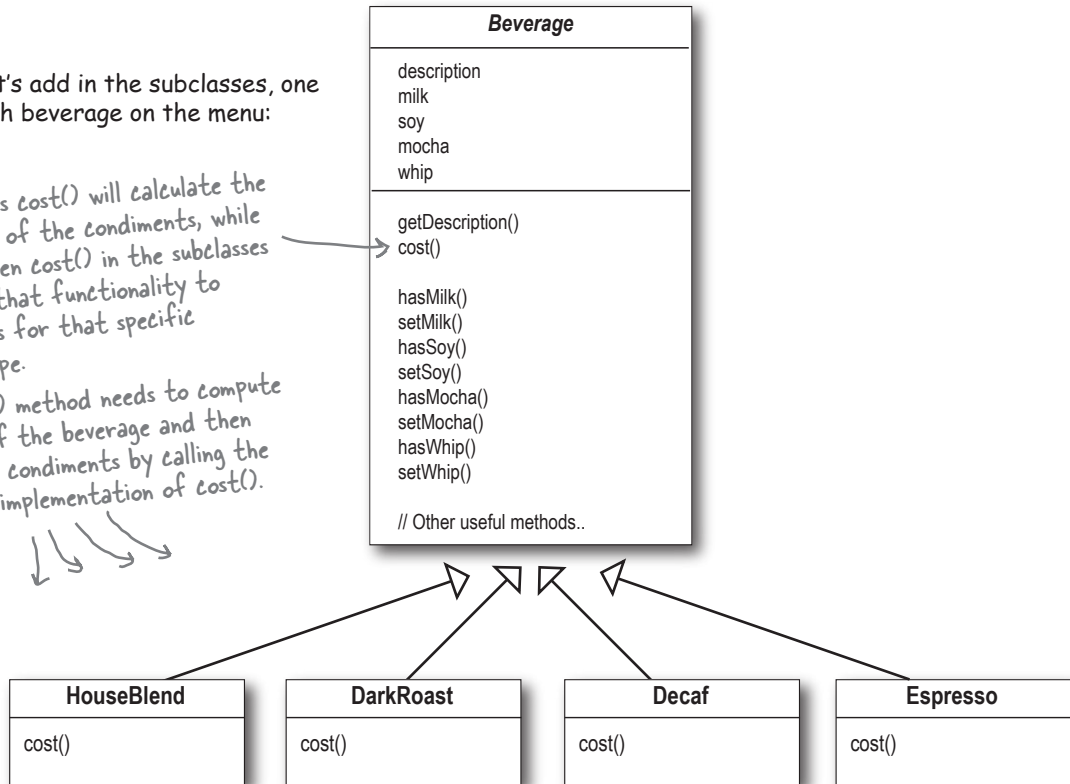
Sharpen your pencil

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() {








    }
}
```

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

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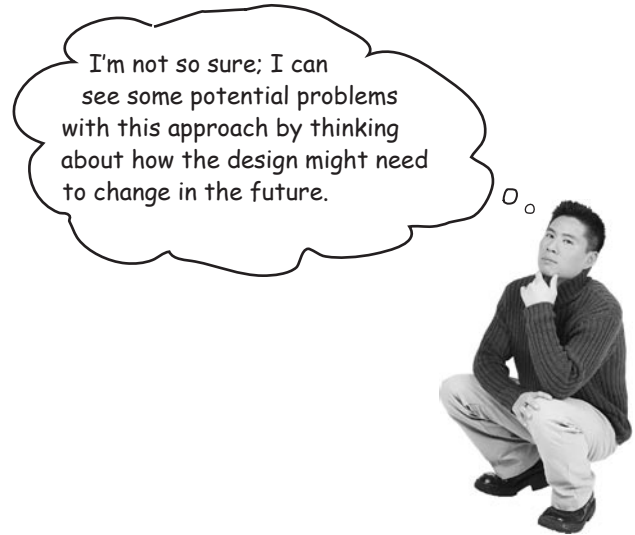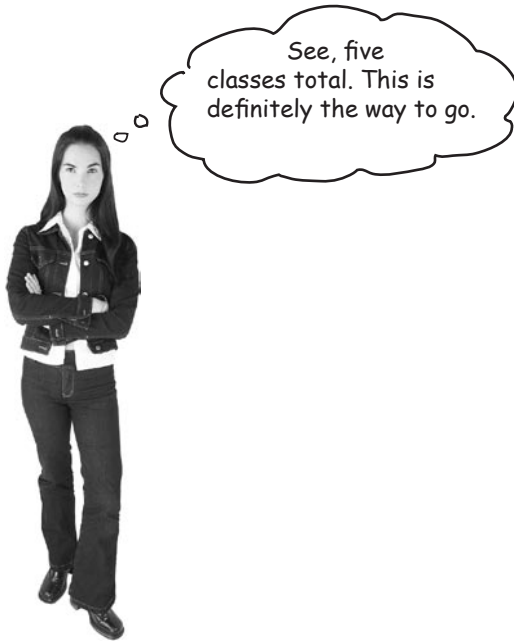
Sharpen your pencil

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

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

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

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

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

What if a customer wants a double mocha?
_____

Your turn:
_____

_____

Master and Student...

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

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

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

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

Master: *Please, go on...*

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

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

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

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

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

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

The Open-Closed Principle

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

> **Design Principle**
>
> *Classes should be open for extension, but closed for modification.*

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

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

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

there are no
Dumb Questions

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

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

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

Q: **Okay, I understand 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 your code from being modified by supplying a means of extension. In this chapter you'll see a good example of using the Decorator pattern to follow the Open-Closed principle.

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

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

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

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

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

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

Meet the Decorator Pattern

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

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:

1 **Take a DarkRoast object**

2 **Decorate it with a Mocha object**

3 **Decorate it with a Whip object**

4 **Call the cost() method and rely on delegation to add on the condiment costs**

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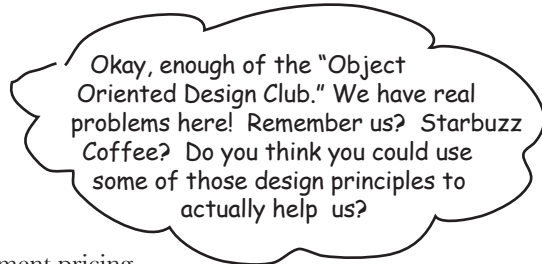
Constructing a drink order with Decorators

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

cost()
DarkRoast

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.**

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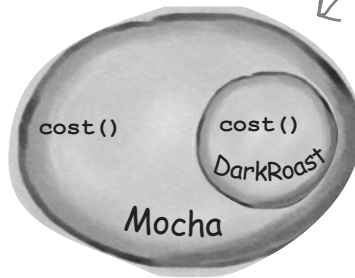
The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type...)

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.**

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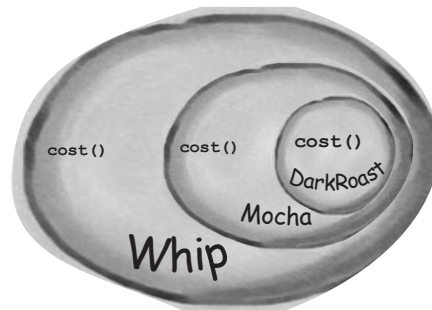
Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

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

④ **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

(You'll see how in a few pages.)

② Whip calls cost() on Mocha.

① First, we call cost() on the outmost decorator, Whip.

③ Mocha calls cost() on DarkRoast.

$1.29 ← .10   cost()   .20   cost()   .99   cost()
DarkRoast
Mocha
Whip

④ DarkRoast returns its cost, 99 cents.

⑤ Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

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

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

- Decorators have the same supertype as the objects they decorate.

- You can use one or more decorators to wrap an object.

- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.

- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.

Key Point!

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

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

The Decorator Pattern defined

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

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

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

Each component can be used on its own, or wrapped by a decorator.

component

Component

methodA()
methodB()
// other methods

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

ConcreteComponent

methodA()
methodB()
// other methods

Decorator

methodA()
methodB()
// other methods

Decorators implement the same interface or abstract class as the component they are going to decorate.

ConcereteDecoratorA

Component wrappedObj

methodA()
methodB()
newBehavior()
// other methods

The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).

ConcreteDecoratorB

Component wrappedObj
Object newState

methodA()
methodB()
// other methods

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Decorating our Beverages

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

Beverage acts as our abstract component class.

Beverage

description

getDescription()
cost()
// other useful methods

component

HouseBlend

cost()

DarkRoast

cost()

Espresso

cost()

Decaf

cost()

The four concrete components, one per coffee type.

CondimentDecorator

getDescription()

Milk

Beverage beverage

cost()
getDescription()

Mocha

Beverage beverage

cost()
getDescription()

Soy

Beverage beverage

cost()
getDescription()

Whip

Beverage beverage

cost()
getDescription()

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

BRAIN POWER

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

Cubicle Conversation

Some confusion over Inheritance versus Composition

> Okay, I'm a little confused...I thought we weren't going to use inheritance in this pattern, but rather we were going to rely on composition instead.
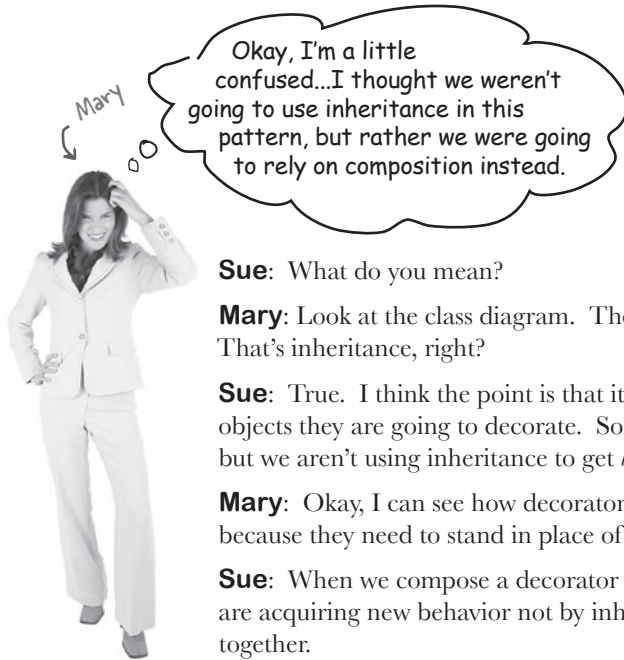
Mary

Sue: What do you mean?

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

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

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

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

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

Sue: That's right.

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

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

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

Sue: Exactly.

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

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

New barista training

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

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

② Whip calls cost() on Mocha.

① First, we call cost() on the outmost decorator, Whip.

③ Mocha calls cost() on DarkRoast.
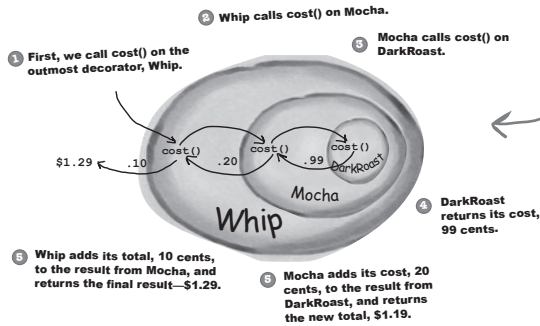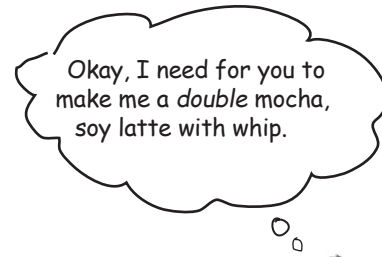
This picture was for a "dark roast mocha whip" beverage.

cost()   .20   cost()   .99   cost()
$1.29 ← .10   DarkRoast
                    Mocha
        Whip

④ DarkRoast returns its cost, 99 cents.

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

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

Sharpen your pencil

Draw your picture here.

Starbuzz Coffee

Coffees

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

Condiments

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

HINT: You can make a "double mocha soy latte with whip" by combining HouseBlend, Soy, two shots of Mocha and Whip!

Writing the Starbuzz code

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

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

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

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

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

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

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

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

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

Coding beverages

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

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

```
public class Espresso extends Beverage {

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

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

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

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

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

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

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

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

Starbuzz Coffee

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

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

Coding condiments

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

Mocha is a decorator, so we extend CondimentDecorator.

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.

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

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

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

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

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

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

Serving some coffees

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

Here's some test code* to make orders:

```java
public class StarbuzzCoffee {

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

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

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

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

Make a DarkRoast object.
Wrap it with a Mocha.
Wrap it in a second Mocha.
Wrap it in a Whip.

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

*We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns.

Now, let's get those orders in:

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

there are no
Dumb Questions

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

A: That is exactly right. If you have code that relies on the concrete component's type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

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

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

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

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

✏ Sharpen your pencil

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

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

Real World Decorators: Java I/O

The large number of classes in the java.io package is... *overwhelming*. Don't feel alone if you said "whoa" the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the java.io package is largely based on Decorator. Here's a typical set of objects that use decorators to add functionality to reading data from a file:

A text file for reading.

FileInputStream

BufferedInputStream

LineNumberInputStream

FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

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

Decorating the java.io classes

Here's our abstract component.

InputStream

FilterInputStream is an abstract decorator.

FileInputStream	**StringBufferInputStream**	**ByteArrayInputStream**	**FilterInputStream**

PushbackInputStream	**BufferedInputStream**	**DataInputStream**	**LineNumberInputStream**

These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.

And finally, here are all our concrete decorators.

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

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

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

Writing your own Java I/O Decorator

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

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

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

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

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

```java
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;
    }
}
```

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

Test out your new Java I/O Decorator

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

```java
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

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

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

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

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

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

test.txt file

You need to make this file.

Give it a spin:

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

Patterns Exposed

This week's interview:
Confessions of a Decorator

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

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

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

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small classes to a design and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

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

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

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

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

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

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

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

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

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

Tools for your Design Toolbox

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

OO Basics
duction
ulation
rphism
ance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

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

OO Patterns

Stra
d
encap
w
inter
d
vary i
a

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

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

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.

- In our designs we should allow behavior to be extended without the need to modify existing code.

- Composition and delegation can often be used to add new behaviors at runtime.

- The Decorator Pattern provides an alternative to subclassing for extending behavior.

- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.

- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)

- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.

- You can wrap a component with any number of decorators.

- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.

- Decorators can result in many small objects in our design, and overuse can be complex.

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 float cost() {

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

```
public class DarkRoast extends Beverage {

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

    public float cost() {

       return 1.99 + super.cost();

    }
}
```

New barista training

"double mocha soy lotte with whip"
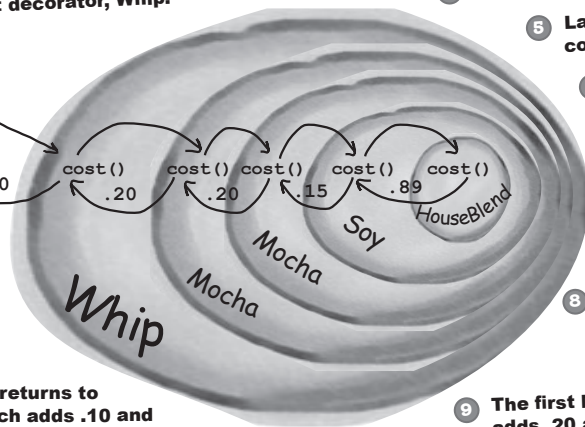
2 Whip calls cost() on Mocha

3 Mocha calls cost() on another Mocha.

4 Next, Mocha calls cost() on Soy.

1 First, we call cost() on the outmost decorator, Whip.

5 Last topping! Soy calls cost() on HouseBlend.

6 HouseBlend's cost() method returns .89 cents and pops off the stack.

7 Soy's cost() method adds .15 and returns the result, and pops off the stack.

8 The second Mocha's cost() method adds .20 and returns the result, and pops off the stack.

9 The first Mocha's cost() method adds .20 and returns the result, and pops off the stack.

11 Finally, the result returns to Whip's cost(), which adds .10 and we have a final cost of $1.54.

$1.54 ← .10   cost() .20   cost() .20   cost() .15   cost() .89   cost()

Whip   Mocha   Mocha   Soy   HouseBlend

Exercise solutions

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

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

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

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

    public 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.

5 the Singleton Pattern

✦ One of a Kind Objects ✦

I tell ya she's ONE OF A KIND. Look at the lines, the curves, the body, the headlights!

You talkin' to me or the car? Oh, and when can I get my oven mitt back?

Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance. You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact, the diagram holds just a single class! But don't get too comfortable; despite its simplicity from a class design perspective, we are going to encounter quite a few bumps and potholes in its implementation. So buckle up.

What is this? An entire chapter about how to instantiate just ONE OBJECT!

That's one and ONLY ONE object.
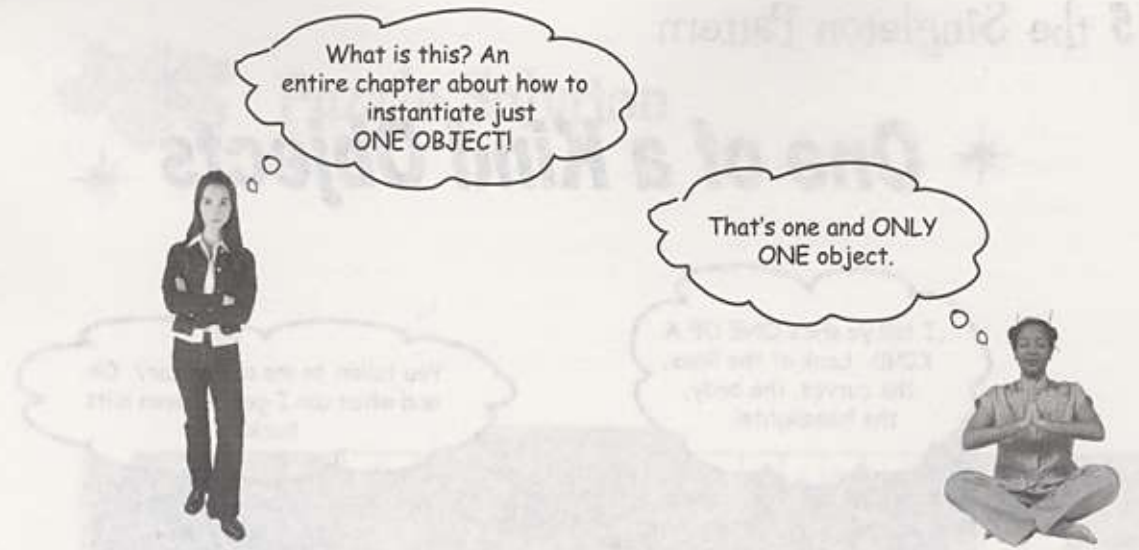
Developer: What use is that?

Guru: There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

Developer: Okay, so maybe there are classes that should only be instantiated once, but do I need a whole chapter for this? Can't I just do this by convention or by global variables? You know, like in Java, I could do it with a static variable.

Guru: In many ways, the Singleton Pattern **is a convention** for ensuring one and only one object is instantiated for a given class. If you've got a better one, the world would like to hear about it; but remember, like all patterns, the Singleton Pattern is a time-tested method for ensuring only one object gets created. The Singleton Pattern also gives us a global point of access, just like a global variable, but without the downsides.

Developer: What downsides?

Guru: Well, here's one example: if you assign an object to a global variable, then you have to create that object when your application begins*. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.

Developer: This still doesn't seem like it should be so difficult.

Guru: If you've got a good handle on static class variables and methods as well as access modifiers, it's not. But, in either case, it is interesting to see how a Singleton works, and, as simple as it sounds, Singleton code is hard to get right. Just ask yourself: how do I prevent more than one object from being instantiated? It's not so obvious, is it?

*This is actually implementation dependent. Some JVM's will create these objects lazily.

The Little Singleton
A small Socratic exercise in the style of The Little Lisper

How would you create a single object?	`new MyObject();`
And, what if another object wanted to create a MyObject? Could it call new on MyObject again?	Yes, of course.
So as long as we have a class, can we always instantiate it one or more times?	Yes. Well, only if it's a public class.
And if not?	Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.
Hmm, interesting. Did you know you could do this?	No, 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?	I suppose it is a class that can't be instantiated because it has a private constructor.
Well, is there ANY object that could use the private constructor?	Hmm, I think the code in MyClass is the only code that could call it. But that doesn't make much sense.

Why not ?	Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken and egg problem: I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use "new MyClass()".

Okay. It was just a thought. What does this mean?	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 CLASS method. You need to use the class name to reference a static method.

Very interesting. What if we put things together. *Now* can I instantiate a MyClass?	Wow, you sure can.

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

So, now can you think of a second way to instantiate an object?	`MyClass.getInstance();`

Can you finish the code so that only ONE instance of MyClass is ever created?	Yes, I think so...

(You'll find the code on the next page.)

Dissecting the classic Singleton Pattern implementation

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

Code Up Close

uniqueInstance holds our ONE instance; remember, it is a static variable.

If uniqueInstance is null, then we haven't created the instance yet...

...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

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.

Patterns Exposed

This week's interview:
Confessions of a Singleton

HeadFirst: Today we are pleased to bring you an interview with a Singleton object. Why don't you begin by telling us a bit about yourself.

Singleton: Well, I'm totally unique; there is just one of me!

HeadFirst: One?

Singleton: Yes, one. I'm based on the Singleton Pattern, which assures that at any one time there is only one instance of me.

HeadFirst: Isn't that sort of a waste? Someone took the time to develop a full-blown class and now all we can get is one object out of it?

Singleton: Not at all! There is power in ONE. Let's say you have an object that contains registry settings. You don't want multiple copies of that object and its values running around – that would lead to chaos. By using an object like me you can assure that every object in your application is making use of the same global resource.

HeadFirst: Tell us more...

Singleton: Oh, I'm good for all kinds of things. Being single sometimes has its advantages you know. I'm often used to manage pools of resources, like connection or thread pools.

HeadFirst: Still, only one of your kind? That sounds lonely.

Singleton: Because there's only one of me, I do keep busy, but it would be nice if more developers knew me – many developers run into bugs because they have multiple copies of objects floating around they're not even aware of.

HeadFirst: So, if we may ask, how do you know there is only one of you? Can't anyone with a new operator create a "new you"?

Singleton: Nope! I'm truly unique.

HeadFirst: Well, do developers swear an oath not to instantiate you more than once?

Singleton: Of course not. The truth be told... well, this is getting kind of personal but... I have no public constructor.

HeadFirst: NO PUBLIC CONSTRUCTOR! Oh, sorry, no public constructor?

Singleton: That's right. My constructor is declared private.

HeadFirst: How does that work? How do you EVER get instantiated?

Singleton: You see, to get a hold of a Singleton object, you don't instantiate one, you just ask for an instance. So my class has a static method called getInstance(). Call that, and I'll show up at once, ready to work. In fact, I may already be helping other objects when you request me.

HeadFirst: Well, Mr. Singleton, there seems to be a lot under your covers to make all this work. Thanks for revealing yourself and we hope to speak with you again soon!

The Chocolate Factory

Everyone knows that all modern chocolate factories have computer controlled chocolate boilers. The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.

Here's the controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler. Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 500 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler!

```java
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.

BRAIN POWER

Choc-O-Holic has done a decent job of ensuring bad things don't happen, don't ya think? Then again, you probably suspect that if two ChocolateBoiler instances get loose, some very bad things can happen.

How might things go wrong if more than one instance of ChocolateBoiler is created in an application?

Sharpen your pencil

Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

```java
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    ┌──────────────────────────────────────────────┐
    │                                              │
    └──────────────────────────────────────────────┘

    ┌────────────┐ ChocolateBoiler() {
    │            │ empty = true;
    └────────────┘ boiled = false;
    }

    ┌──────────────────────────────────────────────┐
    │                                              │
    │                                              │
    │                                              │
    │                                              │
    └──────────────────────────────────────────────┘

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

Singleton Pattern defined

Now that you've got the classic implementation of Singleton in your head, it's time to sit back, enjoy a bar of chocolate, and check out the finer points of the Singleton Pattern.

**Let's start with the concise definition of the pattern:**

> **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

No big surprises there. But, let's break it down a bit more:

- What's really going on here? We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.

- We're also providing a global access point to the instance: whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the Singleton is created in a lazy manner, which is especially important for resource intensive objects.

Okay, let's check out the class diagram:

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.
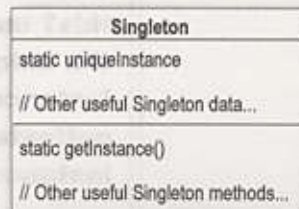
The uniqueInstance class variable holds our one and only instance of Singleton.

Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

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

It looks like the Chocolate Boiler has let us down; despite the fact we improved the code using Classic Singleton, somehow the ChocolateBoiler's fill() method was able to start filling the boiler even though a batch of milk and chocolate was already boiling! That's 500 gallons of spilled milk (and chocolate)! What happened!?

> 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.

Could the addition of threads have caused this? Isn't it the case that once we've set the uniqueInstance variable to the sole instance of ChocolateBoiler, all calls to getInstance() should return the same instance? Right?

BE the JVM

We have two threads, each executing this code. Your job is to play the JVM and determine whether there is a case in which two threads might get ahold of different boiler objects. Hint: you really just need to look at the sequence of operations in the getInstance() method and the value of uniqueInstance to see how they might overlap.
Use the code Magnets to help you study how the code might interleave to create two boiler objects.

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

```
public static ChocolateBoiler
    getInstance() {
```

```
if (uniqueInstance == null) {
```

```
uniqueInstance =
    new ChocolateBoiler();
```

```
}
```

```
return uniqueInstance;
```

```
}
```

Make sure you check your answer on page 188 before turning the page!

Thread One	Thread Two	Value of uniqueInstance

Dealing with multithreading

Our multithreading woes are almost trivially fixed by making getInstance() a synchronized method:

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

I agree this fixes the problem. But synchronization is expensive; is this an issue?

Good point, and it's actually a little worse than you make out: the only time synchronization is relevant is the first time through this method. In other words, once we've set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method. After the first time through, synchronization is totally unneeded overhead!

Can we improve multithreading?

For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads. But, it looks fairly expensive to synchronize the getInstance() method, so what do we do?

Well, we have a few options...

1. Do nothing if the performance of getInstance() isn't critical to your application

That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it. Synchronizing getInstance() is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high traffic part of your code begins using getInstance(), you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one

If your application always creates and uses an instance of the Singleton or the overhead of creation and runtime aspects of the Singleton are not onerous, you may want to create your Singleton eagerly, like this:

```java
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it

Using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.

3. Use "double-checked locking" to reduce the use of synchronization in getInstance()

With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

Let's check out the code:

```java
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.

Double-checked locking doesn't work in Java 1.4 or earlier!

Watch it!

Unfortunately, in Java version 1.4 and earlier, many JVMs contain implementations of the volatile keyword that allow improper synchronization for double-checked locking. If you must use a JVM other than Java 5, consider other methods of implementing your Singleton.

Meanwhile, back at the Chocolate Factory...

While we've been off diagnosing the multithreading problems, the chocolate boiler has been cleaned up and is ready to go. But first, we have to fix the multithreading problems. We have a few solutions at hand, each with different tradeoffs, so which solution are we going to employ?

✏️ Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

_____

_____

Use eager instantiation:

_____

_____

Double-checked locking:

_____

_____

Congratulations!

At this point, the Chocolate Factory is a happy customer and Choc-O-Holic was glad to have some expertise applied to their boiler code. No matter which multithreading solution you applied, the boiler should be in good shape with no more mishaps. Congratulations. You've not only managed to escape 500lbs of hot chocolate in this chapter, but you've been through all the potential problems of the Singleton.

Dumb Questions
there are no

Q: For 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 your class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy, especially if multiple classes are involved. Often this scenario can result in subtle, hard to find bugs involving order of initialization. Unless there is a compelling need to implement your "singleton" this way, it 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.

Relax

Rumors of Singletons being eaten by the garbage collectors are greatly exaggerated

Prior to Java 1.2, a bug in the garbage collector allowed Singletons to be prematurely collected if there was no global reference to them. In other words, you could create a Singleton and if the only reference to the Singleton was in the Singleton itself, it would be collected and destroyed by the garbage collector. This leads to confusing bugs because after the Singleton is "collected," the next call to getInstance() produced a shiny new Singleton. In many applications, this can cause confusing behavior as state is mysteriously reset to initial values or things like network connections are reset.

Since Java 1.2 this bug has been fixed and a global reference is no longer required. If you are, for some reason, still using a pre-Java 1.2 JVM, then be aware of this issue, otherwise, you can sleep well knowing your Singletons won't be prematurely collected.

Q: I've always been taught that a class should do one thing and one thing only. For a class to do two things is considered bad OO design. Isn't a Singleton violating this?

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 your 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 your constructor so that it's public or protected. But then, it's not *really* a Singleton anymore, because other classes can instantiate it.

If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing registry 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. Like 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 any existing class. Last, if you are using a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.

Q: I still don't totally understand why global variables are worse than a Singleton.

A: In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. We've already mentioned one: the issue of lazy versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. A global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.

Tools for your Design Toolbox

You've now added another pattern to your toolbox. Singleton gives you another method of creating objects – in this case, unique objects.

OO Basics

traction
apsulation
ymorphism
eritance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.

OO Patterns

Singleton – Ensure a class only has one instance and provide a global point of access to it

BULLET POINTS

- The Singleton Pattern ensures you have at most one instance of a class in your application.

- The Singleton Pattern also provides a global access point to that instance.

- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.

- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).

- Beware of the double-checked locking implementation; it is not thread-safe in versions before Java 2, version 5.

- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.

- If you are using a JVM earlier than 1.2, you'll need to create a registry of Singletons to defeat the garbage collector.

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.
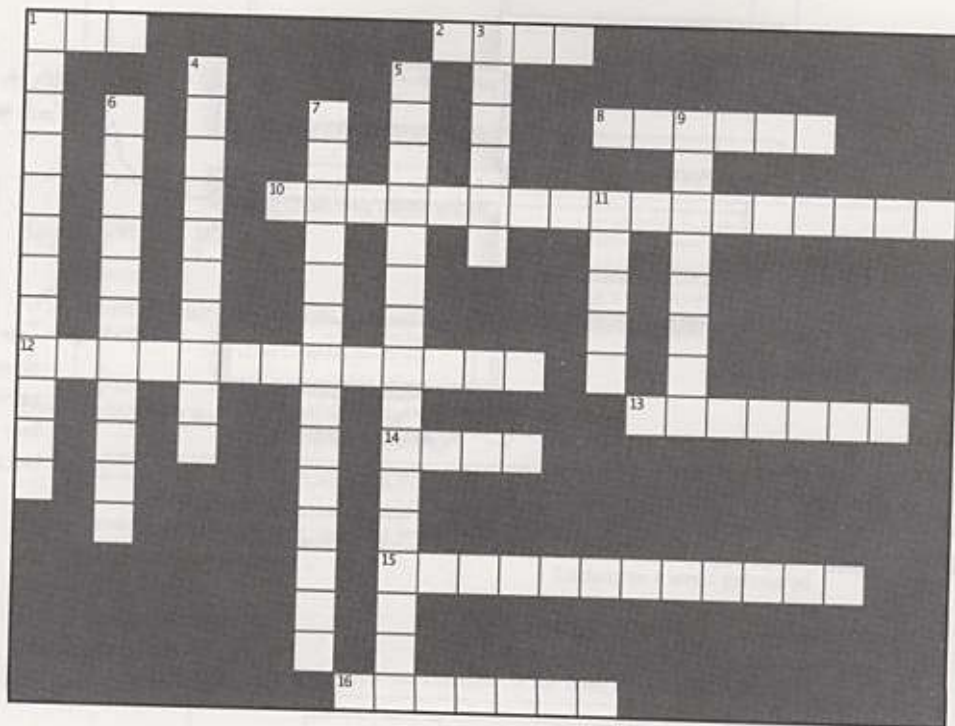
Sit back, open that case of chocolate that you were sent for solving the multithreading problem, and have some downtime working on this little crossword puzzle; all of the solution words are from this chapter.
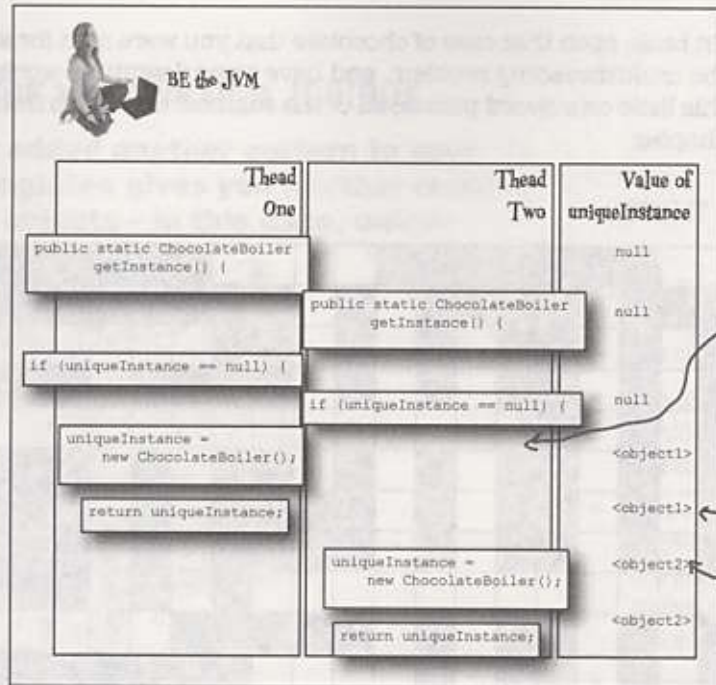
Across

1. It was "one of a kind"
2. Added to chocolate in the boiler
8. An incorrect implementation caused this to overflow
10. Singleton provides a single instance and (three words)
12. Flawed multithreading approach if not using Java 1.5
13. Chocolate capital of the US
14. One advantage over global variables: _____ creation
15. Company that produces boilers
16. To totally defeat the new constructor, we have to declare the constructor _____

Down

1. Multiple _____ can cause problems
3. A Singleton is a class that manages an instance of _____
4. If you don't need to worry about lazy instantiation, you can create your instance _____
5. Prior to 1.2, this can eat your Singletons (two words)
6. The Singleton was embarassed it had no public _____
7. The classic implementation doesn't handle this
9. Singleton ensures only one of these exist
11. The Singleton Pattern has one

Exercise
solutions

BE the JVM

	Thead One	Thead Two	Value of uniqueInstance
	`public static ChocolateBoiler getInstance() {`		null
		`public static ChocolateBoiler getInstance() {`	null
	`if (uniqueInstance == null) {`		null
		`if (uniqueInstance == null) {`	null
	`uniqueInstance = new ChocolateBoiler();`		<object1>
	`return uniqueInstance;`		<object1>
		`uniqueInstance = new ChocolateBoiler();`	<object2>
		`return uniqueInstance;`	<object2>

Uh oh, this doesn't look good!

Two different objects are returned! We have two ChocolateBoiler instances!!!

Sharpen your pencil Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    // rest of ChocolateBoiler code...
}
```

Exercise solutions

Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

A straightforward technique that is guaranteed to work. We don't seem to have any

performance concerns with the chocolate boiler, so this would be a good choice.

Use eager instantiation:

We are always going to instantiate the chocolate boiler in our code, so statically inializing the

instance would cause no concerns. This solution would work as well as the synchronized method,

although perhaps be less obvious to a developer familar with the standard pattern.

Double checked locking:

Given we have no performance concerns, double-checked locking seems like overkill. In addition, we'd

have to ensure that we are running at least Java 5.

Exercise solutions

The crossword grid reads:

- 1 Across: **CAR**
- 2 Across: **MILK**
- 8 Across: **BOILER**
- 10 Across: **GLOBAL ACCESS POINT**
- 12 Across: **DOUBLE CHECKED**
- 13 Across: **HERSHEY**
- 14 Across: **LAZY**
- 15 Across: **CHOC-O-HOLIC**
- 16 Across: **PRIVATE**

Down answers include:
- **CLASSLOADERS**
- **CONSTRUCTOR**
- **STATICALLY**
- **MUTITIT...**
- **MARRAGE**
- **ITSSEF...**
- **INSTANCE**
- **MILREADING**
- **PALLETTO...**

7 the Adapter and Facade Patterns

Being Adaptive

> Do you think the readers are really getting the impression we're watching a horse race rather than sitting in a photo studio?

> That's the beauty of our profession, we can make things look like something they're not!

> You mean it's not supposed to be a football match?

> Wrapped in this coat, I'm a different man!

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

Adapters all around us

**You'll have no trouble understanding what an OO adapter is
because the real world is full of them. How's this for an example:
Have you ever needed to use a US-made laptop in a European
country? Then you've probably needed an AC power adapter...**

European Wall Outlet

AC Power Adapter

Standard AC Plug

The US laptop expects
another interface.

The European wall outlet exposes
one interface for getting power.

The adapter converts one
interface into another.
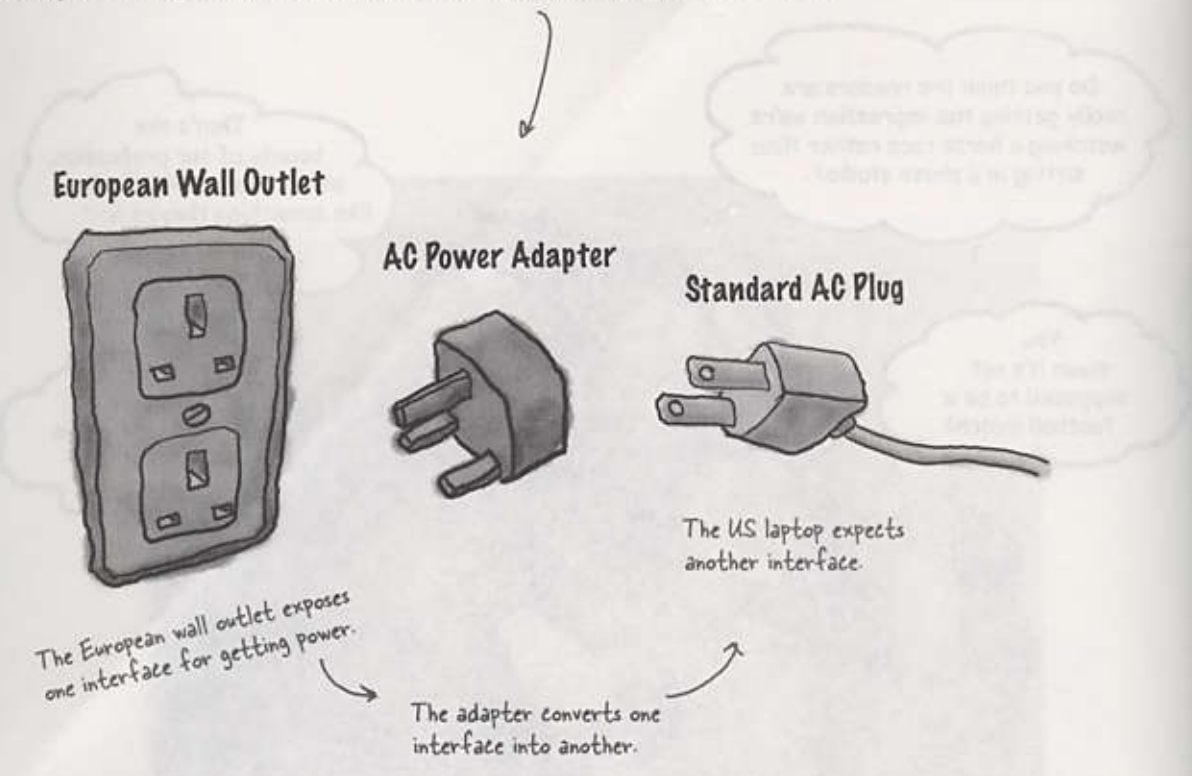
You know what the adapter does: it sits in between the plug of your laptop and the
European AC outlet; its job is to adapt the European outlet so that you can plug your
laptop into it and receive power. Or look at it this way: the adapter changes the interface
of the outlet into one that your laptop expects.

*How many other real world
adapters can you think of?*

Some AC adapters are simple – they only change the shape of the outlet so that it matches
your plug, and they pass the AC current straight through – but other adapters are more
complex internally and may need to step the power up or down to match your devices'
needs.

Okay, that's the real world, what about object oriented adapters? Well, our OO adapters
play the same role as their real world counterparts: they take an interface and adapt it to
one that a client is expecting.
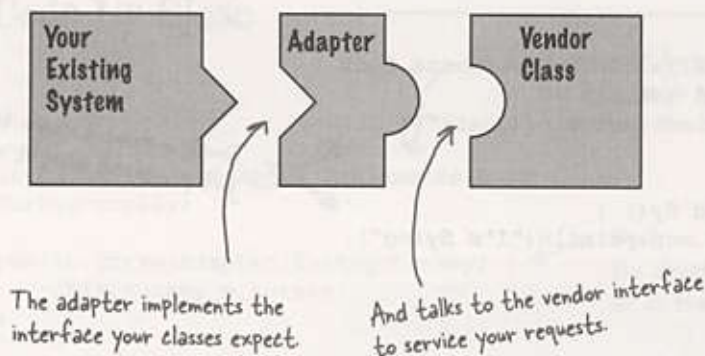
Object oriented adapters

Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:

> **Your Existing System** > **Vendor Class**

Their interface doesn't match the one you've written your code against. This isn't going to work!
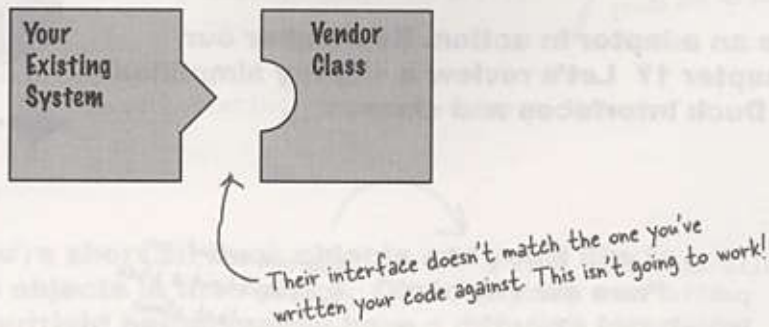
Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.

> **Your Existing System** > **Adapter** > **Vendor Class**

The adapter implements the interface your classes expect

And talks to the vendor interface to service your requests.

The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.

> **Your Existing System** **Adapter** > **Vendor Class**

No code changes.　　New code.　　No code changes.
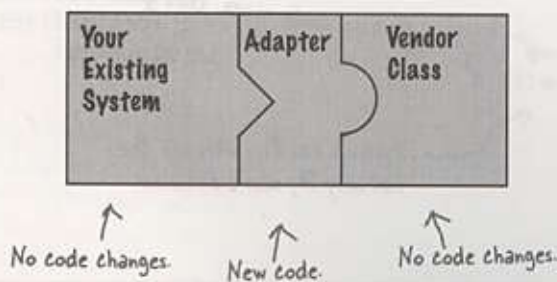
Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

It's time to see an adapter in action. Remember our ducks from Chapter 1? Let's review a slightly simplified version of the Duck interfaces and classes:

```
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

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts — they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test drive the adapter

Now we just need some code to test drive our adapter:

```java
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run

```
File Edit Window Help Don'tForgetToDuck
%java 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
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```
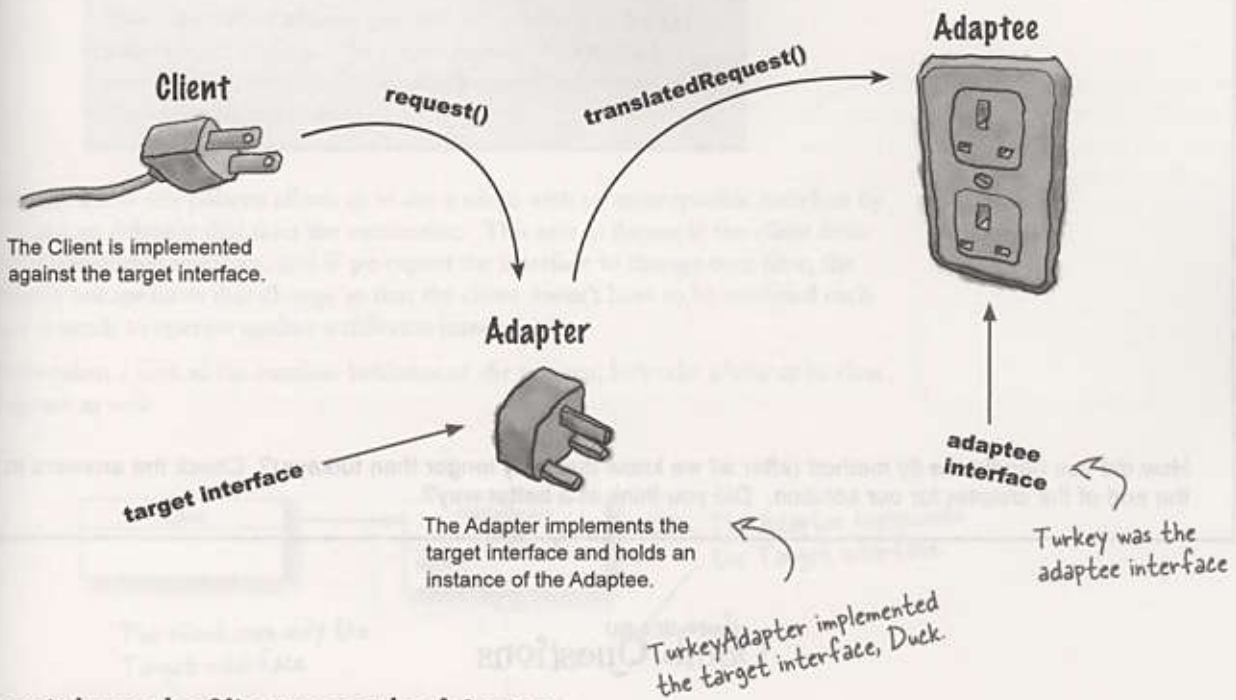
The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained

Now that we have an idea of what an Adapter is, let's step back and look at all the pieces again.

Client

request()

translatedRequest()

Adaptee

The Client is implemented against the target interface.

Adapter

target interface

The Adapter implements the target interface and holds an instance of the Adaptee.

adaptee interface

Turkey was the adaptee interface

TurkeyAdapter implemented the target interface, Duck

Here's how the Client uses the Adapter

❶ The client makes a request to the adapter by calling a method on it using the target interface.

❷ The adapter translates that request into one or more calls on the adaptee using the adaptee interface.

❸ The client receives the results of the call and never knows there is an adapter doing the translation.

Note that the Client and Adaptee are decoupled — neither knows about the other.

Sharpen your pencil

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:

How did you handle the fly method (after all we know ducks fly longer than turkeys)? Check the answers at the end of the chapter for our solution. Did you think of a better way?

there are no Dumb Questions

Q: How much "adapting" does an adapter need to do? It seems like if I need to implement a large target interface, I could have a LOT of work on my hands.

A: You certainly could. The job of implementing an adapter really is proportional to the size of the interface you need to support as your target interface. Think about your options, however. You could rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes. Or, you can cleanly provide one class that encapsulates all the changes in one class.

Q: Does an adapter always wrap one and only one class?

A: The Adapter Pattern's role is to convert one interface into another. While most examples of the adapter pattern show an adapter wrapping one adaptee, we both know the world is often a bit more messy. So, you may well have situations where an adapter holds two or more adaptees that are needed to implement the target interface.

This relates to another pattern called the Facade Pattern; people often confuse the two. Remind us to revisit this point when we talk about facades later in this chapter.

Q: What if I have old and new parts of my system, 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 I be better off just writing my older code and forgetting the adapter?

A: Not necessarily. One thing you can do is create a Two Way Adapter that supports both interfaces. To create a Two Way Adapter, just implement both interfaces involved, so the adapter can act as an old interface or a new interface.
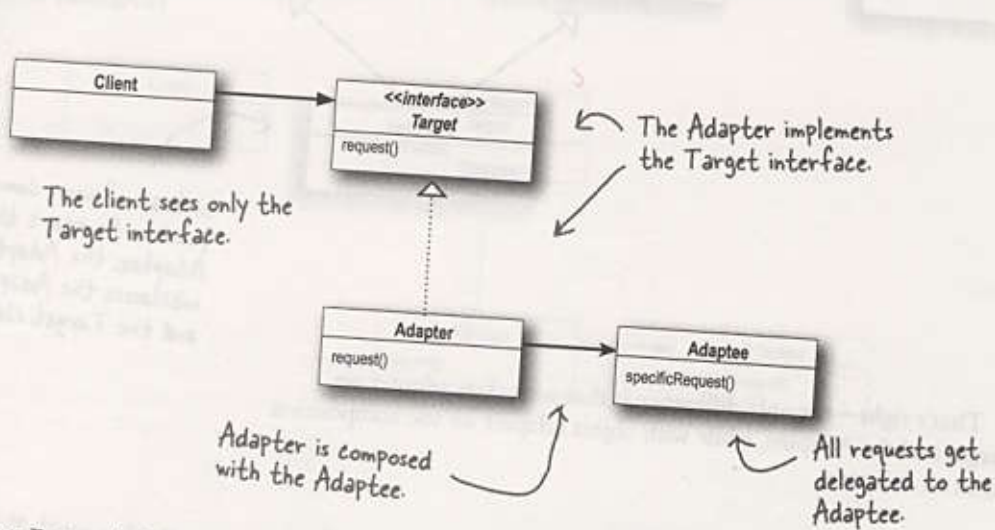
Adapter Pattern defined

Enough ducks, turkeys and AC power adapters; let's get real and look at the official definition of the Adapter Pattern:

> **The Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well:

The client sees only the Target interface.

The Adapter implements the Target interface.

Adapter is composed with the Adaptee.

All requests get delegated to the Adaptee.

The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Also check out how the pattern binds the client to an interface, not an implementation; we could use several adapters, each converting a different backend set of classes. Or, we could add new implementations after the fact, as long as they adhere to the Target interface.

Object and class adapters

Now despite having defined the pattern, we haven't told you the whole story yet. There are actually *two* kinds of adapters: *object* adapters and *class* adapters. This chapter has covered object adapters and the class diagram on the previous page is a diagram of an object adapter.

So what's a *class* adapter and why haven't we told you about it? Because you need multiple inheritance to implement it, which isn't possible in Java. But, that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language! Let's look at the class diagram for multiple inheritance.

Client		Target		Adaptee
		request()		specificRequest()

Adapter
request()

Instead of using com-position to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.
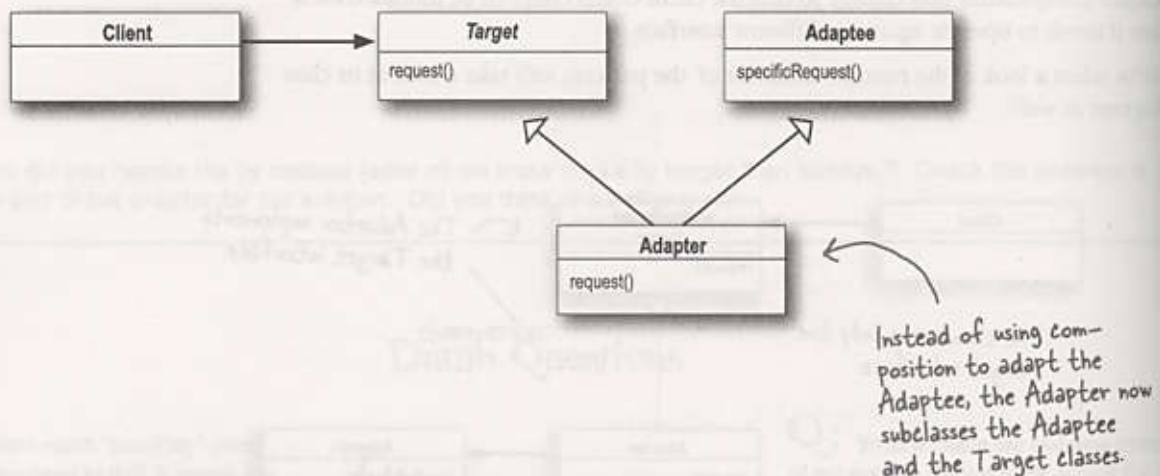
Look familiar? That's right – the only difference is that with class adapter we subclass the Target and the Adaptee, while with object adapter we use composition to pass requests to an Adaptee.

✹ BRAIN POWER

Object adapters and class adapters use two different means of adapting the adaptee (composition versus inheritance). How do these implementation differences affect the flexibility of the adapter?

Duck Magnets

Your job is to take the duck and turkey magnets and drag them over the part of the diagram that describes the role played by that bird, in our earlier example. (Try not to flip back through the pages). Then add your own annotations to describe how it works.
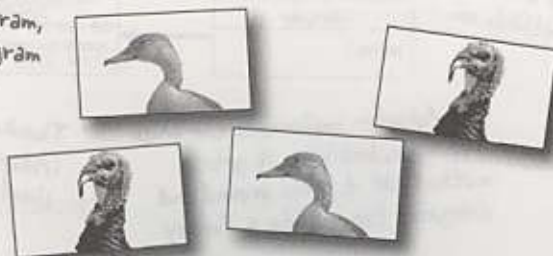
Class Adapter
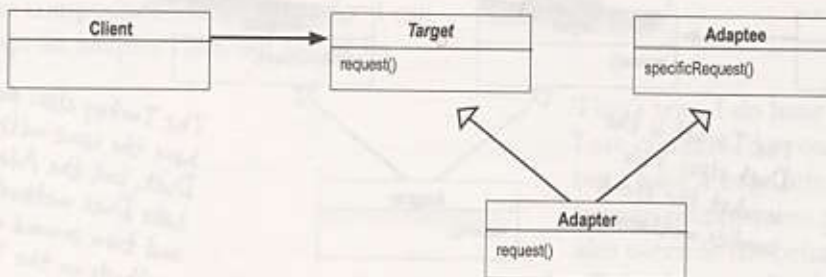
Object Adapter
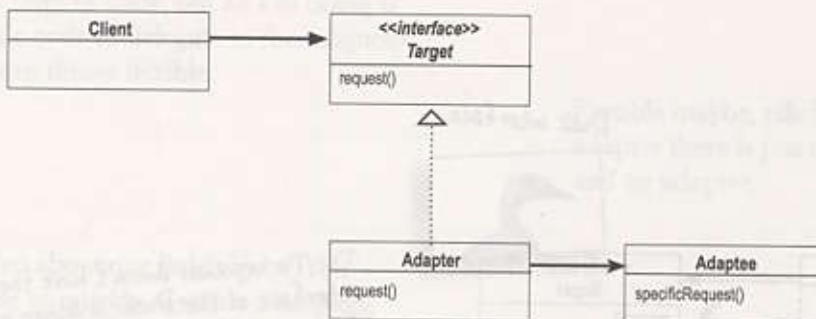
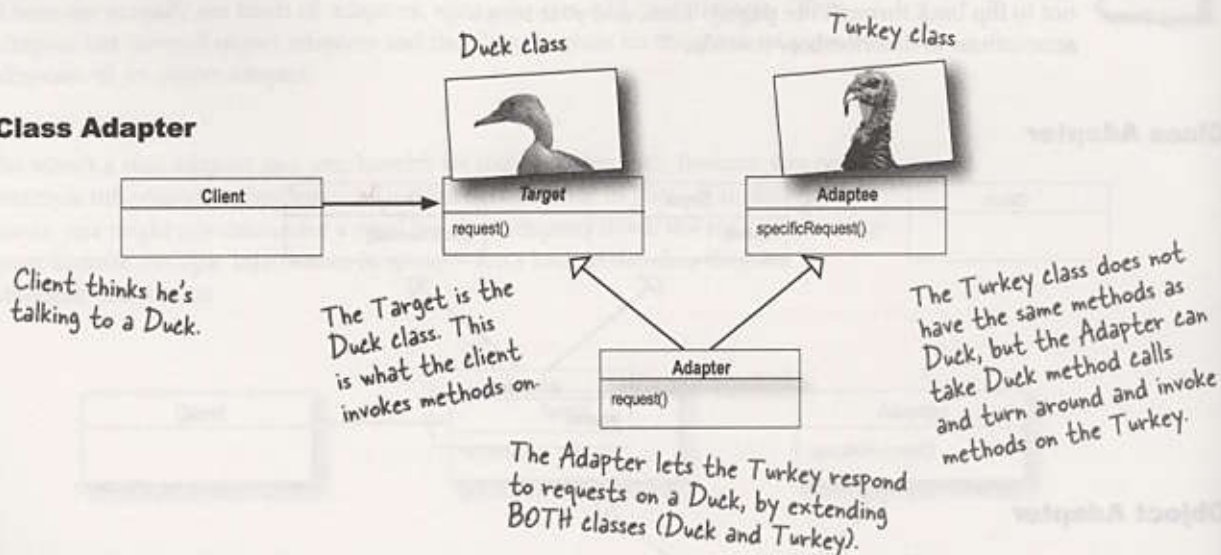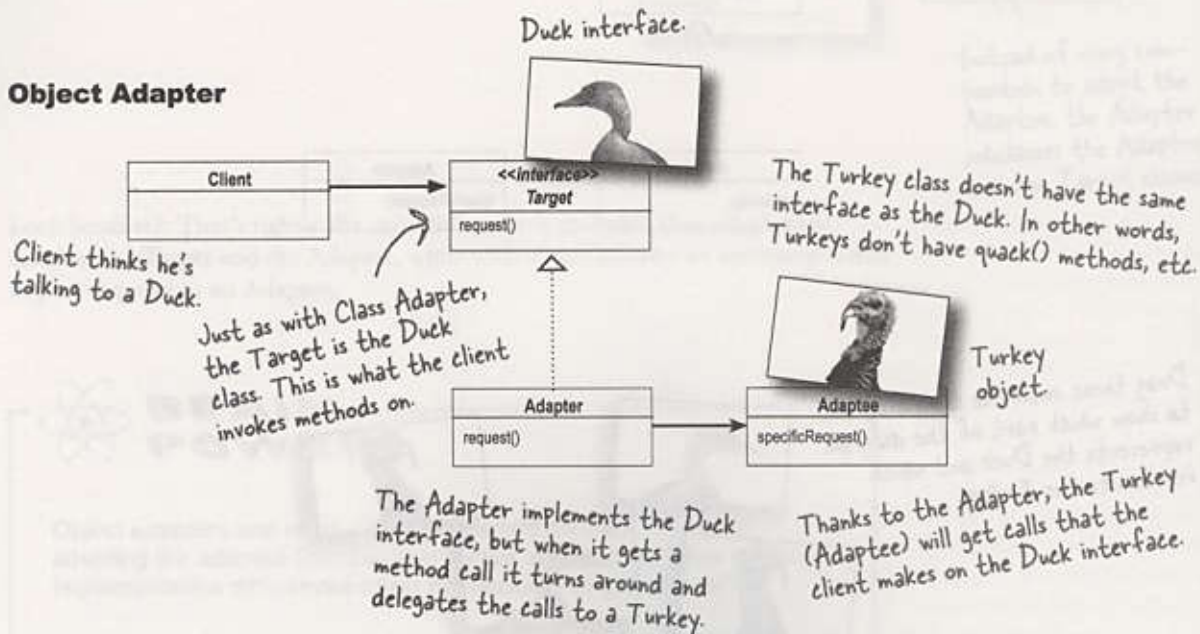Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.

Duck Magnets Answer

Note: the class adapter uses multiple inheritance, so you can't do it in Java...

Class Adapter

Duck class

Turkey class

Client

Target
request()

Adaptee
specificRequest()

Adapter
request()

Client thinks he's talking to a Duck.

The Target is the Duck class. This is what the client invokes methods on.

The Adapter lets the Turkey respond to requests on a Duck, by extending BOTH classes (Duck and Turkey).

The Turkey class does not have the same methods as Duck, but the Adapter can take Duck method calls and turn around and invoke methods on the Turkey.

Object Adapter

Duck interface.

Client

<<interface>> Target
request()

Adapter
request()

Adaptee
specificRequest()

Client thinks he's talking to a Duck.

Just as with Class Adapter, the Target is the Duck class. This is what the client invokes methods on.

The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to a Turkey.

The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.

Turkey object

Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

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 *and* 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...

Real world adapters

Let's take a look at the use of a simple Adapter in the real world (something more serious than Ducks at least)...

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.

Enumeration has a simple interface.

```
<<interface>>
Enumeration

hasMoreElements()
nextElement()
```

Tells you if there are any more elements in the collection.

Gives you the next element in the collection.
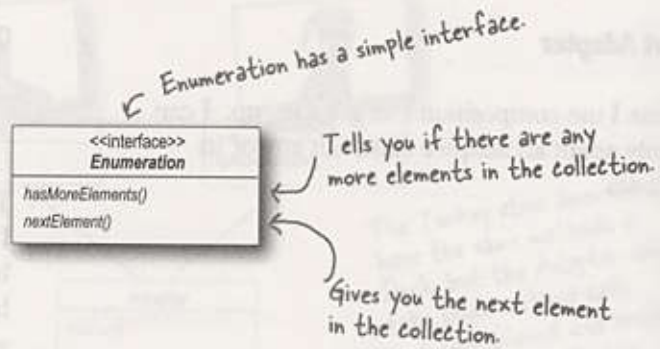
New world Iterators

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.

```
<<interface>
Iterator

hasNext()
next()
remove()
```

Analogous to hasMoreElements() in the Enumeration interface. This method just tells you if you've looked at all the items in the collection.

Gives you the next element in the collection.

Removes an item from the collection.
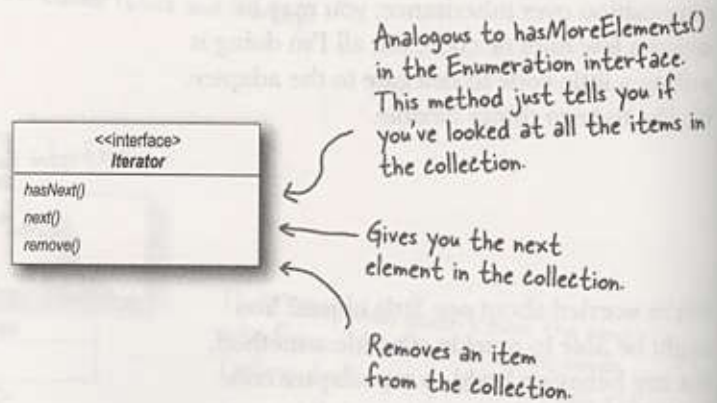
And today...

We are often faced with legacy code that exposes the Enumerator interface, yet we'd like for our new code to only use 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.

These two methods look easy, they map straight to hasNext() and next() in Iterator.

Target interface

<<interface>> Iterator
hasNext() next() remove()

<<interface>> Enumeration
hasMoreElements() nextElement()

Adaptee interface

But what about this method remove() in Iterator? There's nothing like that in Enumeration.
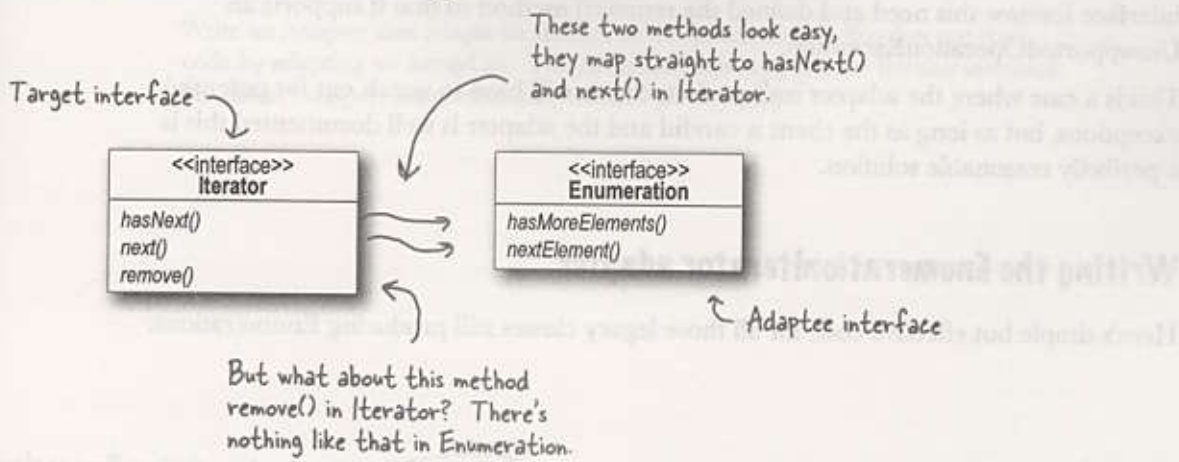
Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and that is composed with an adaptee. The hasNext() and next() methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about remove()? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram:

Your new code still gets to use Iterators, even if there's really an Enumeration underneath.

<<interface>> Iterator
hasNext() next() remove()

We're making the Enumerations in your old code look like Iterators for your new code.

A class implementing the Enumeration interface is the adaptee.

EnumerationIterator is the adapter.

EnumerationIterator
hasNext() next() remove()

<<interface>> Enumeration
hasMoreElements() nextElement()

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
UnsupportedOperationException.

This is a case where the adapter isn't perfect; clients will have to watch out for potential
exceptions, but as long as the client is careful and the adapter is well documented this is
a perfectly reasonable solution.

Writing the EnumerationIterator adapter

Here's simple but effective code for all those legacy classes still producing Enumerations:

```
public class EnumerationIterator implements Iterator
{
    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 Enumerations'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.

Exercise

While Java has gone 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).

BRAIN POWER

Some AC adapters do more than just change the interface – they add other features like surge protection, indicator lights and other bells and whistles.

If you were going to implement these kinds of features, what pattern would you use?

Fireside Chats
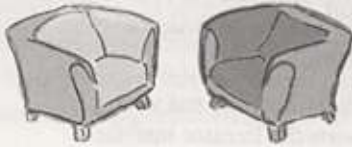
Tonight's talk: **The Decorator Pattern and the Adapter Pattern discuss their differences.**

Decorator

Adapter

I'm important. My job is all about *responsibility* — you know that when a Decorator is involved there's going to be some new responsibilities or behaviors added to your design.

You 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.

That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.

Try being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying: "an uncoupled client is a happy client."

Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.

Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job.

Decorator

Adapter

But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *any* code, they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.

Well us decorators do that as well, only we allow *new behavior* to be added to classes without altering existing code. I still say that adapters are just fancy decorators – I mean, just like us, you wrap an object.

No, no, no, not at all. We *always* convert the interface of what we wrap, you *never* do. I'd say a decorator is like an adapter; it is just that you don't change the interface!

Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap, we aren't a *simple pass through*.

Hey, who are you calling a simple pass through? Come on down and we'll see how long *you* last converting a few interfaces!

Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *miles* away in our *intent*.

Oh yeah, I'm with you there.

And now for something different...

There's another pattern in this chapter.

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.

We're going to look at a pattern now that alters an interface, but for a different reason: to simplify the interface. It's aptly named the Facade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well-lit facade.

WHO DOES WHAT?

Match each pattern with its intent:

Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

Home Sweet Home Theater

Before we dive into the details of the Facade Pattern, let's take a look at a growing national obsession: building your own home theater.

You've done your research and you've assembled a killer system complete with a DVD player, a projection video system, an automated screen, surround sound and even a popcorn popper.

Check out all the components you've put together:

Amplifier
tuner
dvdPlayer
cdPlayer
on()
off()
setCd()
setDvd()
setStereoSound()
setSurroundSoud()
setTuner()
setVolume()

Tuner
amplifier
on()
off()
setAm()
setFm()
setFrequency()

DvdPlayer
amplifier
on()
off()
eject()
pause()
play()
play()
setSurroundAudio()
setTwoChannelAudio()
stop()
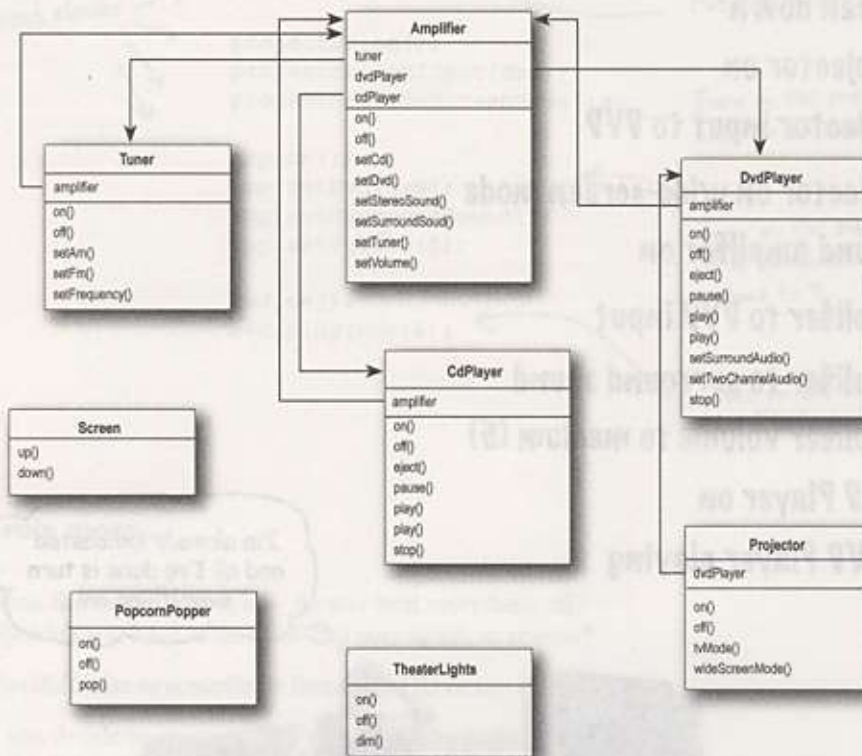
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

CdPlayer
amplifier
on()
off()
eject()
pause()
play()
play()
stop()

Screen
up()
down()

Projector
dvdPlayer
on()
off()
tvMode()
wideScreenMode()

PopcornPopper
on()
off()
pop()

TheaterLights
on()
off()
dim()

You've spent weeks running wire, mounting the projector, making all the connections and fine tuning. Now it's time to put it all in motion and enjoy a movie...

Watching a movie (the hard way)

Pick out a DVD, relax, and get ready for movie magic. Oh, there's just one thing – to watch the movie, you need to perform a few tasks:

❶ **Turn on the popcorn popper**

❷ **Start the popper popping**

❸ **Dim the lights**

❹ **Put the screen down**

❺ **Turn the projector on**

❻ **Set the projector input to DVD**

❼ **Put the projector on wide-screen mode**

❽ **Turn the sound amplifier on**

❾ **Set the amplifier to DVD input**

❿ **Set the amplifier to surround sound**

⓫ **Set the amplifier volume to medium (5)**

⓬ **Turn the DVD Player on**

⓭ **Start the DVD Player playing**

> I'm already exhausted and all I've done is turn everything on!

Let's check out those same tasks in terms of the classes and the method calls needed to perform them:

```
popper.on();
popper.pop();
```
Turn on the popcorn popper and start popping...

```
lights.dim(10);
```
Dim the lights to 10%...

```
screen.down();
```
Put the screen down...

Six different classes involved!

```
projector.on();
projector.setInput(dvd);
projector.wideScreenMode()
```
Turn on the projector and put it in wide screen mode for the movie...

```
amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);
```
Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...

```
dvd.on();
dvd.play(movie);
```
Turn on the DVD player... and FINALLY, play the movie!
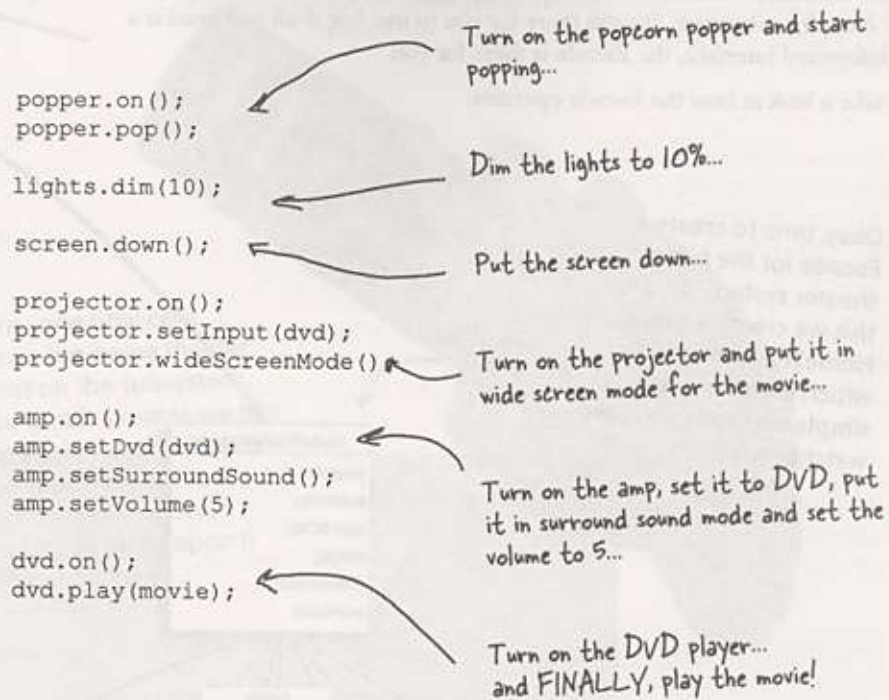
But there's more...

- When the movie is over, how do you turn everything off? Wouldn't you have to do all of this over again, in reverse?

- Wouldn't it be as complex to listen to 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...

Lights, Camera, Facade!

A Facade is just what you need: with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.

Let's take a look at how the Facade operates:

1 Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

2 The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.

The Facade

HomeTheaterFacade

watchMovie()
endMovie()
listenToCd()
endCd()
listenToRadio()
endRadio()

Amplifier

Tuner

DvdPlayer

CdPlayer

Screen

Projector

PopcornPopper

TheaterLights

play()

on()

The subsystem the Facade is simplifying.
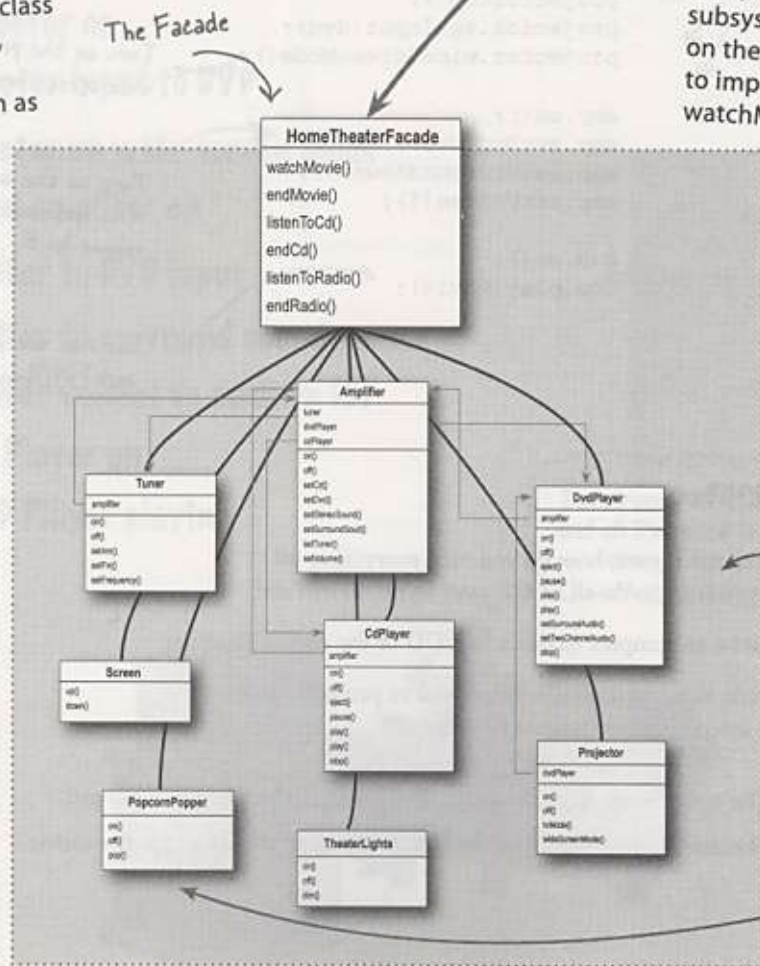
watchMovie()

A client of the ~~subsystem~~ facade
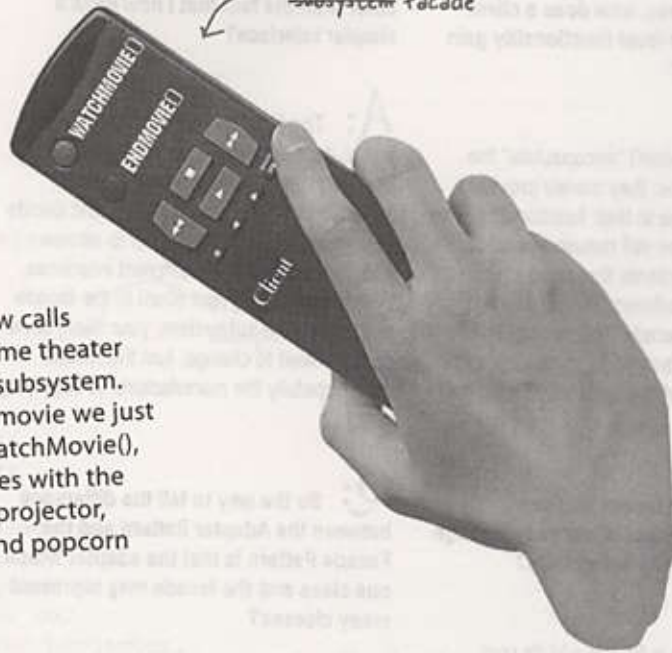
3 Your client code now calls methods on the home theater Facade, not on the subsystem. So now to watch a movie we just call one method, watchMovie(), and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.

I've got to have my low-level access!

Formerly president of the Rushmore High School A/V Science Club.

4 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: Facades don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces. This is a nice property of the Facade Pattern: it provides a simplified interface while still exposing the full functionality of the system to those who may need it.

Q: Does the facade add any functionality or does it just pass through each request to the subsystem?

A: A facade is free to add its own "smarts" in addition to making use of the subsystem. For instance, while our home theater facade doesn't implement any new behavior, it is smart enough to know that the popcorn popper has to be turned on before it can pop (as well as the details of how to turn on and stage a movie showing).

Q: Does each subsystem have only one facade?

A: Not necessarily. The pattern certainly allows for any number of facades to be created for a given subsystem.

Q: What is the benefit of the facade other than the fact that I now have a simpler interface?

A: The Facade Pattern also allows you to decouple your client implementation from any one subsystem. Let's say for instance that you get a big raise and decide to upgrade your home theater to all new components that have different interfaces. Well, if you coded your client to the facade rather than the subsystem, your client code doesn't need to change, just the facade (and hopefully the manufacturer is supplying that!).

Q: So the way to tell the difference between the Adapter Pattern and the Facade Pattern is that the adapter wraps one class and the facade may represent many classes?

A: No! Remember, the Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface.

The difference between the two is not in terms of how many classes they "wrap," it is in their **intent**. The intent of the Adapter Pattern is to **alter** an interface so that it matches one a client is expecting. The intent of the Facade Pattern is to provide a **simplified** interface to a subsystem.

> A facade not only simplifies an interface, it decouples a client from a subsystem of components.
>
> Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.

Constructing your home theater facade

Let's step through the construction of the HomeTheaterFacade: The
first step is to use composition so that the facade has access to all the
components of the subsystem:

```java
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:

```java
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);
}
```

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.

```java
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();
}
```

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

BRAIN POWER

Think about the facades you've encountered in the Java API.
Where would you like to have a few new ones?

Time to watch a movie (the easy way)

It's SHOWTIME!

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
            new HomeTheaterFacade(amp, tuner, 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 SnakesWhy'dItHaveToBeSnakes?

```
%java HomeTheaterTestDrive

Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
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 subsytem. Facade defines a higher-level interface that makes the subsystem easier to use.

There isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade it to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram:

Happy client whose job just became easier because of the facade.

More complex subsystem.

Unified interface that is easier to use.

subsystem classes
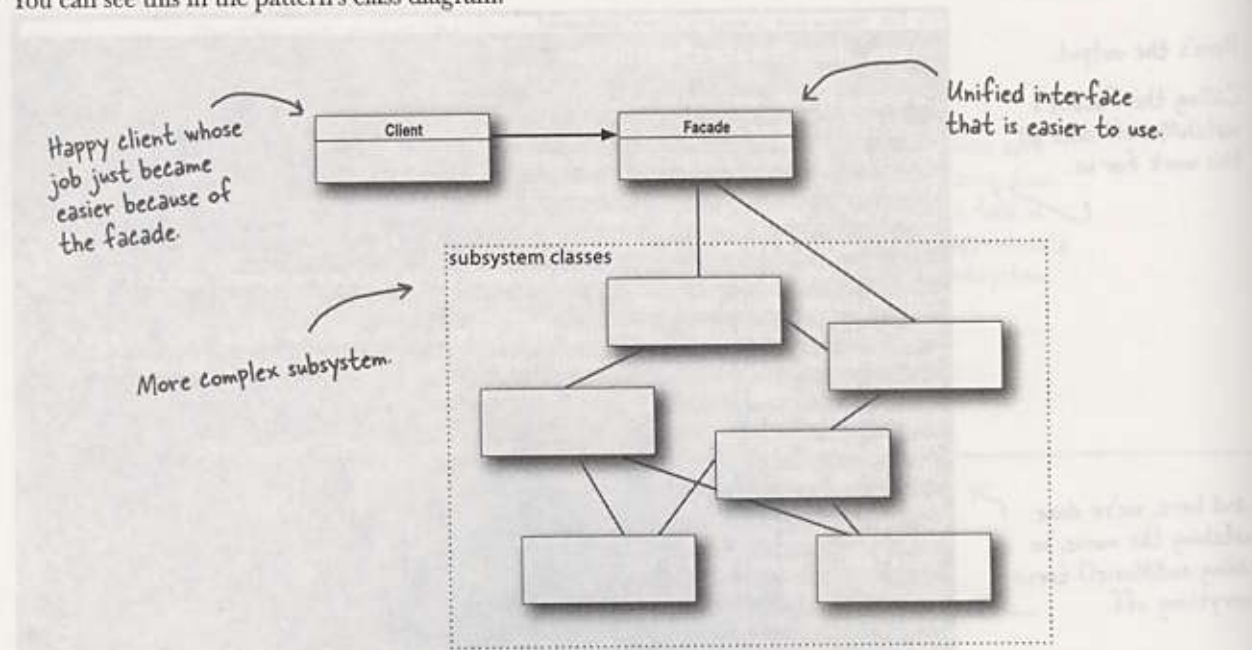
That's it; you've got another pattern under your belt! Now, it's time for that new OO principle. Watch out, this one can challenge some assumptions!

The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close "friends." The principle is usually stated as:

> **Design Principle**
>
> Principle of Least Knowledge - talk only to your immediate friends.

But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

BRAIN POWER

How many classes is this code coupled to?

```
public float getTemp() {
    return station.getThermometer().getTemperature();
}
```

How NOT to Win Friends and Influence Objects

Okay, but how do you keep from doing this? The principle provides some guidelines: take any object; 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:

```java
public class Car {
    Engine engine;
    // other instance variables

    public Car() {
        // initialize engine, etc.
    }

    public void start(Key key) {
        Doors doors = new Doors();

        boolean authorized = key.turns();

        if (authorized) {
            engine.start();
            updateDashboardDisplay();
            doors.lock();
        }
    }

    public void updateDashboardDisplay() {
        // update display
    }
}
```

Here's a component of this class. We can call its methods. → Engine engine;

Here we're creating a new object, its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object → key.turns();

You can call a local method within the object → updateDashboardDisplay();

You can call a method on an object you create or instantiate.

there are no Dumb Questions

Q: There is another principle called the Law of Demeter; how are they related?

A: The two are one and the same and you'll encounter these terms being intermixed. We prefer to use the Principle of Least Knowledge for a couple of reasons: (1) the name is more intuitive and (2) the use of the word "Law" implies we always have to apply this principle. In fact, no principle is a law, all principles should be used when and where they are helpful. All design involves tradeoffs (abstractions versus speed, space versus time, and so on) and while principles provide guidance, all factors should be taken into account before applying them.

Q: Are there any disadvantages to applying the Principle of Least Knowledge?

A: Yes; while the principle reduces the dependencies between objects and studies have shown this reduces software maintenance, it is also the case that applying this principle results in more "wrapper" classes being written to handle method calls to other components. This can result in increased complexity and development time as well as decreased runtime performance.

Sharpen your pencil

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
```

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

HARD HAT AREA. WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN POWER

Can you think of a common use of Java that violates the Principle of Least Knowledge?

Should you care?

Answer: How about System.out.println()?

The Facade and the Principle of Least Knowledge

Client

This client only has one friend; the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!

The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

HomeTheaterFacade

watchMovie()
endMovie()
listenToCd()
endCd()
listenToRadio()
endRadio()

We can upgrade the home theater components without affecting the client.

Amplifier

tuner
dvdPlayer
cdPlayer

on()
off()
setDvd()
setCd()
setStereoSound()
setSurroundSound()
setTuner()
setVolume()

Tuner

amplifier

on()
off()
setAm()
setFm()
setFrequency()

DvdPlayer

amplifier

on()
off()
eject()
pause()
play()
play()
setSurroundAudio()
setTwoChannelAudio()
stop()

CdPlayer

amplifier

on()
off()
eject()
pause()
play()
play()
stop()

Screen

up()
down()

PopcornPopper

on()
off()
pop()

TheaterLights

on()
off()
dim()

Projector

dvdPlayer

on()
off()
tvMode()
wideScreenMode()

We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.
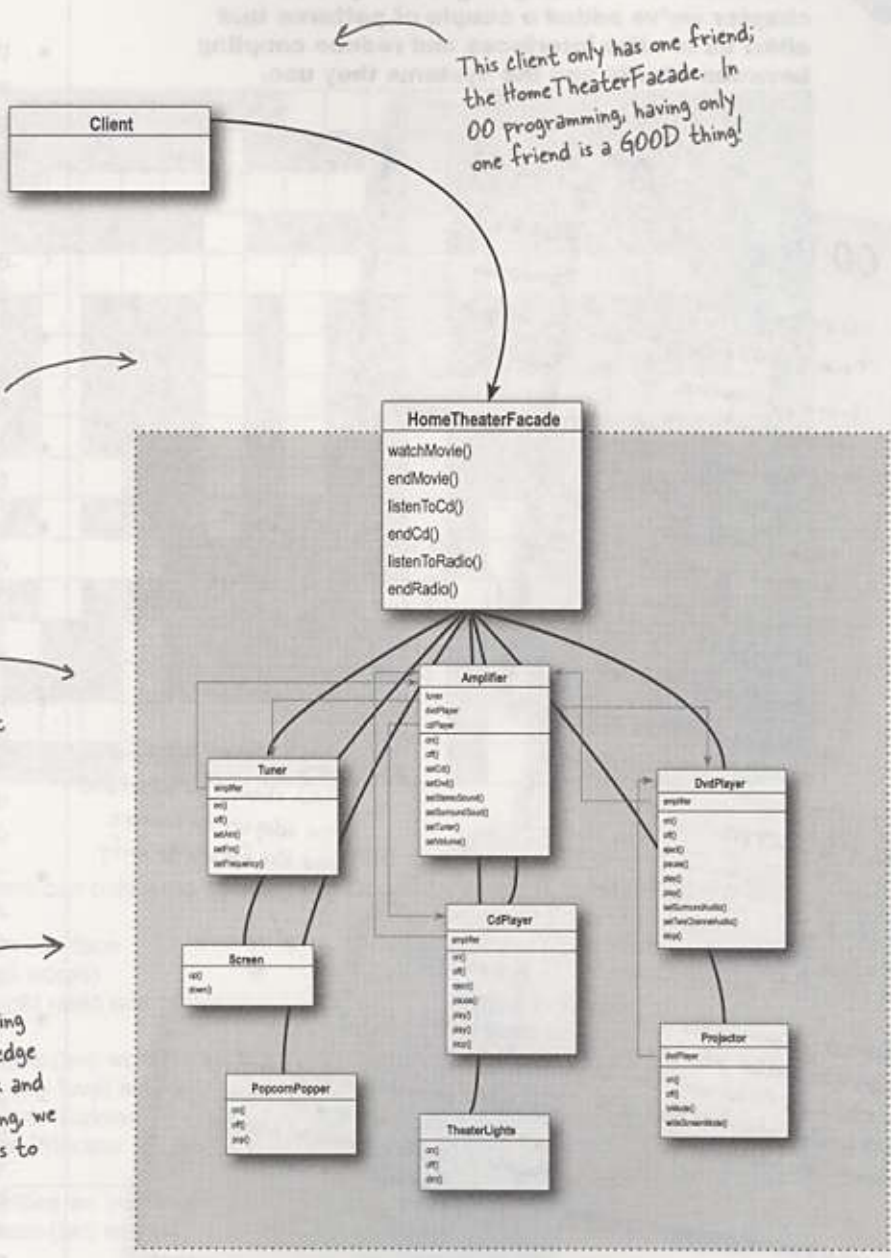
Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

OO Principles

Encapsulate what varies
Favor composition over inheritance
Program to interfaces, not implementations
Strive for loosely coupled designs between objects that interact
Classes should be open for extension but closed for modification
Depend on abstractions. Do not depend on concretions
Only talk to your friends

We have a new technique for maintaining a low level of coupling in our designs. (remember, talk only to your friends)...

OO Patterns

Adapter - Converts the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

...and TWO new patterns. Each changes an interface, the adapter to convert and the facade to unify and simplify

...es a request ...ing you different

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.

BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.

- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.

- An adapter changes an interface into one a client expects.

- A facade decouples a client from a complex subsystem.

- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.

- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.

- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.

- You can implement more than one facade for a subsystem.

- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

Yes, it's another crossword. All of the solution words are from this chapter.
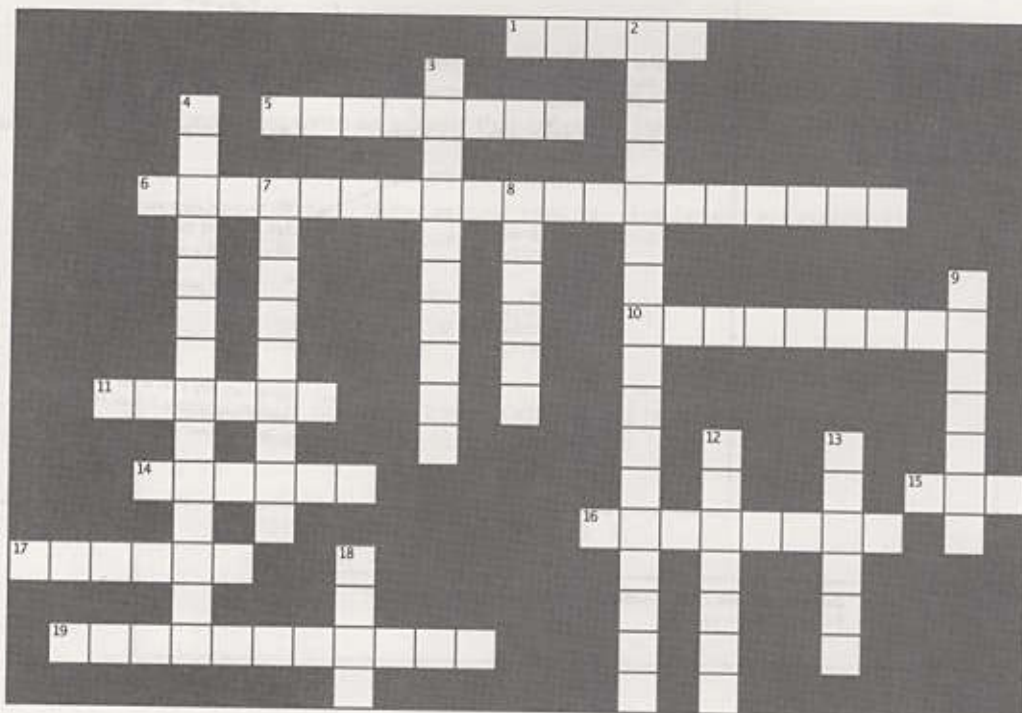
Across

1. True or false, Adapters can only wrap one object
5. An Adapter _____ an interface
6. Movie we watched (5 words)
10. If in Europe you might need one of these (two words)
11. Adapter with two roles (two words)
14. Facade still _____ low level access
15. Ducks do it better than Turkeys
16. Disadvantage of the Principle of Least Knowledge: too many _____
17. A _____ simplifies an interface
19. New American dream (two words)

Down

2. Decorator called Adapter this (3 words)
3. One advantage of Facade
4. Principle that wasn't as easy as it sounded (two words)
7. A _____ adds new behavior
8. Masquerading as a Duck
9. Example that violates the Principle of Least Knowledge: System.out._____
12. No movie is complete without this
13. Adapter client uses the _____ interface
18. An Adapter and a Decorator can be said to _____ an object

Exercise solutions

Let's say we also need an adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:

Now we are adapting Turkeys to Ducks so we implement the Turkey interface

```
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();
        }
    }
}
```

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.

Do either of these 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();
    }
}
```

Violates the Principle of Least Knowledge! You are calling the method of an object returned from another call.

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

Exercise 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 Enumaration.

```java
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
Decorator	Convert one interface to another
Adapter	Don't alter interface, but add responsibility
Facade	Make interface simpler

Exercise solutions

1 FALSE
3 D
4 L 5 CONVERTS 2 IMP
  E C
6 RAIDERS OF THE LOST ARK
R A S T K N
E C O R A T R
D U P L I N G
7 DECORATOR
8 TURKEY
9 P R I N T I N T
10 AC ADAPTER
11 TWO WAY
A S S T H O U G H
12 P O C O R N
13 T A R G E T
14 ALLOWS
15 FLY
16 WRAPPERS
17 FACADE
18 W R A P
19 HOME THEATER

8 the Template Method Pattern

Encapsulating Algorithms

> Yeah, he's a great boss until it comes to getting down in this hole, then it ALL becomes MY job. See what I mean? He's nowhere in sight!
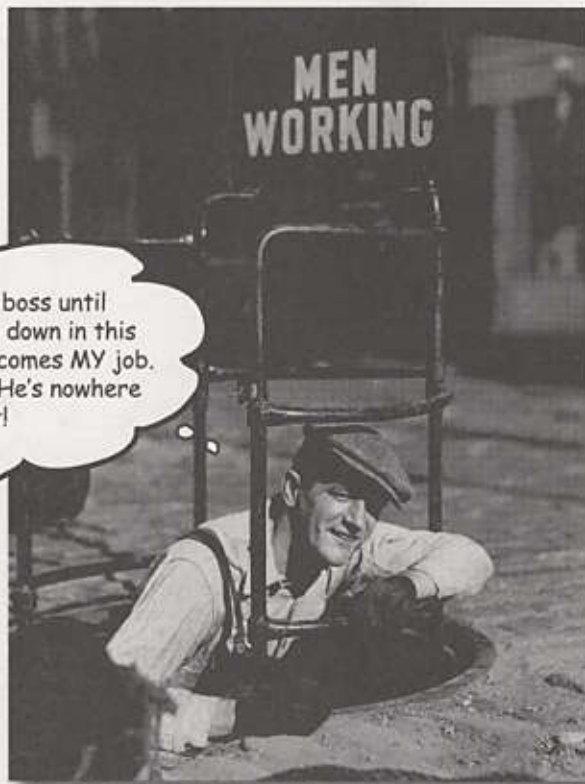
We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas... what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.

It's time for some more caffeine

Some people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine of course!

But there's more; tea and coffee are made in very similar ways. Let's check it out:

Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon
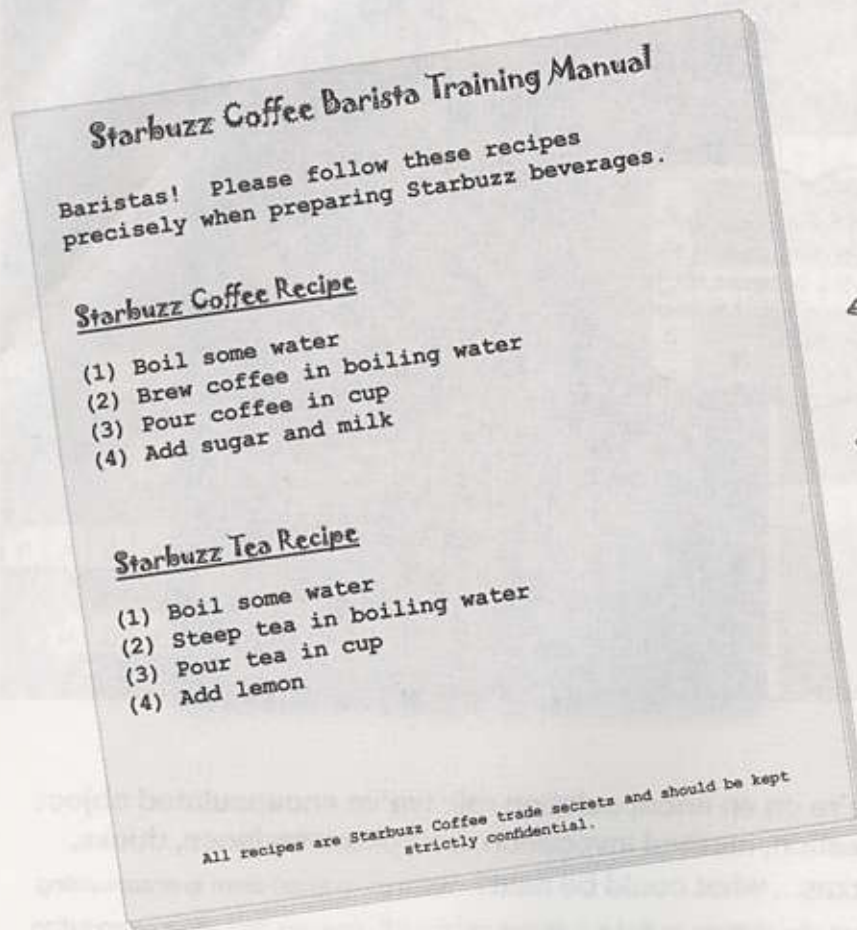
The recipe for coffee looks a lot like the recipe for tea, doesn't it?

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

Whipping up some coffee and tea classes (in Java)

Let's play "coding barista" and write some code for creating coffee and tea.

Here's the coffee:

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```java
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup and add sugar and milk.

and now the Tea...

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:
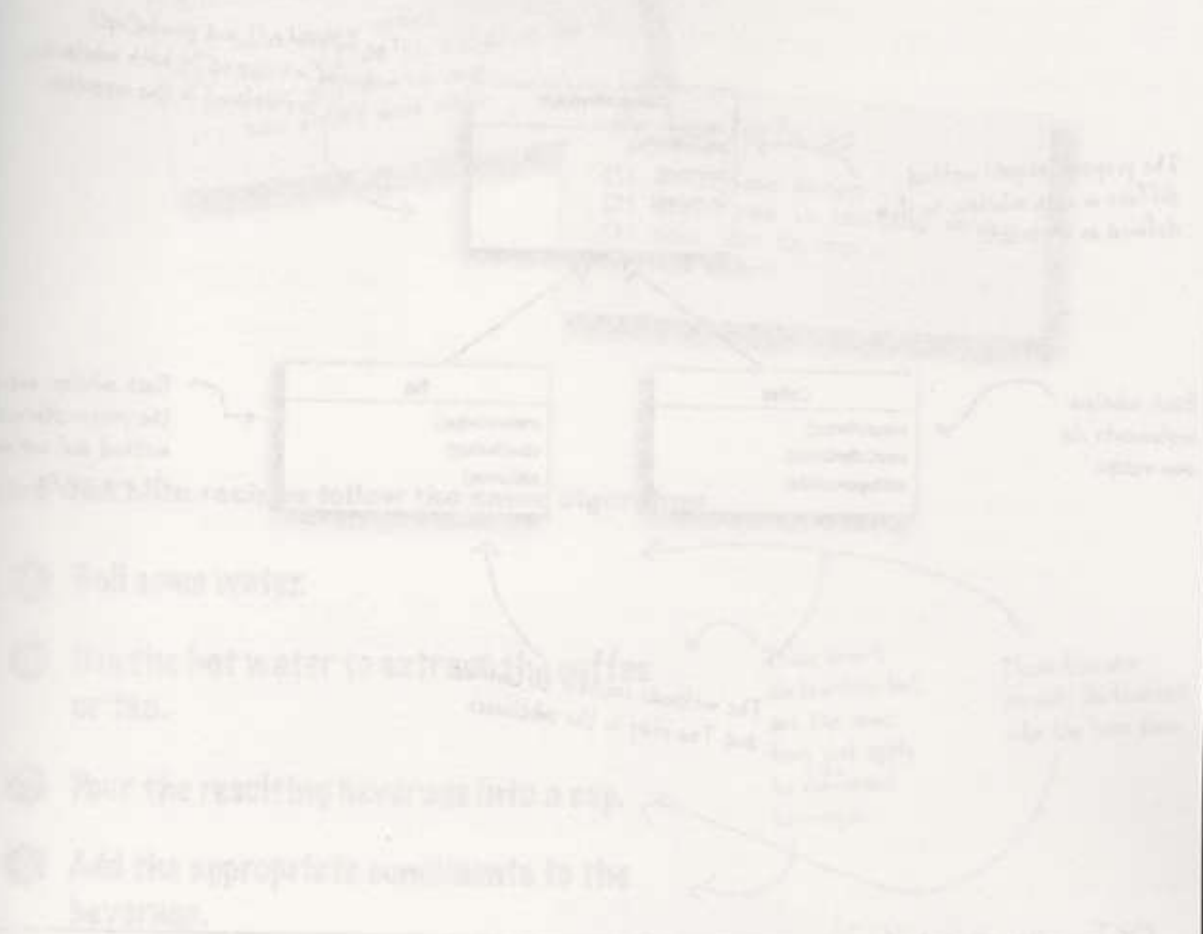
Sir, may I abstract your Coffee, Tea?

It looks like we've got a pretty straightforward design
exercise on our hands with the Coffee and Tea classes.
Your first cut might have looked something like this:

The boilWater() and pourInCup()
methods are shared by both subclasses,
so they are defined in the superclass.

The prepareRecipe() method
differs in each subclass, so it is
defined as abstract.

CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()

Each subclass
implements its
own recipe.

Coffee

prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()

Tea

prepareRecipe()
steepTeaBag()
addLemon()

Each subclass overrides
the prepareRecipe()
method and implements
its own recipe.

The methods specific to Coffee
and Tea stay in the subclasses.

BRAIN POWER

Did we do a good job on the redesign? Hmmmm, take another look. Are we overlooking some other
commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

So what else do Coffee and Tea have in common? Let's start with
the recipes.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon
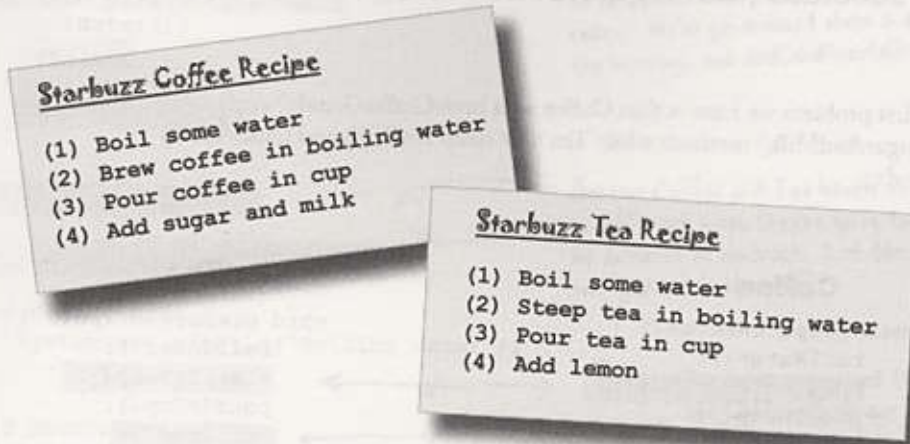
Notice that both recipes follow the same algorithm:

1 Boil some water.

2 Use the hot water to extract the coffee
or tea.

*These aren't
abstracted, but
are the same,
they just apply
to different
beverages.*

*These two are
already abstracted
into the base class.*

3 Pour the resulting beverage into a cup.

4 Add the appropriate condiments to the
beverage.
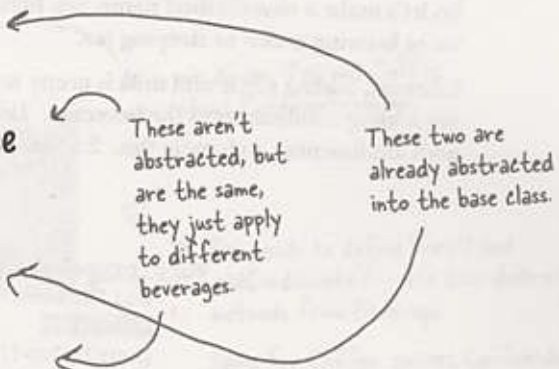
So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

① The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.

Coffee
```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea
```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```
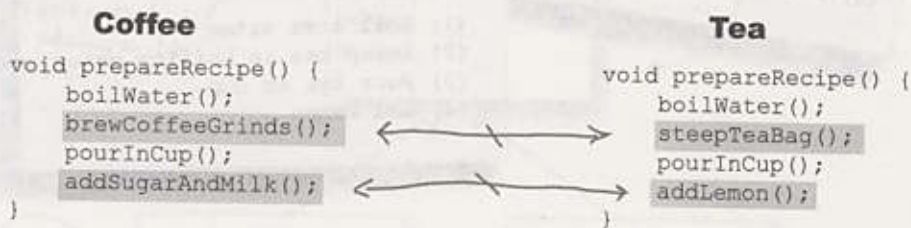
Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

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 prepareRecipe() method will look like this:

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

② Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass:

CaffeineBeverage is abstract, just like in the class design.

```java
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addConditions();
    }

    abstract void brew();

    abstract void addConditions();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

③ Finally we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

As in our design, Tea and Coffee now extend CaffeineBeverage.

```java
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addConditions() {
        System.out.println("Adding Lemon");
    }
}
```

Tea needs to define brew() and addCondiments() — the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```java
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addConditions() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Sharpen your pencil
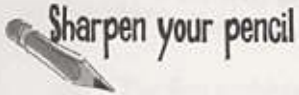
Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:
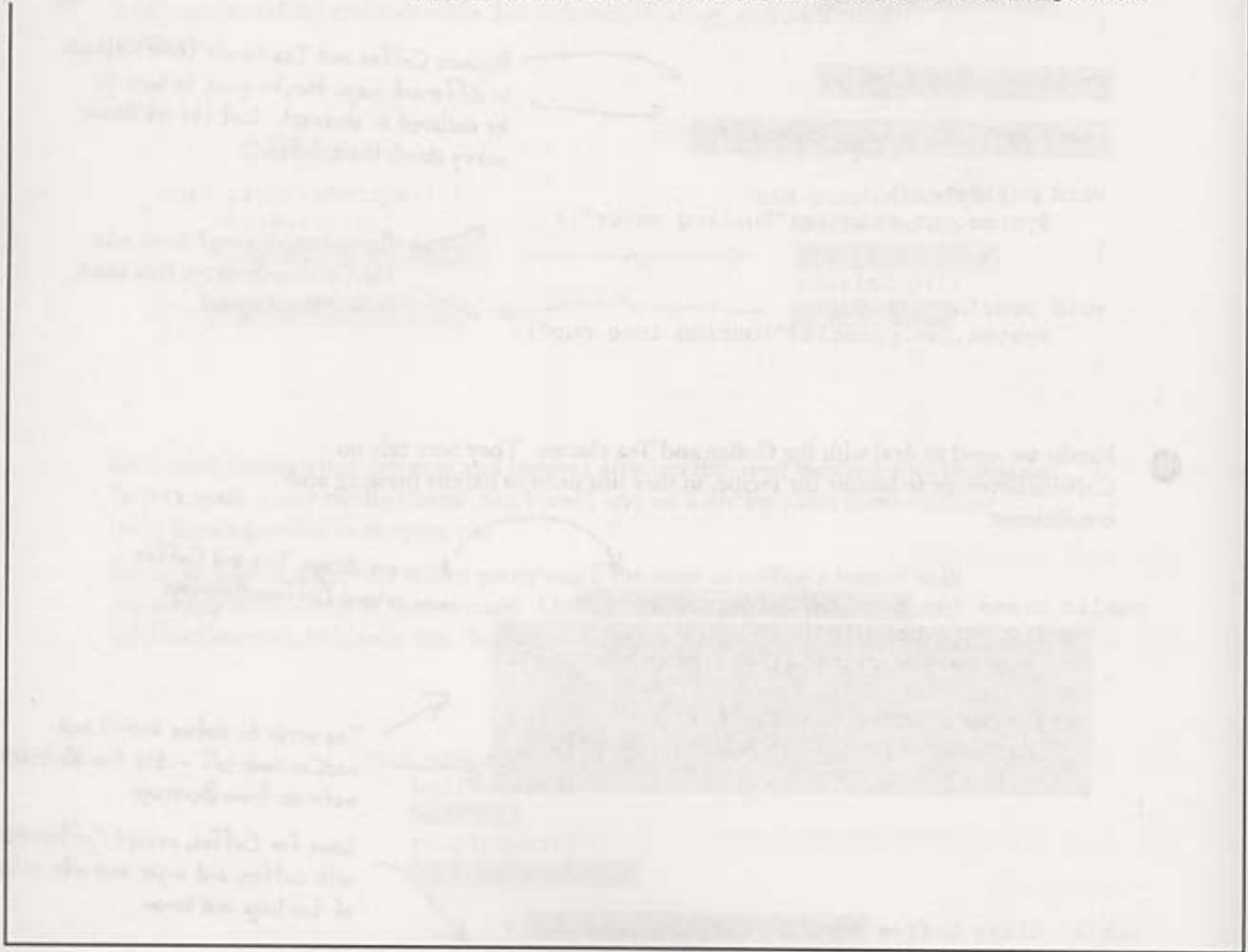
What have we done?

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize →

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

← generalize

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

Coffee subclass

2. Steep the teabag in the water
4. Add lemon

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

2. Brew the coffee grinds
4. Add sugar and milk

Meet the Template Method

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method:"

```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {

        boilWater();

        brew();

        pourInCup();

        addCondiments();

    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }

}
```

prepareRecipe() is our template method. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Let's make some tea...

Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

Behind
the Scenes

```
boilWater();
brew();
pourInCup();
addConditions();
```

1 Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

2 Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

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.

```
CaffeineBeverage
─────────────────
prepareRecipe()
boilWater()
pourInCup()
```

4 Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

5 Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

```
pourInCup();
```

```
Tea
───────────────
brew()
addCondiments();
```

6 Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addConditions();
```

What did the Template Method get us?

Underpowered Tea & Coffee implementation

New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.	The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
Code is duplicated across Coffee and Tea.	The CaffeineBeverage class maximizes reuse among the subclasses.
Code changes to the algorithm require opening the subclasses and making multiple changes.	The algorithm lives in one place and code changes only need to be made there.
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	The Template Method version provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
Knowledge of the algorithm and how to implement it is distributed over many classes.	The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

> **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

Let's check out the class diagram:

The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

The AbstractClass contains the template method.

...and abstract versions of the operations used in the template method.

AbstractClass

| templateMethod() |
| primitiveOperation1() |
| primitiveOperation2() |

| primitiveOperation1(); |
| primitiveOperation2(); |

ConcreteClass

| primitiveOperation1() |
| primitiveOperation2() |

There may be many ConcreteClasses, each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...

Code Way Up Close

Now we're going to look even closer at the types of method that can go in the abstract class:

We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Hooked on Template Method...

A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

There are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later:

> With a hook, I can override the method, or not. It's my choice. If I don't, the abstract class provides a default implementation.

```java
public abstract class CaffeineBeverageWithHook {

    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 a 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 !

```java
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.

Let's run the TestDrive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee

```java
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

← Create a tea.

← A coffee.

And call prepareRecipe() on both!

And let's give it a run...

```
File Edit Window Help send-more-honestea
%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

A steaming cup of tea, and yes, of course we want that lemon!

And a nice hot cup of coffee, but we'll pass on the waistline expanding condiments

Now, I would have thought that functionality like asking the customer could have been used by all subclasses?

You know what? We agree with you. But you have to admit before you thought of that it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

---

there are no Dumb Questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: What are hooks really supposed to be used for?

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass' implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen, or just happened. For instance, a hook method like justReOrderedList() allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: Does a subclass have to implement all the abstract methods in the AbstractClass?

A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: It seems like I should keep my abstract methods small in number, otherwise it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes this can be done by not making the steps of your algorithm too granular. But it's obviously a trade off: the less granularity, the less flexibility.

Remember, too, that some steps will be optional; so you can implement these as hooks rather than abstract classes, easing the burden on the subclasses of your abstract class.

The Hollywood Principle

We've got another design principle for you; it's called the Hollywood Principle:

> You've heard me say it before, and I'll say it again: don't call me, I'll call you!

> ## The Hollywood Principle
> *Don't call us, we'll call you.*

Easy to remember, right? But what has it got to do with OO design?

The Hollywood principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components a "don't call us, we'll call you" treatment.

Low-level components can participate in the computation.

But the high-level components control when and how.

A low-level component never calls a high-level component directly.

High-Level Component

Low-Level Component

Another Low-Level Component
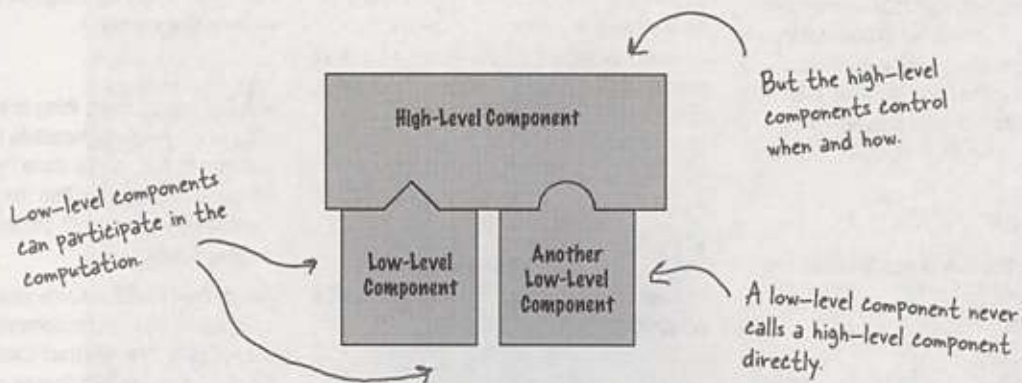
The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How? Let's take another look at our CaffeineBeverage design:

CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.

CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()
brew()
addConditments()

Coffee

brew()
addCondiments()

Tea

brew()
addCondiments()

The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being "called" first.

BRAIN POWER
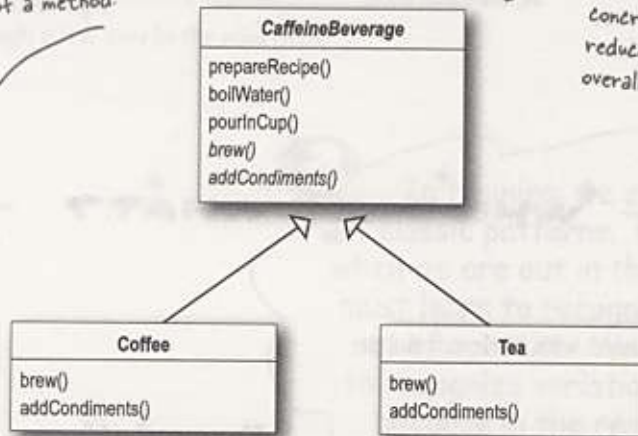
What other patterns make use of the Hollywood Principle?

The Factory Method, Observer, any others?

there are no
Dumb Questions

Q: How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

A: The Dependency Inversion Principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. The Hollywood Principle is a technique for building frameworks or components so that lower-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers. So, they both have the goal of decoupling, but the Dependency Inversion Principle makes a much stronger and general statement about how to avoid dependencies in design.

The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.

Q: Is a low-level component disallowed from calling a method in a higher-level component?

A: Not really. In fact, a low level component will often end up calling a method defined above it in the inheritance hierarchy purely through inheritance. But we want to avoid creating explicit circular dependencies between the low-level component and the high-level ones.

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Factory Method	Subclasses decide which concrete classes to create
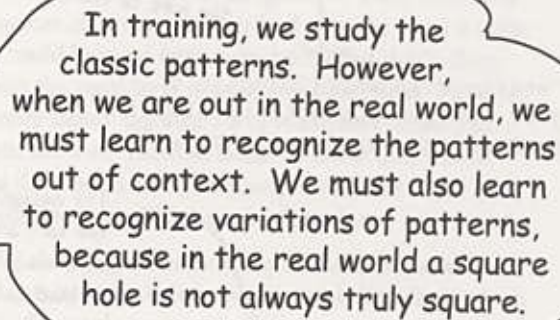
Template Methods in the Wild

The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild. You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.

This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...

> In training, we study the classic patterns. However, when we are out in the real world, we must learn to recognize the patterns out of context. We must also learn to recognize variations of patterns, because in the real world a square hole is not always truly square.

Sorting with Template Method

What's something we often need to do with arrays?
Sort them!

Recognizing that, the designers of the Java Arrays class
have provided us with a handy template method for
sorting. Let's take a look at how this method operates:

We've paired 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...

We actually have two methods here and they act together to provide the sort functionality.

The first method, sort(), is just a helper method that creates a copy of the array and passes it along as the destination array to the mergeSort() method. It also passes along the length of the array and tells the sort to start at the first element.

```java
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.

Think of this as the template method.

```java
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;
}
```

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?
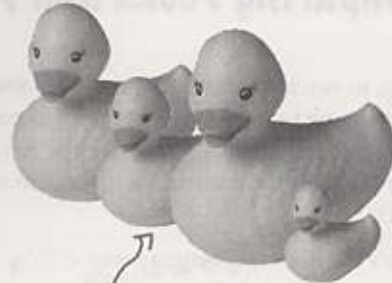
We've got an array of Ducks we need to sort.

No, it doesn't. Aren't we supposed to be subclassing something? I thought that was the point of Template Method. An array doesn't subclass anything, so I don't get how we'd use 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.

Am I greater than you?

I don't know, that's what compareTo() tells us.

Comparing Ducks and Ducks

Okay, so you know that if you want to sort Ducks, you're going to have to implement this compareTo() method; by doing that you'll give the Arrays class what it needs to complete the algorithm and sort your ducks.

Here's the duck implementation:

> Remember, we need to implement the Comparable interface since we aren't really subclassing.
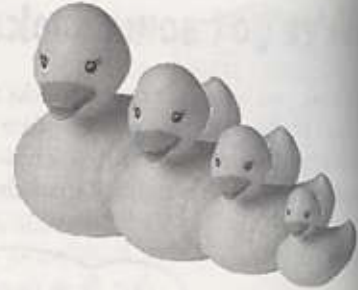
```
public class Duck implements Comparable {
    String name;
    int weight;
```

> Our Ducks have a name and a weight

```
    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }
```

> We're keepin' it simple; all Ducks do is print their name and weight!

```
    public String toString() {
        return name + " weighs " + weight;
    }
```

> Okay, here's what sort needs...

```
    public int compareTo(Object object) {

        Duck otherDuck = (Duck)object;
```

> compareTo() takes another Duck to compare THIS Duck to.

```
        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

> 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.

Let's sort some Ducks

Here's the test drive for sorting Ducks...

```java
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]);
        }
    }
}
```

We need an array of Ducks; these look good.

Notice that we call Arrays' static method sort, and pass it our Ducks.

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.

Let the sorting commence!

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive

Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

The unsorted Ducks

The sorted Ducks

The making of the sorting duck machine

Let's trace through how the Arrays sort() template method works. We'll check out how the template method controls the algorithm, and at certain points in the algorithm, how it asks our Ducks to supply the implementation of a step...

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

① First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

The sort() method controls the algorithm, no class can change this sort() counts on a Comparable class to provide the implementation of compareTo()

② Then 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()) control the sort procedure.

③ To sort an array, you need to compare two items one by one until the entire list is in sorted order.

When it comes to comparing two ducks, the sort method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to:

```
ducks[0].compareTo(ducks[1]);
```

First Duck

Duck to compare it to

Duck
compareTo()
toString()

No inheritance, unlike a typical template method.
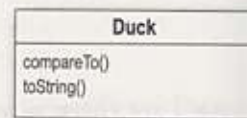
④ If the Ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays:

```
swap()
```

Arrays
sort()
swap()

⑤ The sort method continues comparing and swapping Ducks until the array is in the correct order!

there are no
Dumb Questions

Q: Is this really the Template Method Pattern, or are you trying too hard?

A: The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps – and the Arrays sort is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints.

The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted.

So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.

Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?

A: You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way – we're *using* the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the *entire* algorithm. The algorithm that Arrays implements for sort is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.

Q: Are there other examples of template methods in the Java API?

A: Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the tempate method read(byte b[], int off, int len).

BRAIN POWER

We know that we should favor composition over inheritance, right? Well, the implementers of the sort() template method decided not to use inheritance and instead to implement sort() as a static method that is composed with a Comparable at runtime. How is this better? How is it worse? How would you approach this problem? Do Java arrays make this particularly tricky?

BRAIN² POWER

Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

Swingin' with Frames

Up next on our Template Method safari... keep your eye out for swinging JFrames!

If you haven't encountered JFrame, it's the most basic Swing container and inherits a paint() method. By default, paint() does nothing because it's a *hook*! By overriding paint(), you can insert yourself into JFrame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the JFrame. Here's an embarrassingly simple example of using a JFrame to override the paint() hook method:

> We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```

> Don't look behind the curtain! Just some initialization here...

> JFrame's update algorithm calls paint(). By default, paint() does nothing... it's a hook. We're overriding paint(), and telling the JFrame to draw a message in the window.

Head First Design Patterns

I rule!!
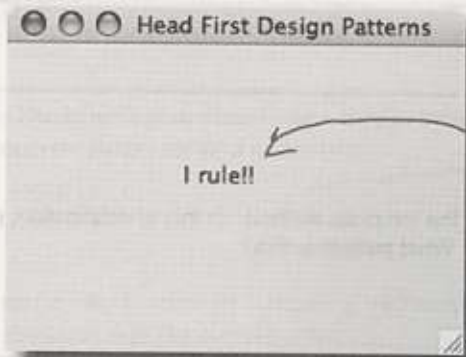
> Here's the message that gets painted in the frame because we've hooked into the paint() method.

Applets

Our final stop on the safari: the applet.

You probably know an applet is a small program that runs in a web page. Any applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them:

```java
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
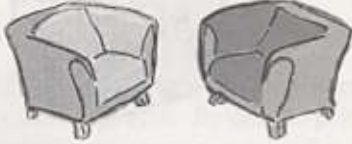
Tonight's talk: **Template Method and Strategy compare methods.**

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. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.

Strategy

Factory Method

Hey, I heard that!

Nope, it's me, although be careful — you and Factory Method are related, aren't you?

I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern..." So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

I'm not sure I'd put it quite like *that*... and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

Template Method

I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

Strategy

You *might* be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose *me* for Chapter 1 for nothing!

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

Yeah, I guess... but, what about dependency? You're way more dependent than me.

How's that? My superclass is abstract.

But you have to depend on methods implemented in your superclass, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

Like I said Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway, I'm always glad to help.

Got it. Don't call us, we'll call you...

It's that time again....
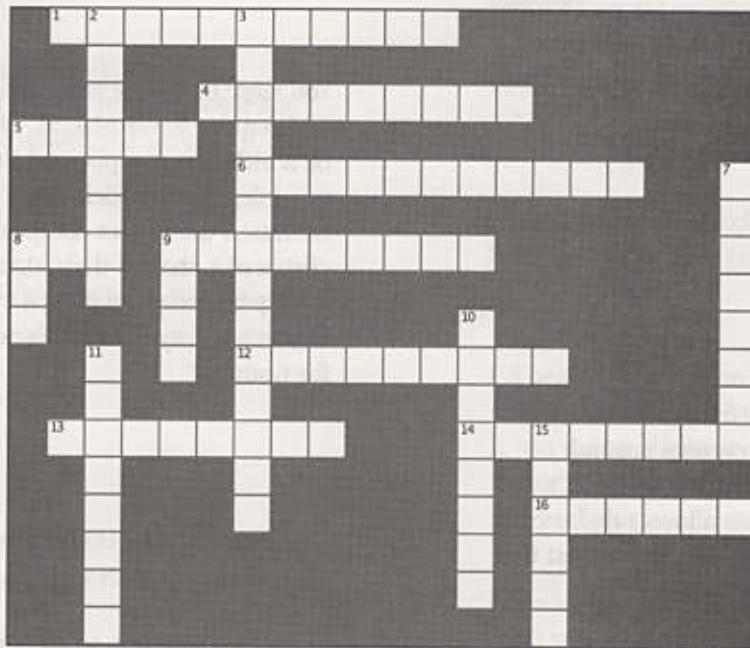
Across

1. Strategy uses _____ rather than inheritance
4. Type of sort used in Arrays
5. The JFrame hook method that we overrode to print "I Rule"
6. The Template Method Pattern uses _____ to defer implementation to other classes
8. Coffee and _____ the same. Thus
9. Don't call us, we'll call you is known as the _____ Principle
12. A template method defines the steps of an _____
13. In this chapter we gave you more _____
14. The template method is usually defined in an _____ class
16. Class that likes web pages

Down

2. _____ algorithm steps are implemented by hook methods
3. Factory Method is a _____ of Template Method
7. The steps in the algorithm that must be supplied by the subclasses are usually declared _____
8. Huey, Louie and Dewey all weigh _____ pounds
9. A method in the abstract superclass that does nothing or provides default behavior is called a _____ method
10. Big headed pattern
11. Our favorite coffee shop in Objectville
15. The Arrays class implements its template method as a _____ method

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method you can reuse code like a pro while keeping control of your algorithms.

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you.

Our newest principle reminds you that your superclasses are running the show, so let them call your subclasses when they're needed, just like they do in Hollywood.

OO Patterns

And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Exercise solutions

Sharpen your pencil

Draw the new class diagram now that we've moved prepareRecipe() into the CaffeineBeverage class:

```
          CaffeineBeverage
         ┌──────────────────┐
         │ prepareRecipe()  │
         │ boilWater()      │
         │ pourInCup()      │
         │ brew()           │
         │ addCondiments()  │
         └──────────────────┘
             ▲          ▲
             │          │
    ┌──────────────┐  ┌──────────────┐
    │    Coffee    │  │     Tea      │
    ├──────────────┤  ├──────────────┤
    │ brew()       │  │ brew()       │
    │ addCondiments() │ addCondiments() │
    └──────────────┘  └──────────────┘
```

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Factory Method	Subclasses decide which concrete classes to create

Exercise solutions

Crossword solution:

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

14 Appendix

Appendix: Leftover Patterns

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

Bridge

Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

A scenario

Imagine you're going to revolutionize "extreme lounging." You're writing the code for a new ergonomic and user-friendly remote control for TVs. You already know that you've got to use good OO techniques because while the remote is based on the same *abstraction*, there will be lots of *implementations* – one for each model of TV.

This is an abstraction. It could be an interface or an abstract class.

Every remote has the same abstraction. →

```
RemoteControl
-----------------
on()
off()
setChannel()
// more methods
```

Lots of implementations, one for each TV.

```
RCAControl
-----------------
on()
off()
setChannel()
// more methods
```

```
SonyControl
-----------------
on()
off()
setChannel()
// more methods
```

```
{
    tuneChannel(channel);
}
```
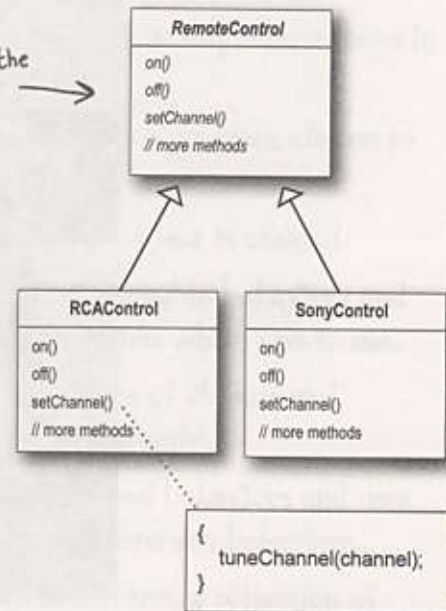
Your dilemma

You know that the remote's user interface won't be right the first time. In fact, you expect that the product will be refined many times as usability data is collected on the remote control.

So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to *vary the abstraction* because it is going to change over time as the remote is improved based on the user feedback.

So how are you going to create an OO design that allows you to vary the implementation *and* the abstraction?

Using this design we can vary only the TV implementation, not the user interface.

Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation *and* the abstraction by placing the two in separate class hierarchies.

Abstraction class hierarchy.

The relationship between the two is referred to as the "bridge."

Implementation class hierarchy.

RemoteControl

| implementor |
| on() |
| off() |
| setChannel() |
| // more methods |

Has-A

TV

| on() |
| off() |
| tuneChannel() |
| // more methods |

implementor.tuneChannel(channel);

All methods in the abstraction are implemented in terms of the implementation.

ConcreteRemote

| currentStation |
| on() |
| off() |
| setStation() |
| nextChannel() |
| previousChannel() |
| // more methods |

RCA

| on() |
| off() |
| tuneChannel() |
| // more methods |

Sony

| on() |
| off() |
| tuneChannel() |
| // more methods |

setChannel(currentStation + 1);

Concrete subclasses are implemented in terms of the abstraction, not the implementation.

Now you have two hierarchies, one for the remotes and a separate one for platform specific TV implementations. The bridge allows you to vary either side of the two hierarchies independently.

Bridge Benefits

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don't affect the client.

Bridge Uses and Drawbacks

- Useful in graphic and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

Builder

Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

A scenario

You've just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this:

Each vacation is planned over some number of days.

Each day can have any combination of hotel reservations, tickets, meals and special events.
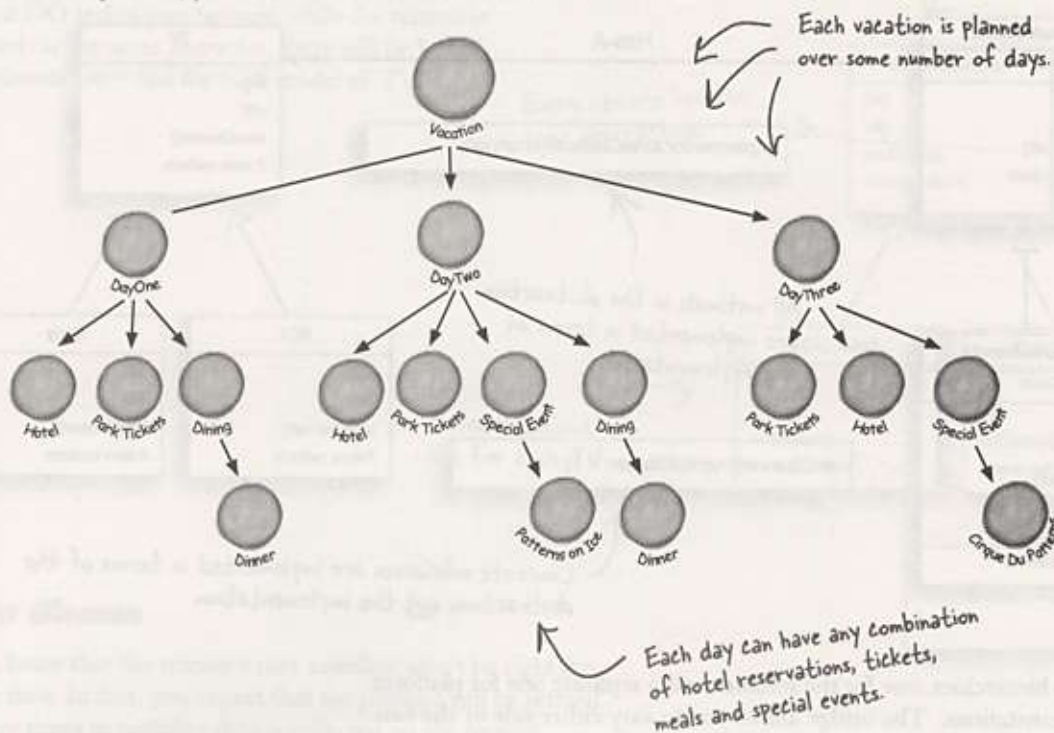
You need a flexible design

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be flying into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

Why use the Builder Pattern?

Remember Iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here: we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.

The client uses an abstract interface to build the planner.

The Client directs the builder to construct the planner.

Client

constructPlanner()

builder →

AbstractBuilder

buildDay()
addHotel()
addReservation()
addSpecialEvent()
addTickets()
getVacationPlanner()

The concrete builder creates real products and stores them in the vacation composite structure.

```
builder.buildDay(date);
builder.addHotel(date, "Grand Facadian");
builder.addTickets("Patterns on Ice");

// plan rest of vacation

Planner yourPlanner =
        builder.getVacationPlanner();
```

VacationBuilder

vacation

buildDay()
addHotel()
addReservation()
addSpecialEvent()
addTickets()
getVacationPlanner()

The Client directs the builder to create the planner in a number of steps and then calls the getVacationPlanner() method to retrieve the complete object.

Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

Builder Uses and Drawbacks

- Often used for building composite structures.
- Constructing objects requires more domain knowledge of the client than when using a Factory.

Chain of Responsibility

Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

A scenario

Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine. From their own analysis they get four kinds of email: fan mail from customers that love the new 1 in 10 game, complaints from parents whose kids are addicted to the game and requests to put machines in new locations. They also get a fair amount of spam.

All fan mail needs to go straight to the CEO, all complaints go to the legal department and all requests for new machines go to business development. Spam needs to be deleted.

Your task

Mighty Gumball has already written some AI detectors that can tell if an email is spam, fan mail, a complaint, or a request, but they need you to create a design that can use the detectors to handle incoming email.

> You've got to help us deal with the flood of email we're getting since the release of the Java Gumball Machine.
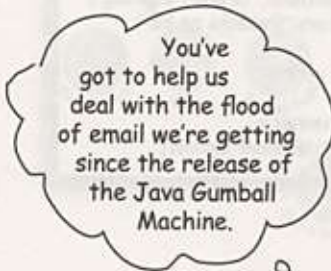
How to use the Chain of Responsibility Pattern

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.

Handler
successor
handleRequest()

SpamHandler
handleRequest()

FanHandler
handleRequest()

ComplaintHandler
handleRequest()

NewLocHandler
handleRequest()
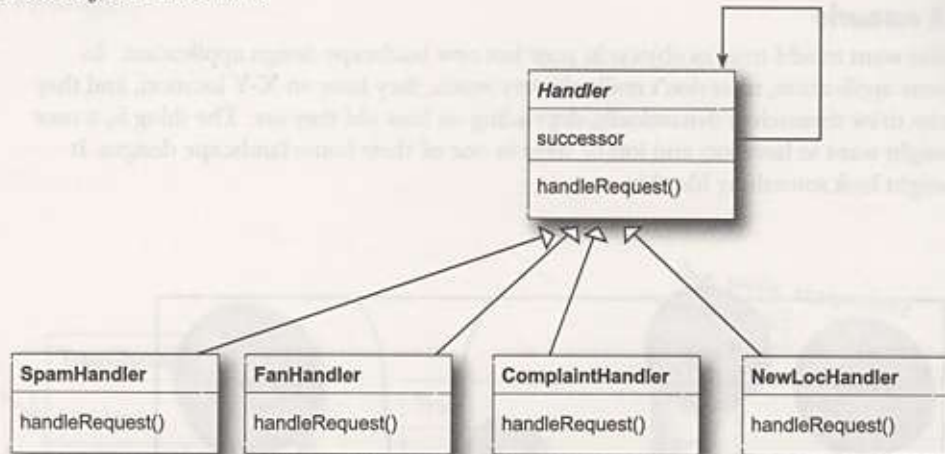
As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain — although, you can always implement a catch-all handler.

Spam Handler → Fan Handler → Complaint Handler → NewLoc Handler

Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe the runtime characteristics and debug.

Flyweight

Use the Flyweight Pattern when one instance of a class can be used to provide many "virtual instances."

A scenario

You want to add trees as objects in your hot new landscape design application. In your application, trees don't really do very much; they have an X-Y location, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this:
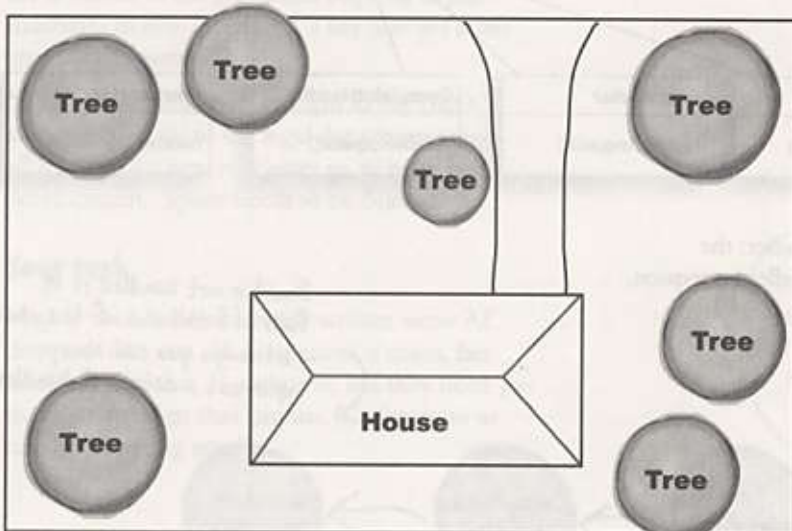
Each Tree instance maintains its own state.

```
Tree

xCoord
yCoord
age

display() {
   // use X-Y coords
   // & complex age
   // related calcs
}
```

Your big client's dilemma

You've just landed your "reference account." That key client you've been pitching for months. They're going to buy 1,000 seats of your application, and they're using your software to do the landscape design for huge planned communities. After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish...

Why use the Flyweight Pattern?

What if, instead of having thousands of Tree objects, you could redesign your system so that you've got only one instance of Tree, and a client object that maintains the state of ALL your trees? That's the Flyweight!

All the state, for ALL of your virtual Tree objects, is stored in this 2D-array.

TreeManager

treeArray

displayTrees() {
  // for all trees {
    // get array row
    display(x, y, age);
  }
}
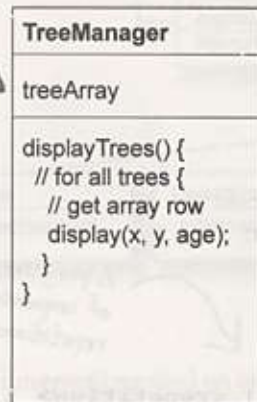
One, single, state-free Tree object

Tree

display(x, y, age) {
  // use X-Y coords
  // & complex age
  // related calcs
}
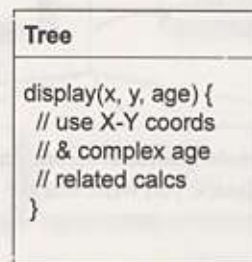
Flyweight Benefits

- Reduces the number of object instances at runtime, saving memory.
- Centralizes state for many "virtual" objects into a single location.

Flyweight Uses and Drawbacks

- The Flyweight is used when a class has many instances, and they can all be controlled identically.
- A drawback of the Flyweight pattern is that once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

Interpreter

Use the Interpreter Pattern to build an interpreter for a language.

A scenario

Remember the Duck 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:

> *Relax*
>
> The Interpreter Pattern requires some knowledge of formal grammars. If you've never studied formal grammars, go ahead and read through the pattern; you'll still get the gist of it.

```
right;
while (daylight) fly;
quack;
```

← Turn the duck right

← Fly all day...

← ...and then quack.

Now, remembering how to create grammars from one of your old introductory programming classes, you write out the grammar:

A program is an expression consisting of sequences of commands and repetitions ("while" statements).

```
expression  ::=  <command> | <sequence> | <repetition>
sequence    ::= <expression> ';' <expression>
command     ::= right | quack | fly
repetition  ::= while '(' <variable> ')' '<expresion>
variable    ::= [A-Z,a-z]+
```

A sequence is a set of expressions separated by semicolons.

We have three commands: right, quack, and fly.

A while statement is just a conditional variable and an expression.

Now what?

You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

How to implement an interpreter

When you need to implement a simple language, the
Interpreter Pattern defines a class-based representation for its
grammar along with an interpreter to interpret its sentences.
To represent the language, you use a class to represent each
rule in the language. Here's the duck language translated
into classes. Notice the direct mapping to the grammar.
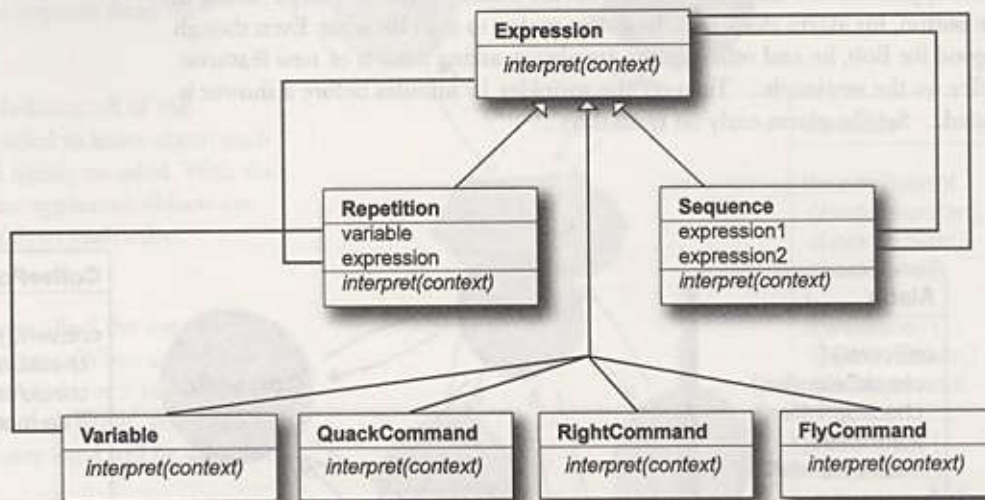
To interpret the language, call the interpret() method on each
expression type. This method is passed a context – which
contains the input stream of the program we're parsing – and
matches the input and evaluates it.

Interpreter Benefits

- Representing each grammar rule in a class makes the language easy to implement.
- Because the grammar is represented by classes, you can easily change or extend the language.
- By adding additional methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation.

Interpreter Uses and Drawbacks

- Use interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases a parser/compiler generator may be more appropriate.

Mediator
Use the Mediator Pattern to centralize complex communications and control between related objects.

A scenario

Bob has a Java-enabled auto-house, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other clients are always asking for lots of new features: No coffee on the weekends... Turn off the sprinkler 15 minutes before a shower is scheduled... Set the alarm early on trash days...

Alarm

```
onEvent() {
  checkCalendar()
  checkSprinkler()
  startCoffee()
  // do more stuff
}
```

CoffeePot

```
onEvent() {
  checkCalendar()
  checkAlarm()
  // do more stuff
}
```

Calendar

```
onEvent() {
  checkDayOfWeek()
  doSprinkler()
  doCoffee()
  doAlarm()
  // do more stuff
}
```

Sprinkler

```
onEvent() {
  checkCalendar()
  checkShower()
  checkTemp()
  checkWeather()
  // do more stuff
}
```
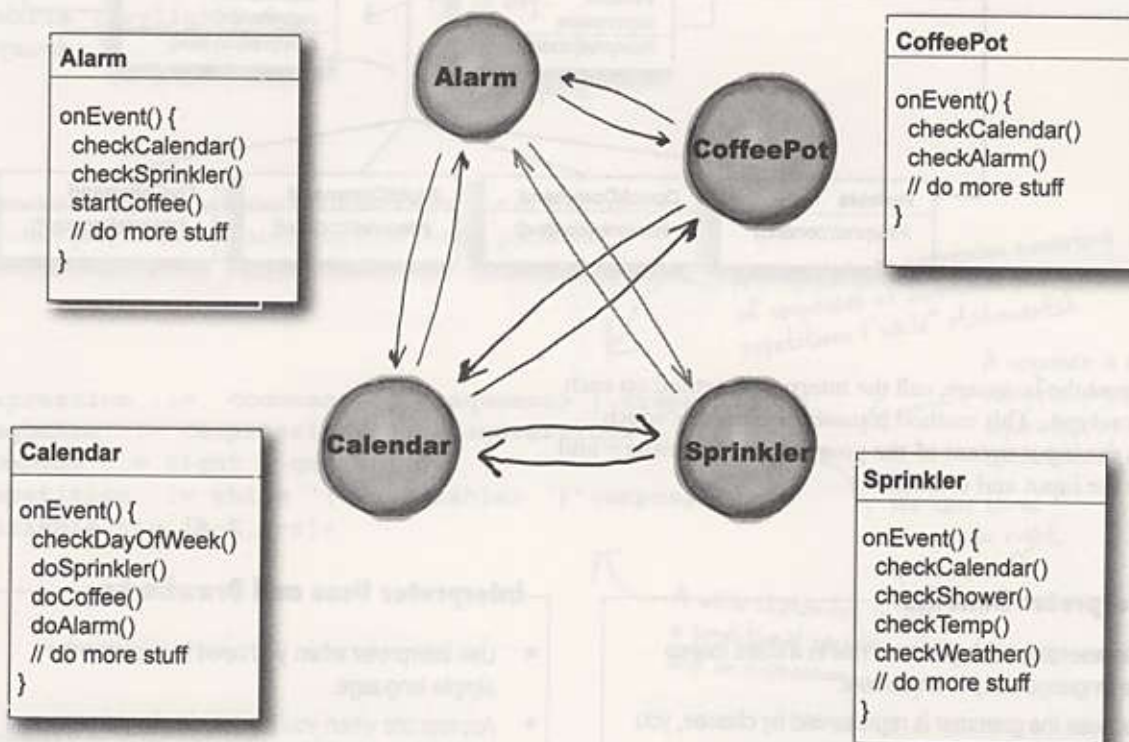
HouseOfTheFuture's dilemma

It's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified:

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before 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 *completely decoupled* from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.

> It's such a relief, not having to figure out that Alarm clock's picky rules!

Alarm

CoffeePot

Mediator

Calendar

Sprinkler

Mediator

```
if(alarmEvent){
  checkCalendar()
  checkShower()
  checkTemp()
}
if(weekend) {
  checkWeather()
  // do more stuff
}
if(trashDay) {
  resetAlarm()
  // do more stuff
}
```
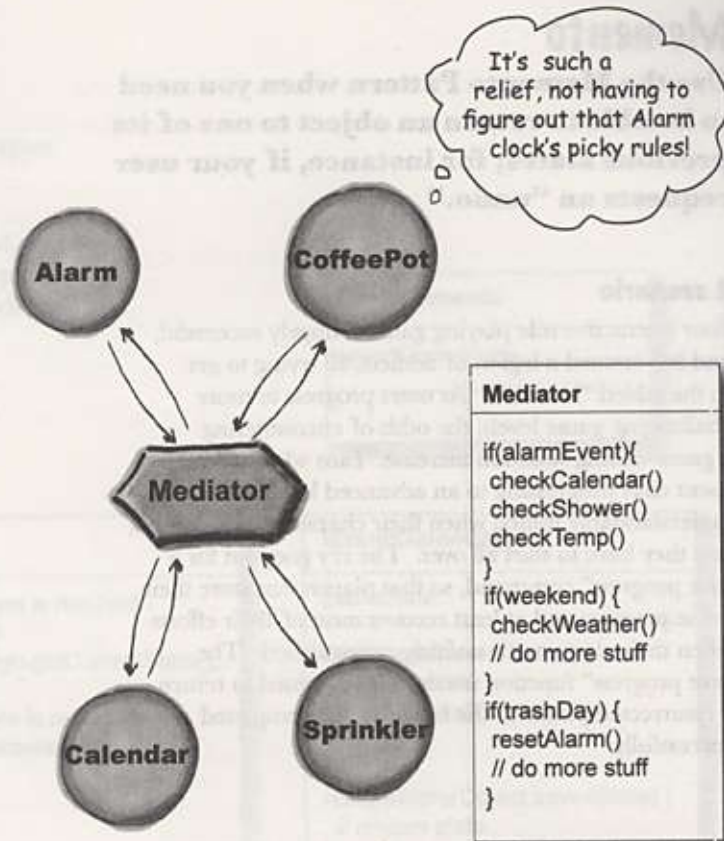
Mediator Benefits

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
- Simplifies maintenance of the system by centralizing control logic.
- Simplifies and reduces the variety of messages sent between objects in the system.

Mediator Uses and Drawbacks
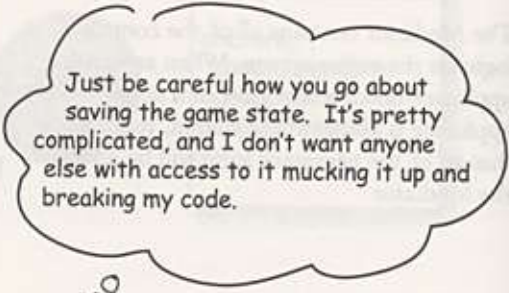
- The Mediator is commonly used to coordinate related GUI components.
- A drawback of the Mediator pattern is that without proper design, the Mediator object itself can become overly complex.

Memento

Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an "undo."

A scenario

Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled "level 13." As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a "save progress" command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The "save progress" function needs to be designed to return a resurrected player to the last level she completed successfully.

> Just be careful how you go about saving the game state. It's pretty complicated, and I don't want anyone else with access to it mucking it up and breaking 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.

GameMemento

savedGameState

Client

```
// when new level is reached
Object saved =
    (Object) mgo.getCurrentState();

// when a restore is required
mgo.restoreState(saved);
```

While this isn't a terribly fancy implementation, notice that the Client has no access to the Memento's data.

MasterGameObject

gameState

```
Object getCurrentState() {
    // gather state
    return(gameState);
}

restoreState(Object savedState) {
    // restore state
}

// do other game stuff
```

--- **Memento Benefits** ---

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object's data encapsulated.
- Provides easy-to-implement recovery capability.

--- **Memento Uses and Drawbacks** ---

- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system's state.

Prototype

Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

A scenario

Your interactive role playing game has an insatiable appetite for monsters. As your heros 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.

It would be a lot cleaner if we could decouple the code that handles the **details** of creating the monsters from the code that actually needs to create the instances on the fly.

Yikes! Just the act of **creating** all of these different kinds of monster instances is getting tricky... Putting all sorts of state detail in the constructors doesn't seem to be very cohesive. It would be great if there was a single place where all of the instantiation details could be encapsulated...

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 de-serialization 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.

```
<<interface>>
Monster
```

```
WellKnownMonster
```

```
DynamicPlayerGeneratedMonster
```

```
MonsterMaker

makeRandomMonster() {
    Monster m =
        MonsterRegistry.getMonster();
}
```
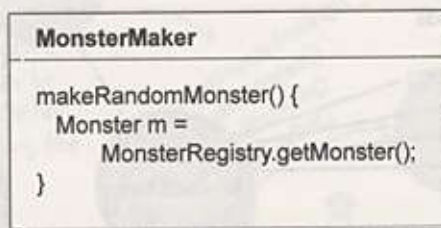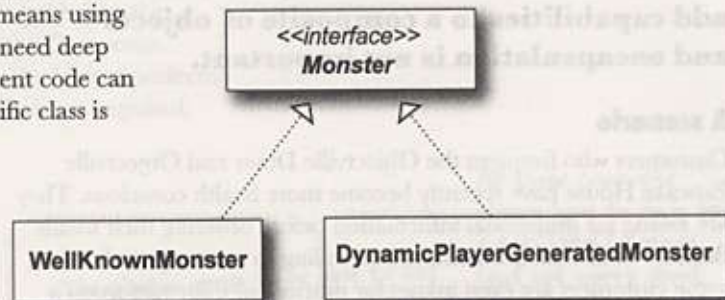
The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)

```
MonsterRegistry

Monster getMonster() {
    // find the correct monster
    return correctMonster.clone();
}
```
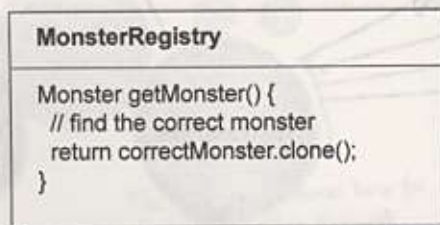
The registry finds the appropriate monster, makes a clone of it, and returns the clone.

Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

Visitor

Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

A scenario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per ingredient basis.

Lou's proposed solution:

// new methods

getHealthRating
getCalories
getProtein
getCarbs

// new methods

getHealthRating
getCalories
getProtein
getCarbs

Menu

MenuItem

MenuItem

Ingredient

Ingredient
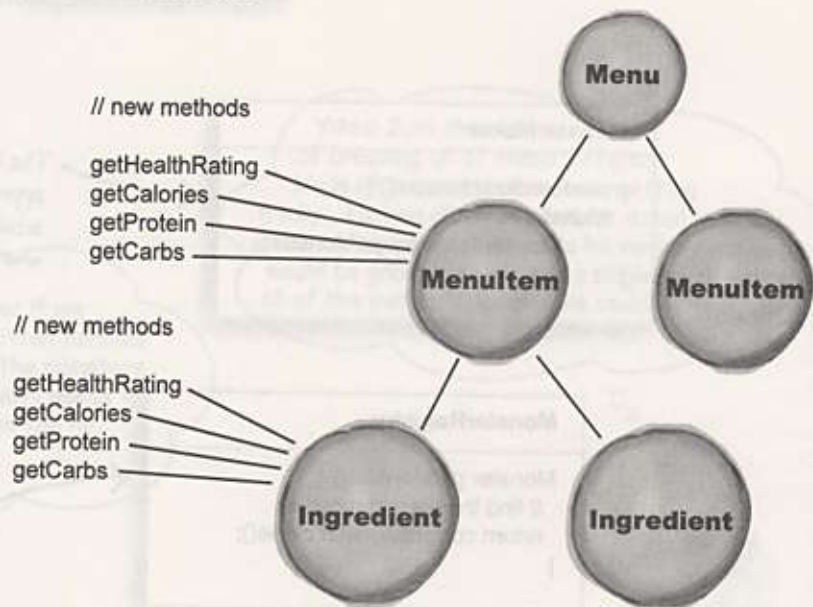
Mel's concerns...

"Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places..."

The Visitor drops by

The Visitor 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.

All these composite classes have to do is add a getState() method (and not worry about exposing themselves).

The Visitor needs to be able to call getState() across classes, and this is where you can add new methods for the client to use.

The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.

getHealthRating()
getCalories()
getProtein()
getCarbs()

Visitor

Client / Traverser

getState()
getState()
getState()
getState()
getState()
getState()

Menu

MenuItem

MenuItem

Ingredient

Ingredient

The Traverser knows how to guide the Visitor through the Composite structure.

Visitor Benefits

- Allows you to add operations to a Composite structure without changing the structure itself.
- Adding new operations is relatively easy.
- The code for operations performed by the Visitor is centralized.

Visitor Drawbacks

- The Composite classes' encapsulation is broken when the Visitor is used.
- Because the traversal function is involved, changes to the Composite structure are more difficult.

Index

A

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

B

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

Colophon

All interior layouts were designed by Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates. Kathy and Bert created the look & feel of the Head First series. The book was produced using Adobe InDesign CS (an unbelievably cool design tool that we can't get enough of) and Adobe Photoshop CS. The book was typeset using Uncle Stinky, Mister Frisky (you think we're kidding), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman LET, Courier New and Woodrow typefaces.

Interior design and production all happened exclusively on Apple Macintoshes—at Head First we're all about "Think Different" (even if it isn't grammatical). All Java code was created using James Gosling's favorite IDE, *vi*, although we really should try Erich Gamma's Eclipse.

Long days of writing were powered by the caffeine fuel of Honest Tea and Tejava, the clean Santa Fe air, and the grooving sounds of Banco de Gaia, Cocteau Twins, Buddha Bar I-VI, Delerium, Enigma, Mike Oldfield, Olive, Orb, Orbital, LTJ Bukem, Massive Attack, Steve Roach, Sasha and Digweed, Thievery Corporation, Zero 7 and Neil Finn (in all his incarnations) along with a heck of a lot of acid trance and more 80s music that you'd care to know about.

And now, a final word from the Head First Institute...

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 protoypes 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@wickedlysmart.com

And next time you're in Objectville, drop by and take one of our behind the scenes laboratory tours.

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

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.

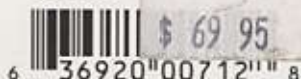Visit O'Reilly on the Web at *www.oreilly.com*

ISBN 0-596-00712-4   US $44.95

In Canada
$ 69 95

9 780596 007126   6 36920 00712 8

O'REILLY®