

1 intro to design patterns

Welcome to Design Patterns

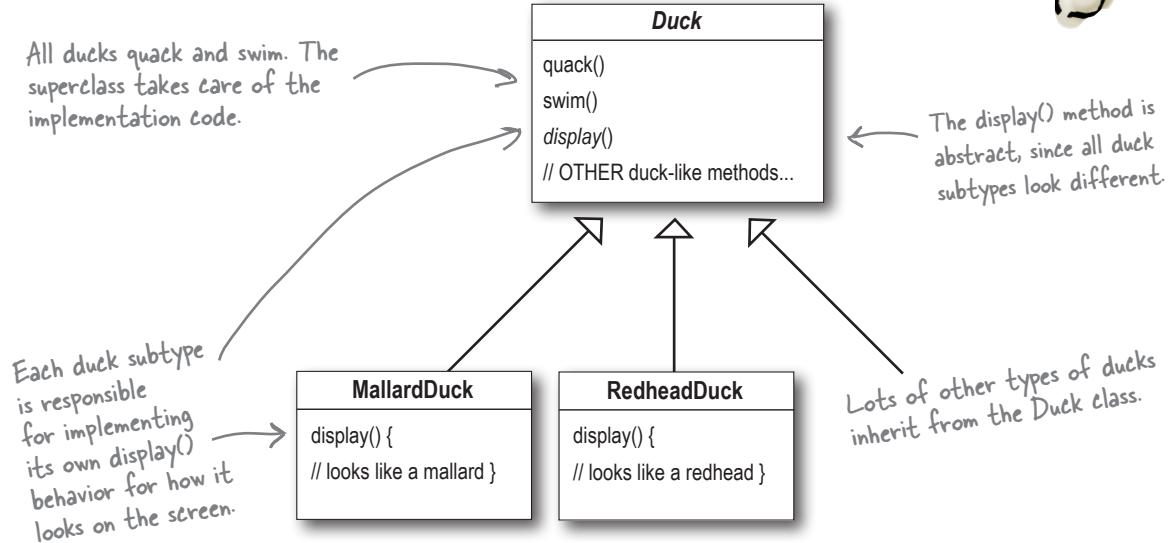
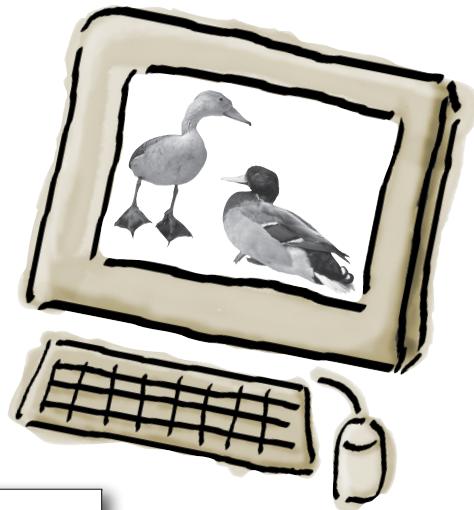


Now that we're living in Objectville, we've just got to get into Design Patterns... everyone is doing them. Soon we'll be the hit of Jim and Betty's Wednesday night patterns group!

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

It started with a simple SimUDuck app

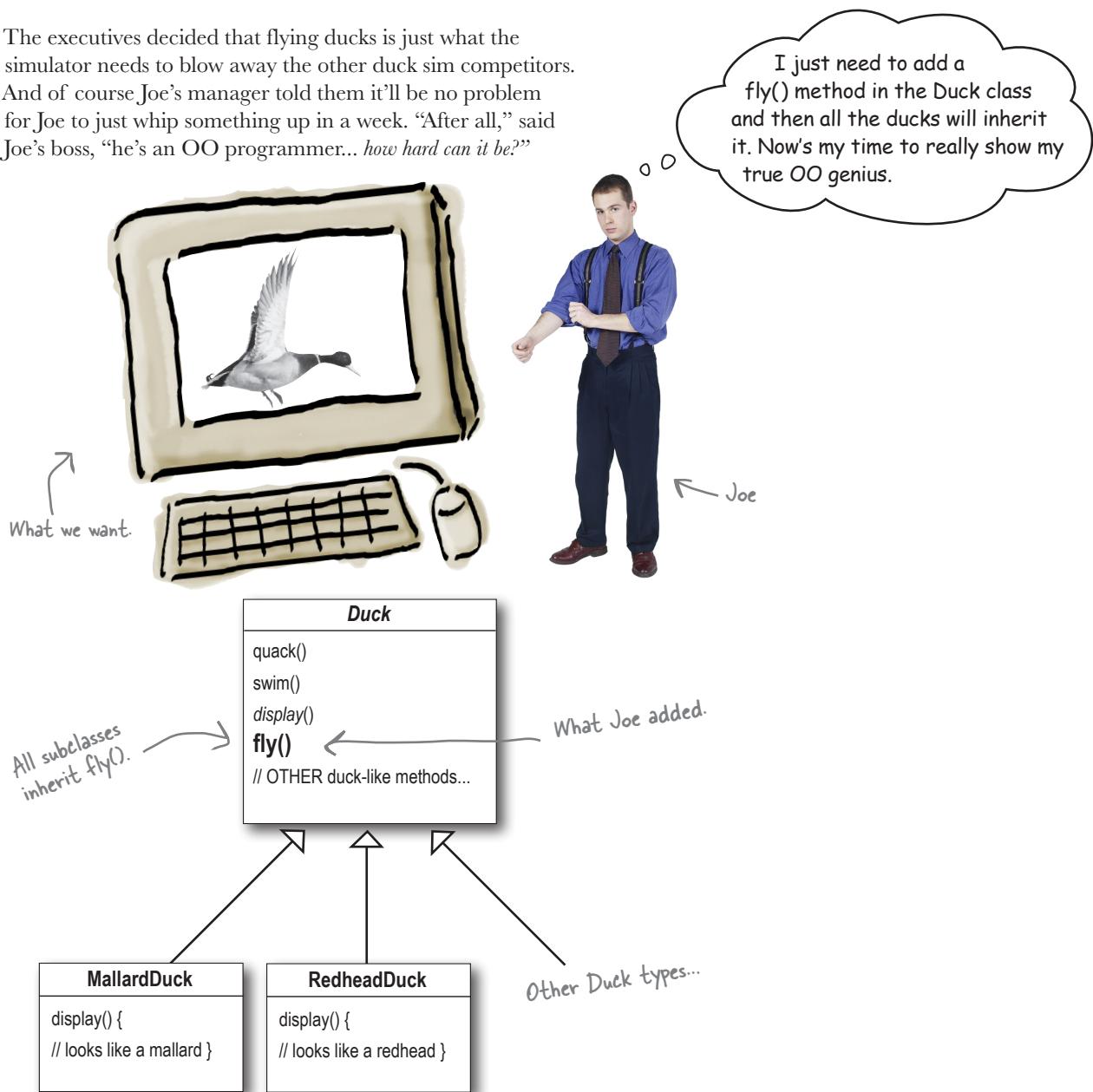
Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck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



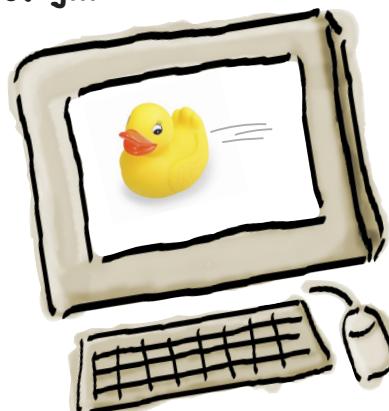
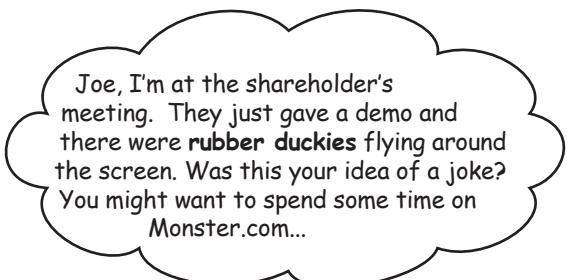
In the last year, the company has been under increasing pressure from competitors. After a week long off-site brainstorming session over golf, the company executives think it's time for a big innovation. They need something *really* impressive to show at the upcoming shareholders meeting in Maui *next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

But now we need the ducks to FLY

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



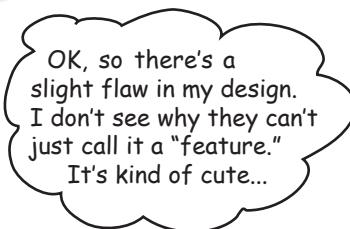
But something went horribly wrong...



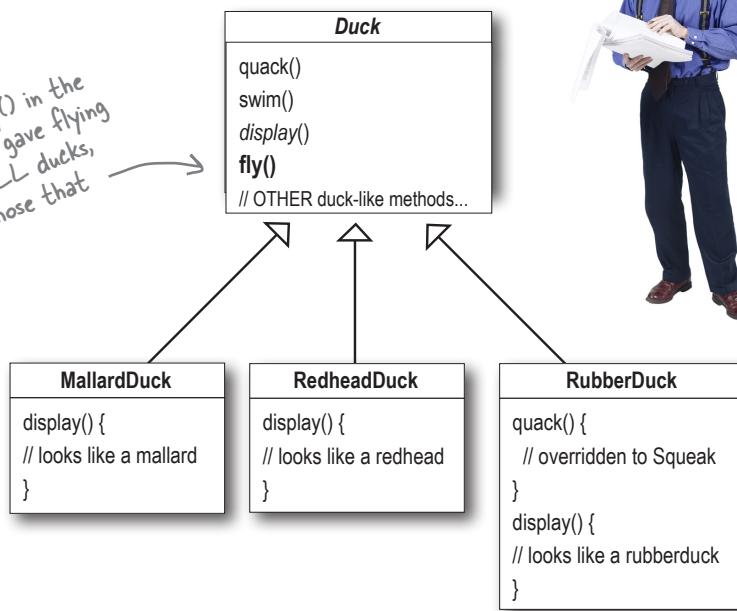
What happened?

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

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



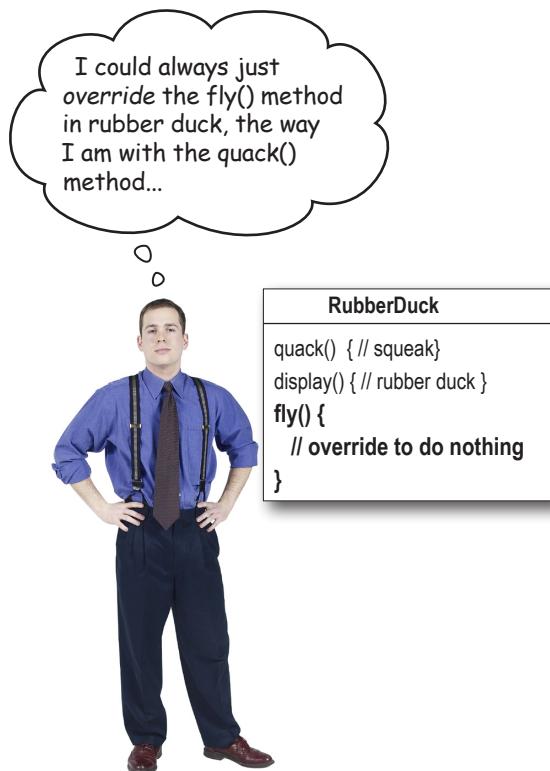
By putting `fly()` in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.



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

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

Joe thinks about inheritance...



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



Sharpen your pencil

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

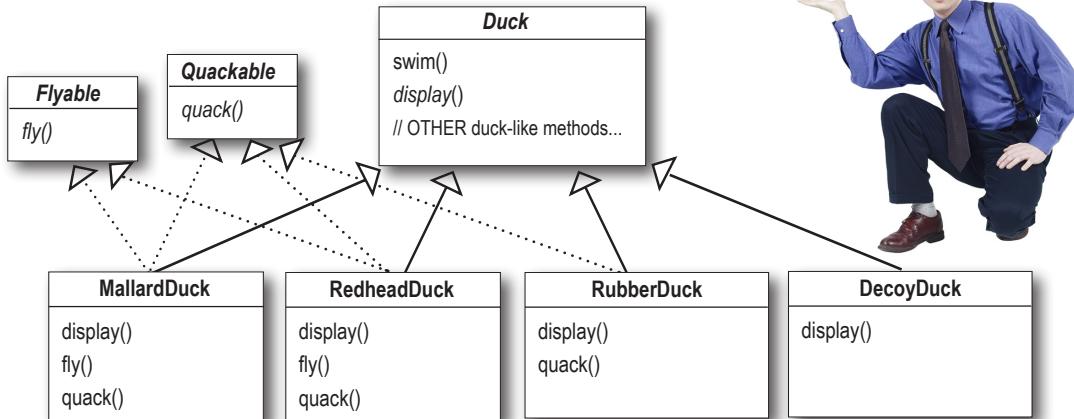
- A. Code is duplicated across subclasses.
- B. Runtime behavior changes are difficult.
- C. We can't make ducks dance.
- D. Hard to gain knowledge of all duck behaviors.
- E. Ducks can't fly and quack at the same time.
- F. Changes can unintentionally affect other ducks.

How about an interface?

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

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

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



What do YOU think about this design?

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



What would you do if you were Joe?

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

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



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

The one constant in software development

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

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

CHANGE

(use a mirror to see the answer)

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



Sharpen your pencil

Lots of things can drive change. List some reasons you've had to change code in your applications (we put in a couple of our own to get you started).

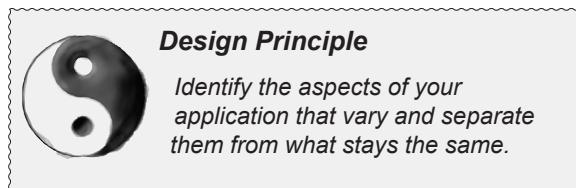
My customers or users decide they want something else, or they want new functionality.

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

Zeroing in on the problem...

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

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



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

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

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

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

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

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

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

Separating what changes from what stays the same

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

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

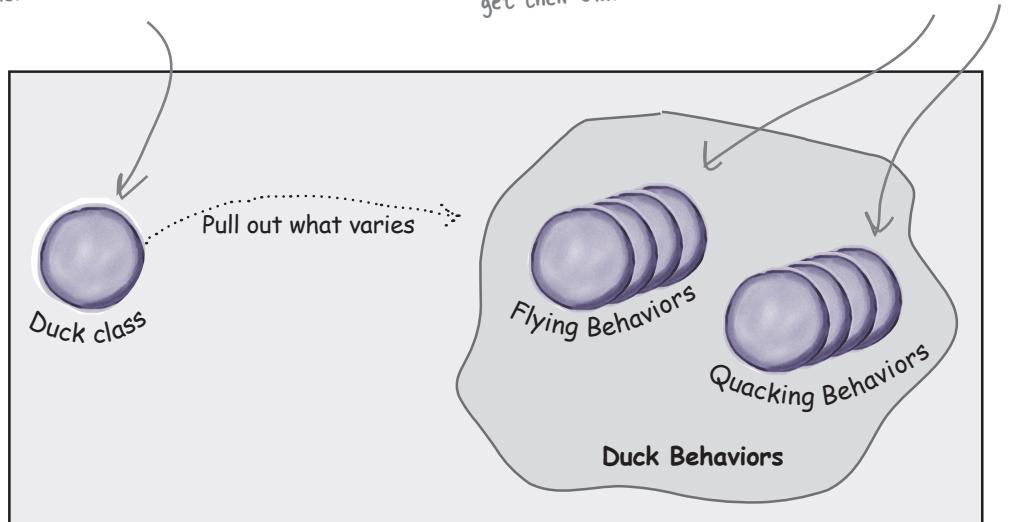
We know that `fly()` and `quack()` are the parts of the Duck class that vary across ducks.

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

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

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

Various behavior implementations are going to live here.



Designing the Duck Behaviors

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

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

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



Design Principle

Program to an interface, not an implementation.

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

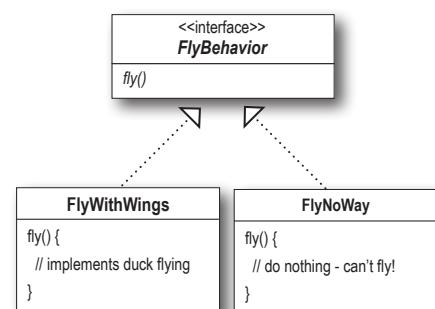
So this time it won't be the *Duck* classes that will implement the flying and quacking interfaces. Instead, we'll make a set of classes whose entire reason for living is to represent a behavior (for example, "squeaking"), and it's the *behavior* class, rather than the *Duck* class, that will implement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

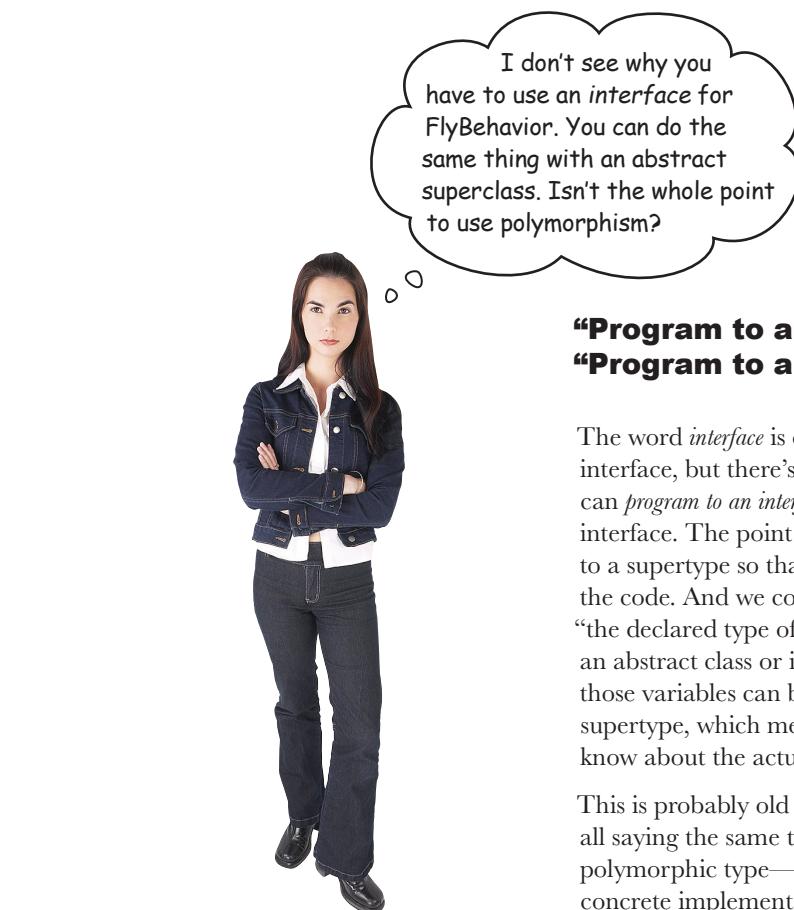
This is in contrast to the way we were doing things before, where a behavior came either from a concrete implementation in the superclass Duck, or by providing a specialized implementation in the subclass itself. In both cases we were relying on an *implementation*. We were locked into using that specific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

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

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

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





“Program to an *interface*” really means “Program to a supertype.”

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

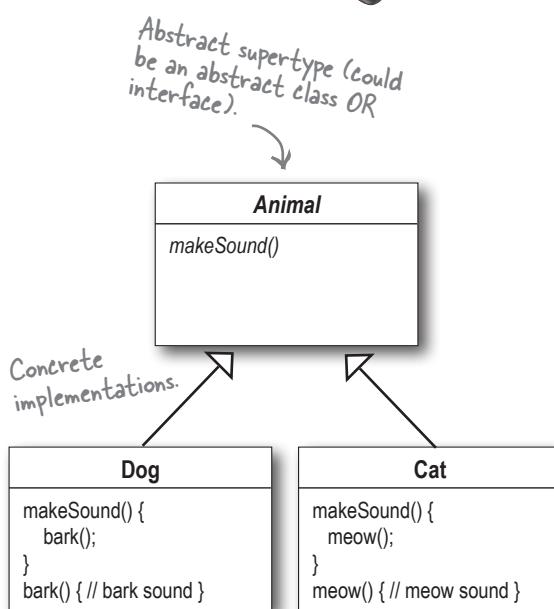
This is probably old news to you, but just to make sure we're all saying the same thing, here's a simple example of using a polymorphic type—imagine an abstract class *Animal*, with two concrete implementations, *Dog* and *Cat*.

Programming to an implementation would be:

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

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

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

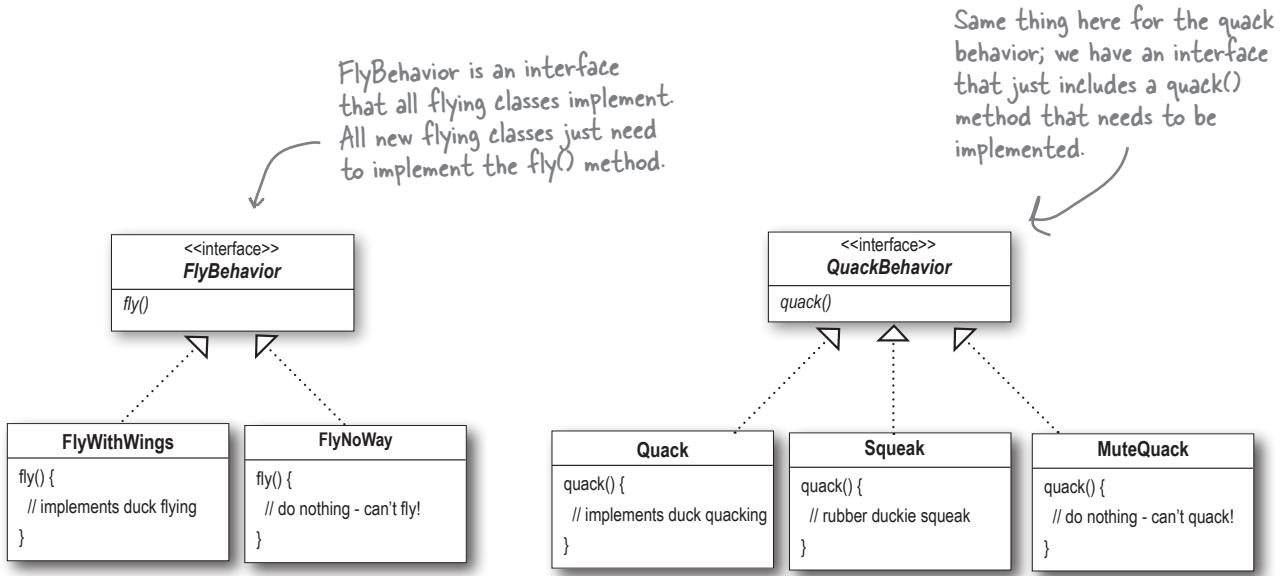


Even better, rather than hardcoding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime**:

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

Implementing the Duck Behaviors

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



Here's the implementation for all ducks that can't fly.
And here's the implementation of flying for all ducks that have wings.

Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

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

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

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

there are no Dumb Questions

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

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

Q: Should we make Duck an interface too?

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

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

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



Sharpen your pencil

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

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

Answers:

- 1) Create a FlyRocketPowered class that implements the FlyBehavior interface.
- 2) One example, a duck call (a device that makes duck sounds).

Integrating the Duck Behavior

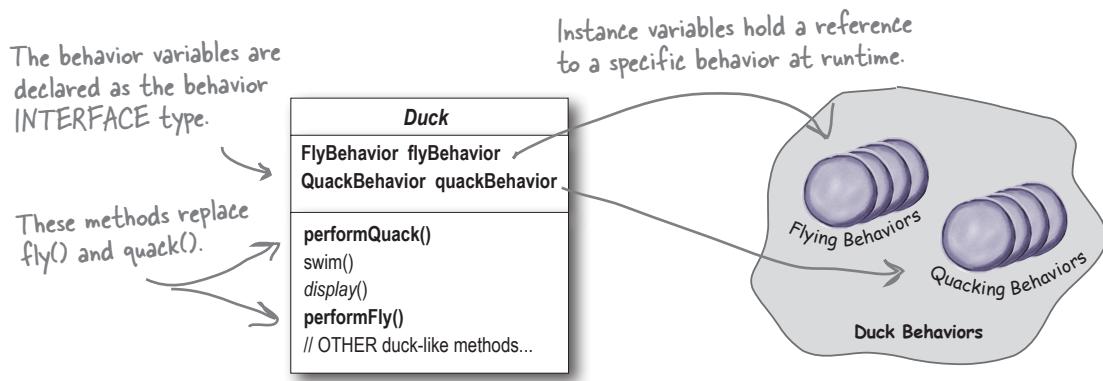
The key is that a Duck will now delegate its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).

Here's how:

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

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

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



- Now we implement `performQuack()`:

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

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

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

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

Pretty simple, huh? To perform the quack, a Duck just allows the object that is referenced by `quackBehavior` to quack for it.

In this part of the code we don't care what kind of object it is, ***all we care about is that it knows how to quack()***!

More integration...

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

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

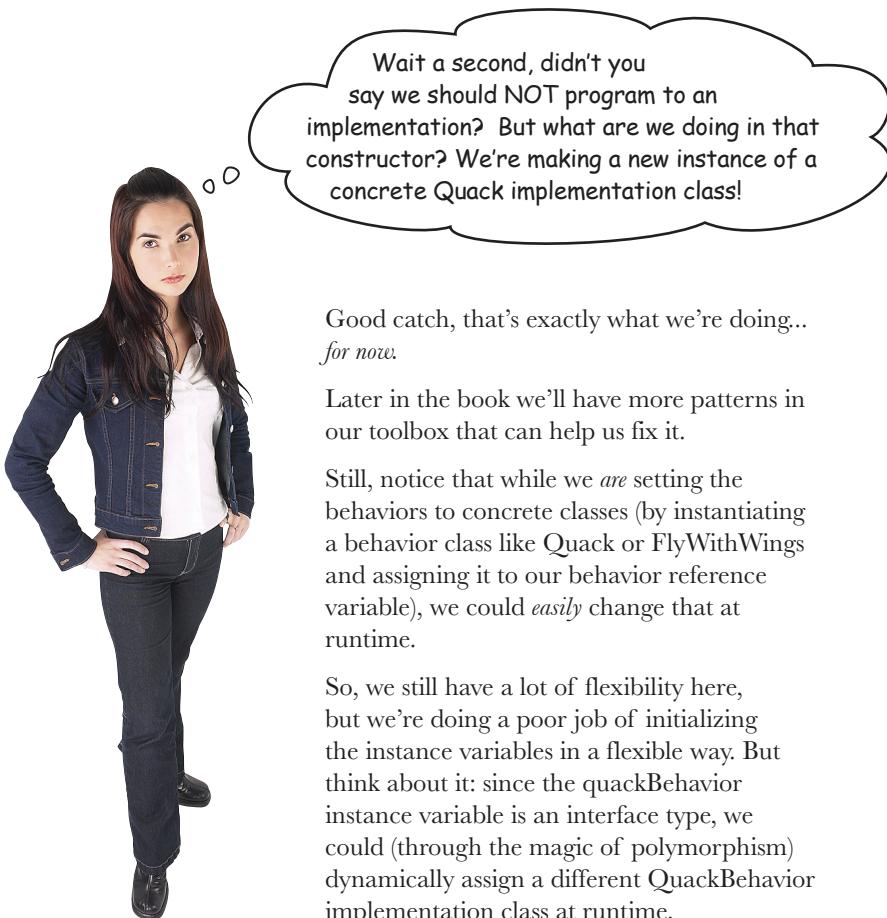
Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

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

And it uses FlyWithWings as its FlyBehavior type.

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

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



Wait a second, didn't you say we should NOT program to an implementation? But what are we doing in that constructor? We're making a new instance of a concrete Quack implementation class!

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

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

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

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

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

Testing the Duck code

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

```
public abstract class Duck {  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }  
  
    public abstract void display();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

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

Delegate to the behavior class.

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

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


---

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


---

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

The interface that all flying behavior classes implement.

Flying behavior implementation for ducks that DO fly...

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

Testing the Duck code, continued...

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

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

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

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

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

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

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

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

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

- ⑤ Run the code!**

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

Setting behavior dynamically

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

① Add two new methods to the Duck class:

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

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

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

Editor note: gratuitous pun – fix

Duck
FlyBehavior flyBehavior;
QuackBehavior quackBehavior;
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...

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

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

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

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

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

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

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



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

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

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

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

- ⑤ Run it!

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

Before

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

After

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

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

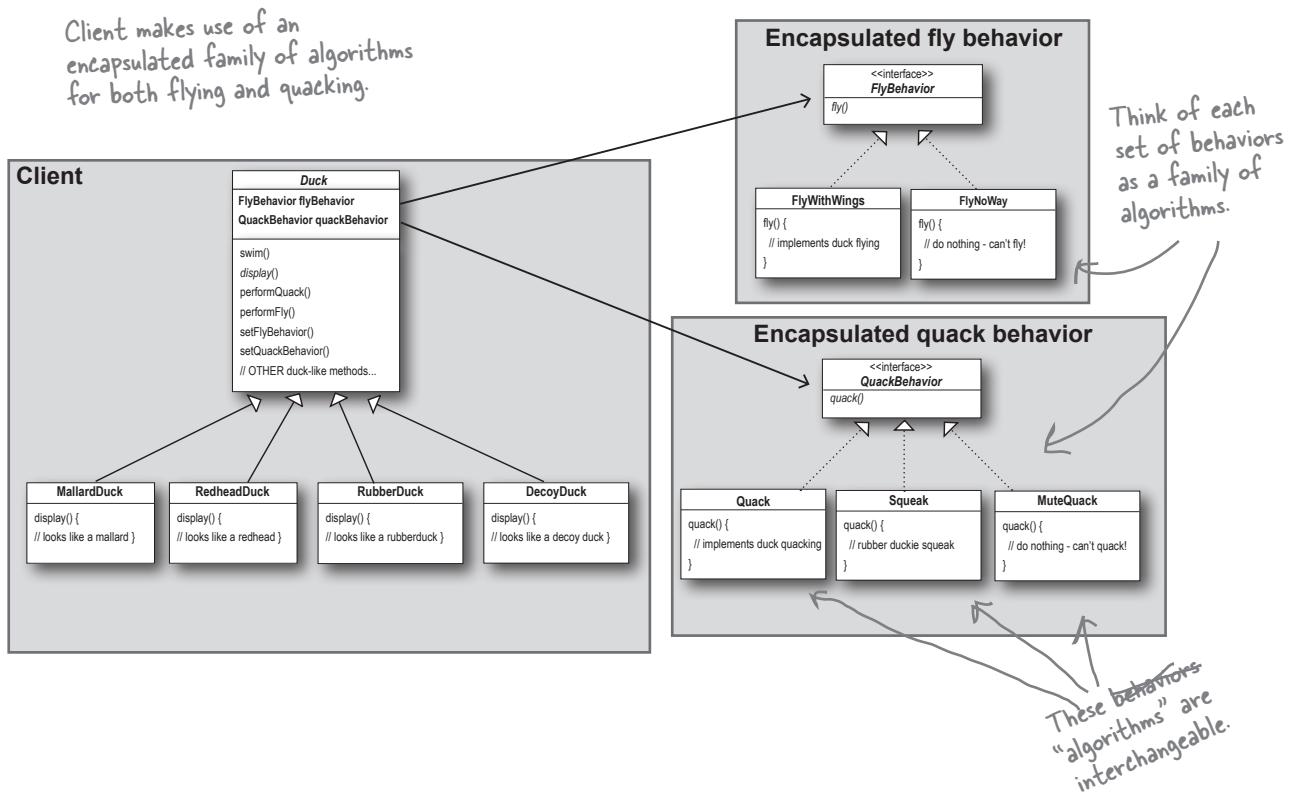
The Big Picture on encapsulated behaviors

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

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

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

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



HAS-A can be better than IS-A

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

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

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



Design Principle

Favor composition over inheritance.

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

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



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



Master and Student...

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

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

Master: Grasshopper, continue...

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

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

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

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

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

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

Speaking of Design Patterns...



Congratulations on
your first pattern!

You just applied your first design pattern—the **STRATEGY Pattern**. That's right, you used the Strategy Pattern to rework the SimUDuck app. Thanks to this pattern, the simulator is ready for any changes those execs might cook up on their next business trip to Maui.

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

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

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



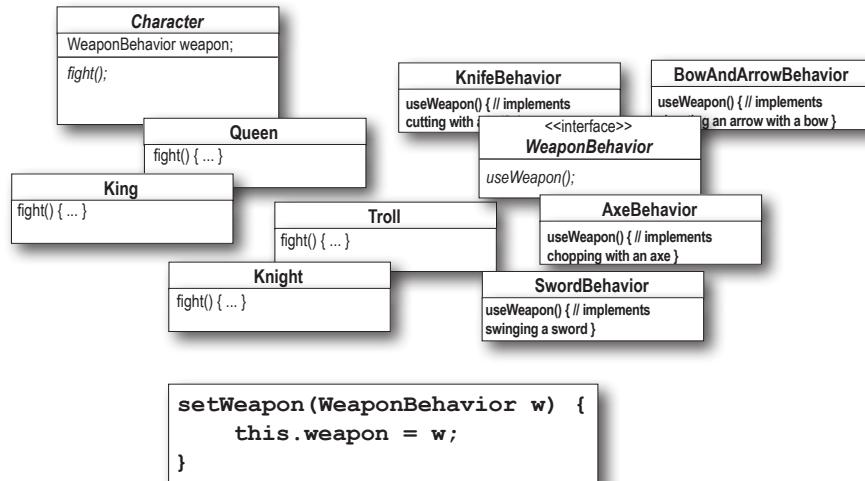
Design Puzzle

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

(Answers are at the end of the chapter.)

Your task:

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



Overheard at the local diner...

Alice

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

Flo

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



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

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

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

Overheard in the next cubicle...

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



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



BRAIN POWER

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

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

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

The power of a shared pattern vocabulary

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

Shared pattern vocabularies are POWERFUL.

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

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

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

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

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

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

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

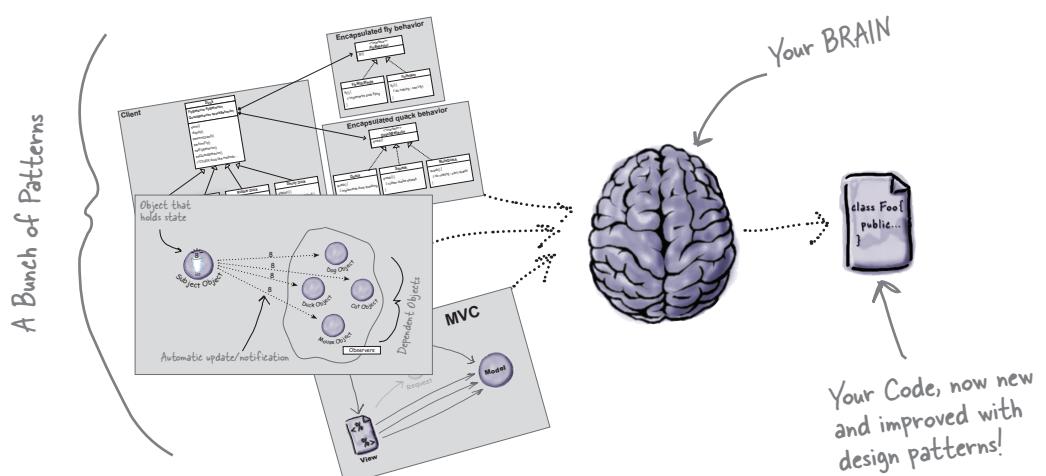
Shared vocabularies encourage more junior developers to get up to speed. Junior developers look up to experienced developers. When senior developers make use of design patterns, junior developers also become motivated to learn them. Build a community of pattern users at your organization.

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

How do I use Design Patterns?

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

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



there are no
Dumb Questions

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

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

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

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

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

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

why design patterns?



Skeptical Developer



Friendly Patterns Guru

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

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

Developer: No?

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

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

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

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

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

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

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



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

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

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



Tools for your Design Toolbox

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

OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

OO Principles

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

OO Patterns

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

One down, many to go!

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

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

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

BULLET POINTS

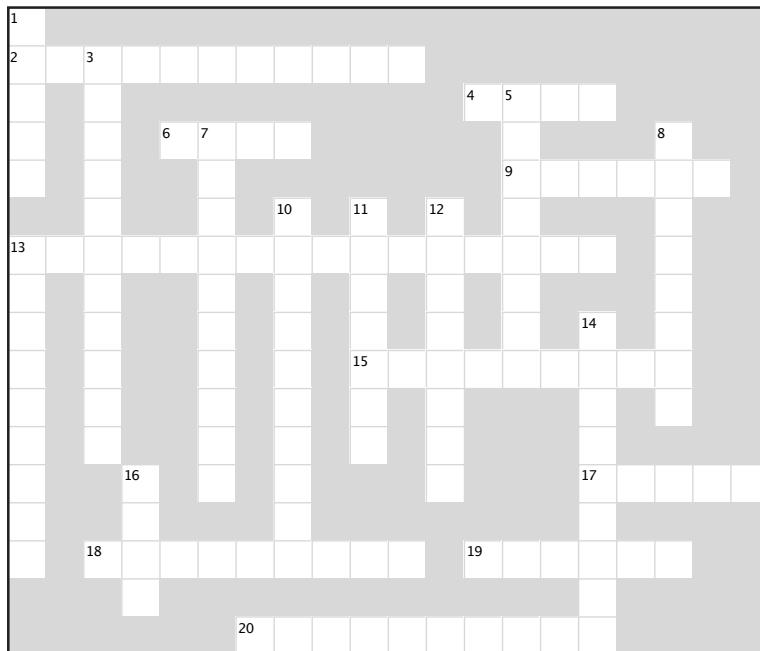
- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible, and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions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



Design Patterns Crossword

Let's give your right brain something to do.

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



ACROSS

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

DOWN

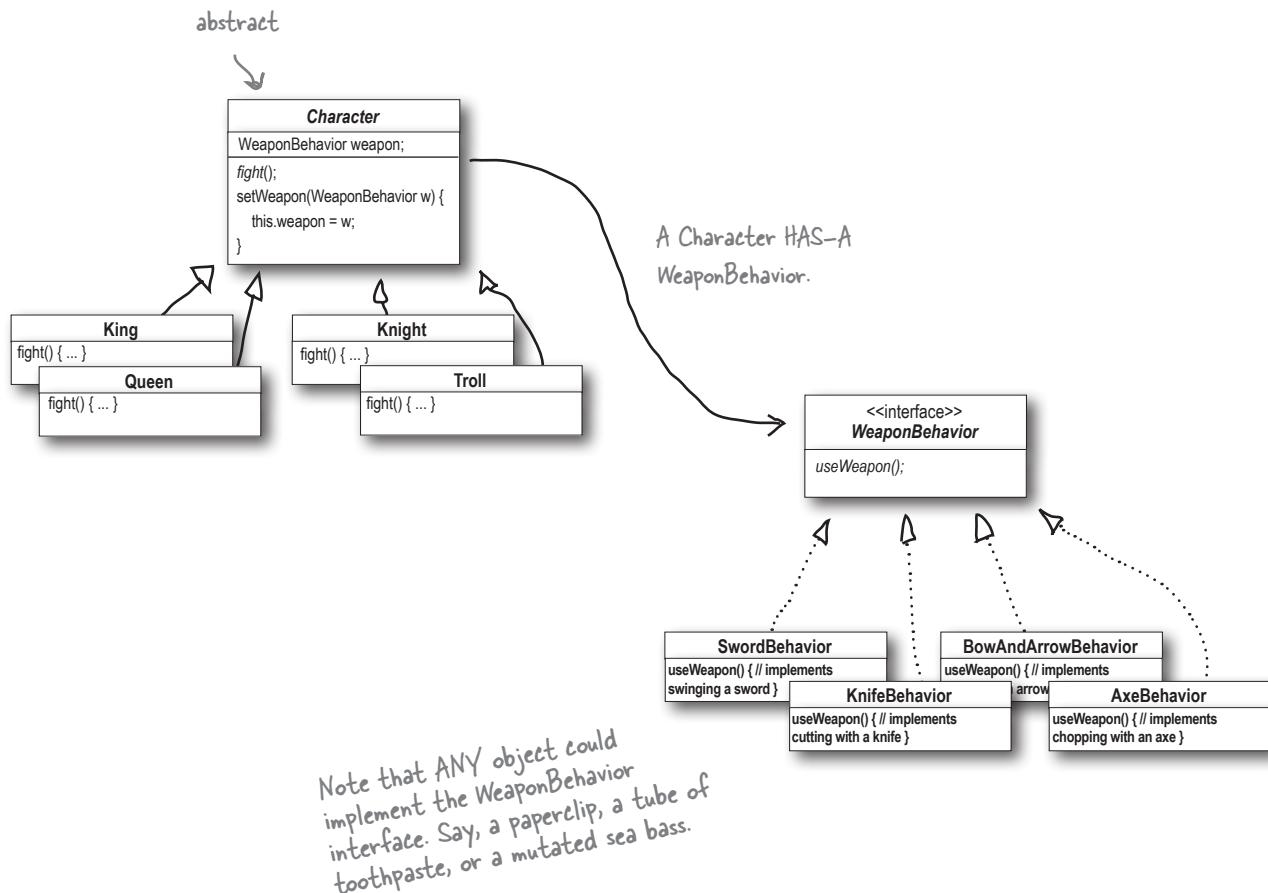
1. Patterns _____ in many applications.
3. Favor this over inheritance.
5. Dan was thrilled with this pattern.
7. Most patterns follow from OO _____.
8. Not your own _____.
10. High level libraries.
11. Joe's favorite drink.
12. Pattern that fixed the simulator.
13. Duck that can't quack.
14. Grilled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



Design Puzzle Solution

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

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





Sharpen your pencil Solution

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

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



Sharpen your pencil Solution

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

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

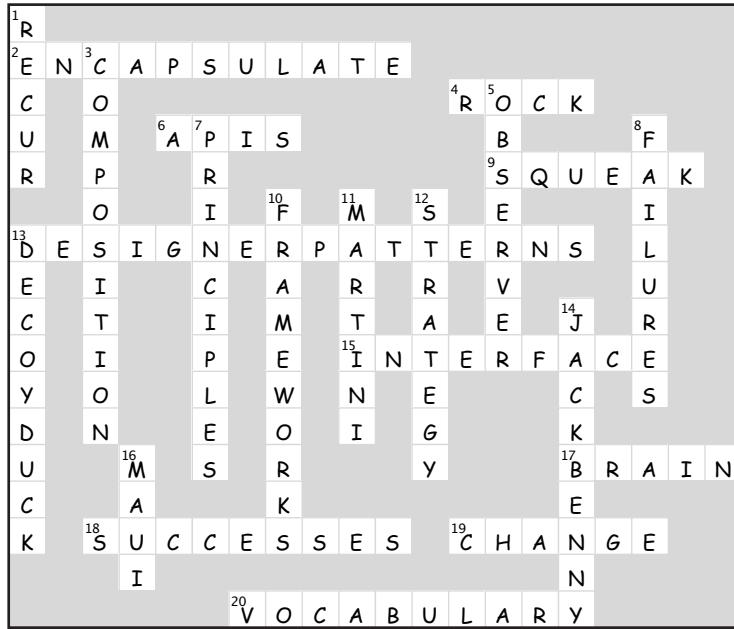
My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

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

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



Design Patterns Crossword Solution



2 the Observer Pattern

Keeping your Objects in the know



Hey Jerry, I'm notifying everyone that the Patterns Group meeting moved to Saturday night. We're going to be talking about the Observer Pattern. That pattern is the best! It's the BEST, Jerry!

Don't miss out when something interesting happens!

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

Congratulations!

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



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

Statement of Work

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

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

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

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

We look forward to seeing your design and alpha application.

Sincerely,

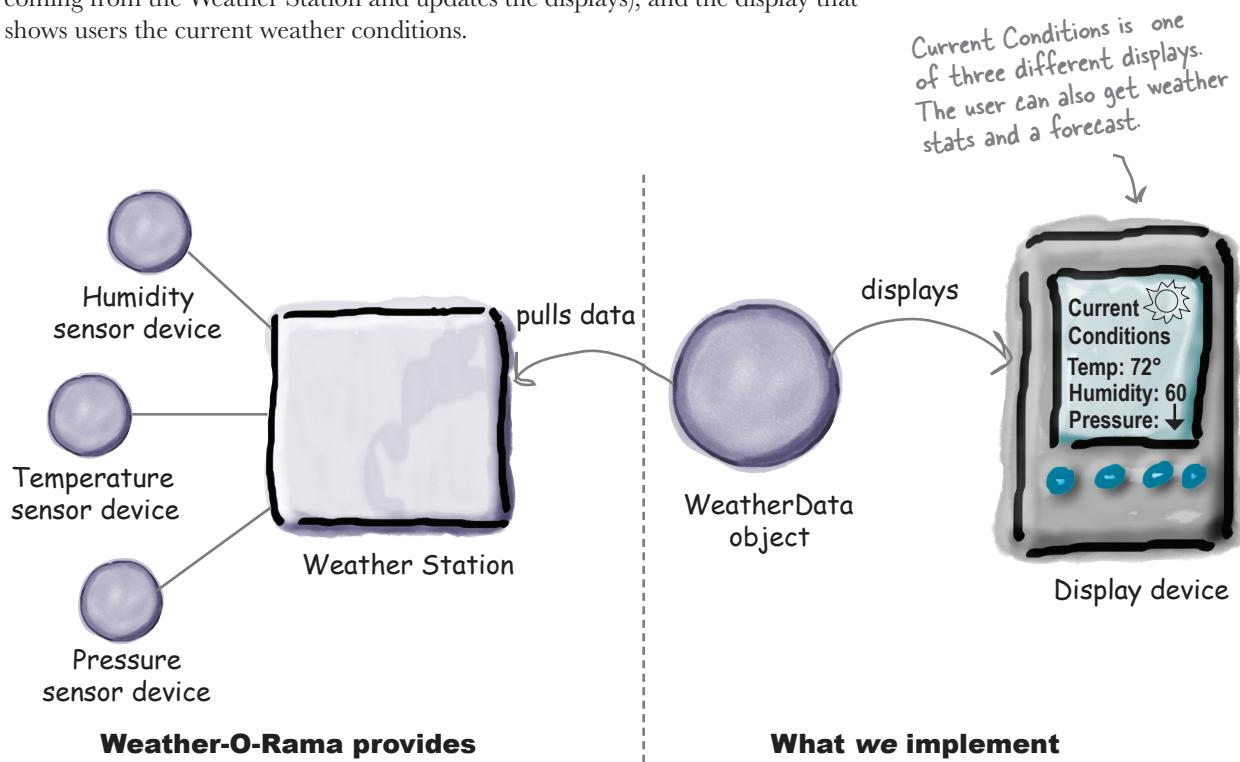
Johnny Hurricane

Johnny Hurricane, CEO

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

The Weather Monitoring application overview

The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.

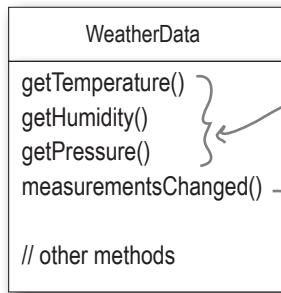


The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast.

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

Unpacking the WeatherData class

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



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

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

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

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



Display device

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

WeatherData.java

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

What do we know so far?

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

- The WeatherData class has getter methods for three measurement values: temperature, humidity, and barometric pressure.
- The measurementsChanged() method is called any time new weather measurement data is available. (We don't know or care how this method is called; we just know that it *is*.)
- We need to implement three display elements that use the weather data: a *current conditions* display, a *statistics display*, and a *forecast* display. These displays must be updated each time WeatherData has new measurements.
- The system must be expandable—other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial *three* display types (current conditions, statistics, and forecast).

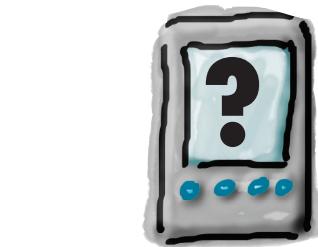
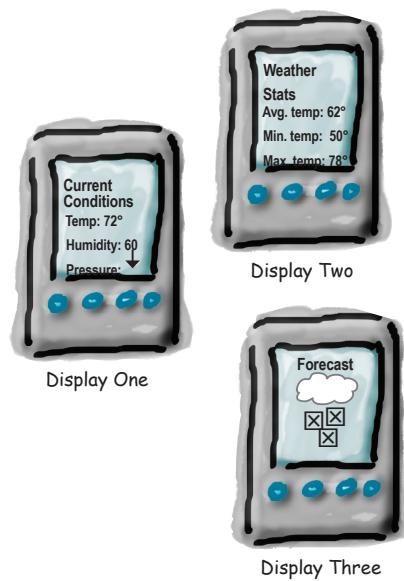


`getTemperature()`

`getHumidity()`

`getPressure()`

`measurementsChanged()`



Future displays

Taking a first, misguided SWAG at the Weather Station

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

```
public class WeatherData {  
  
    // instance variable declarations  
  
    public void measurementsChanged() {  
  
        float temp = getTemperature(); }  
        float humidity = getHumidity(); }  
        float pressure = getPressure(); }  
  
        currentConditionsDisplay.update(temp, humidity, pressure); }  
        statisticsDisplay.update(temp, humidity, pressure); }  
        forecastDisplay.update(temp, humidity, pressure); }  
    }  
  
    // other WeatherData methods here  
}
```

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

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.



Sharpen your pencil

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

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

What's wrong with our implementation?

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

```
public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

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

Area of change. We need to encapsulate this.

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

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



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

Meet the Observer Pattern

You know how newspaper or magazine subscriptions work:

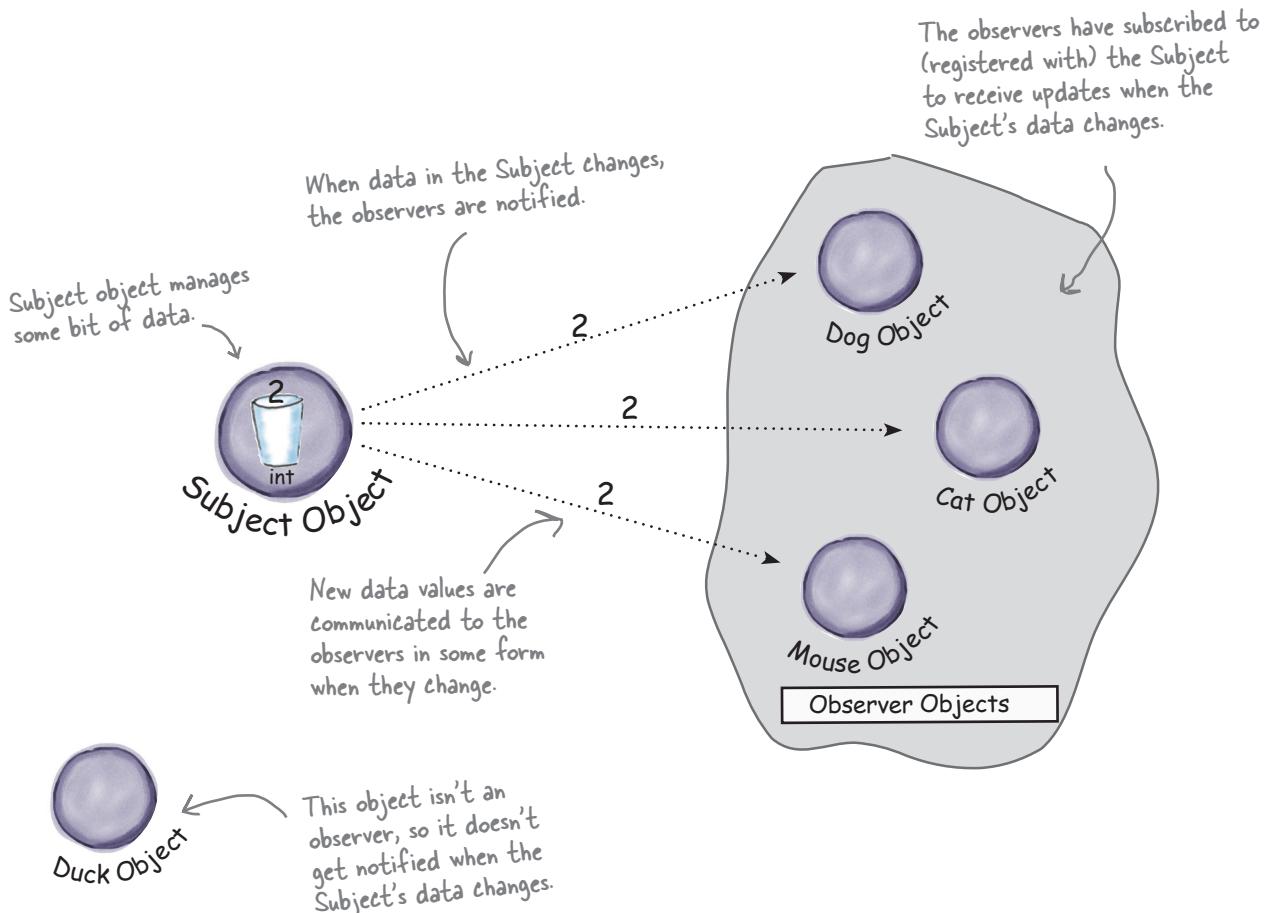
- ❶ A newspaper publisher goes into business and begins publishing newspapers.
- ❷ You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- ❸ You unsubscribe when you don't want papers anymore, and they stop being delivered.
- ❹ While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.



Publishers + Subscribers = Observer Pattern

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

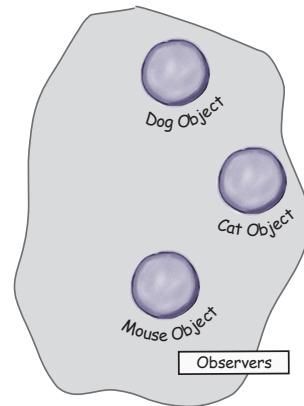
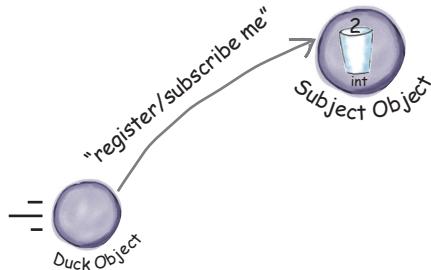
Let's take a closer look:



A day in the life of the Observer Pattern

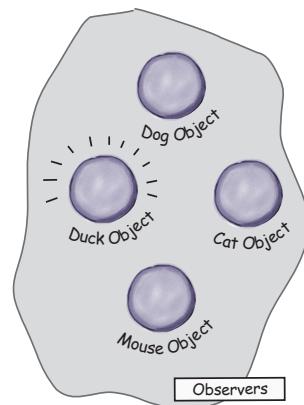
A Duck object comes along and tells the Subject that it wants to become an observer.

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



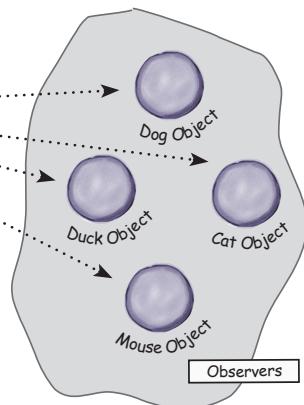
The Duck object is now an official observer.

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



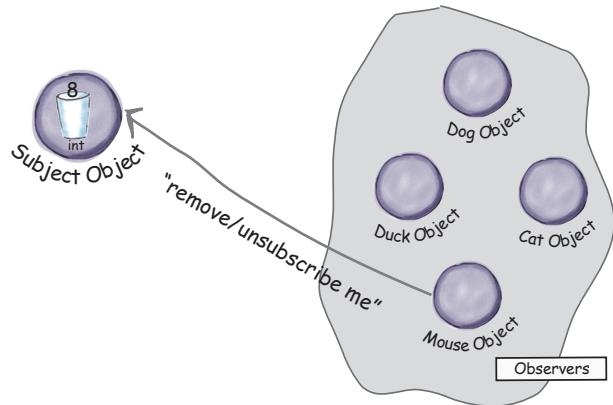
The Subject gets a new data value!

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



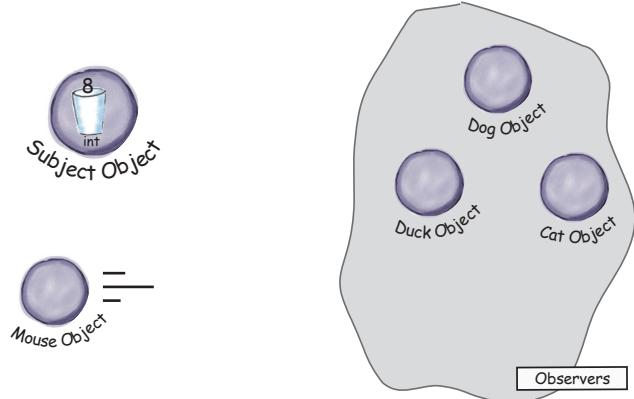
The Mouse object asks to be removed as an observer.

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



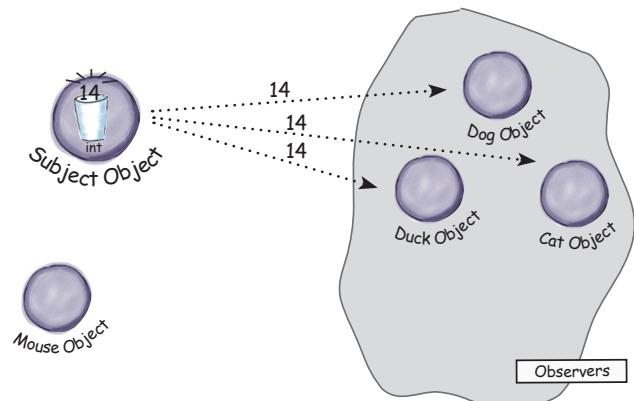
Mouse is outta here!

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



The Subject has another new int.

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





Five-minute drama: a subject for observation

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

This is Lori. I'm looking for a Java development position. I've got five years of experience and...



1

Uh, yeah, you and everybody else, baby. I'm putting you on my list of Java developers. Don't call me, I'll call you!



2

Headhunter/Subject

Software Developer #1

Hi, I'm Jill. I've written a lot of EJB systems. I'm interested in any job you've got with Java development.



3

Software Developer #2

I'll add you to the list - you'll know along with everyone else.



4

Subject

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



6
Subject

Jill lands her own job!



8
Observer



7
Observer

Observer



9
Subject

Two weeks later...



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

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



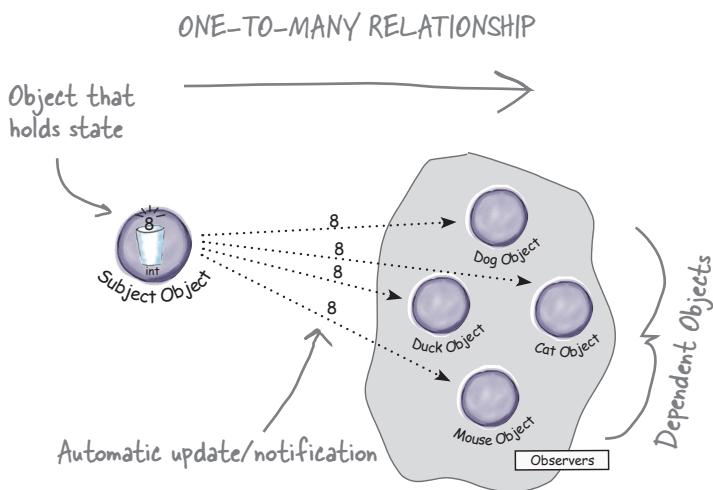
The Observer Pattern defined

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

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

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

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



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

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

Let's take a look...

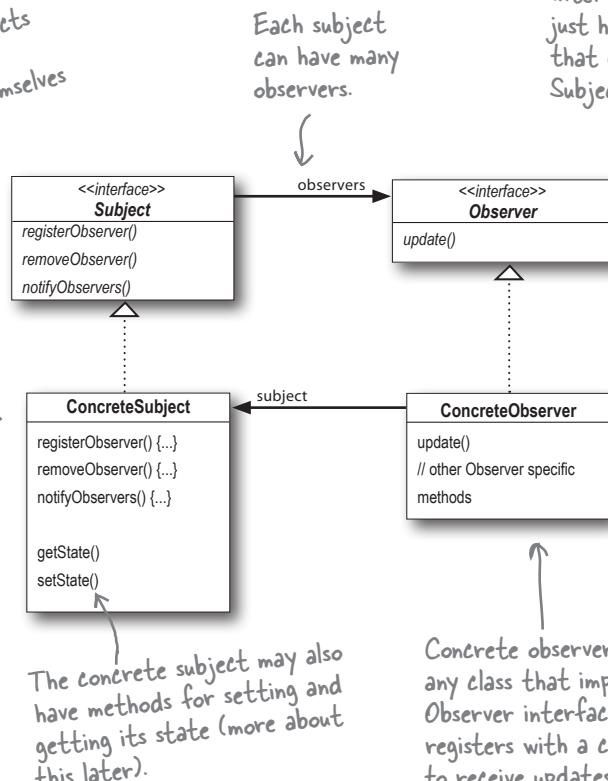
The Observer Pattern defines a one-to-many relationship between a set of objects.

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

The Observer Pattern defined: the class diagram

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

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



Each subject can have many observers.

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

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

there are no
Dumb Questions

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

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

Q: How does dependence come into this?

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

The power of Loose Coupling

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

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

Why?

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

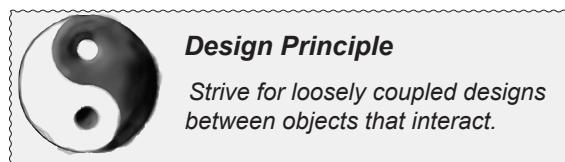
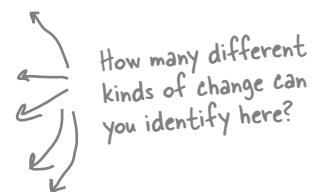
We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise,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

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

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

Changes to either the subject or an observer will not affect the other.

Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.



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



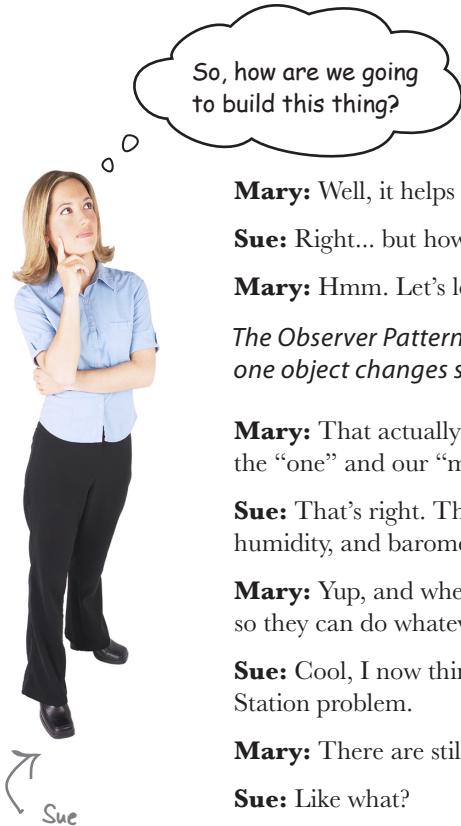
Sharpen your pencil

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

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

Cubicle conversation

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



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

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

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

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

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

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

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

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

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

Sue: Like what?

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

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

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

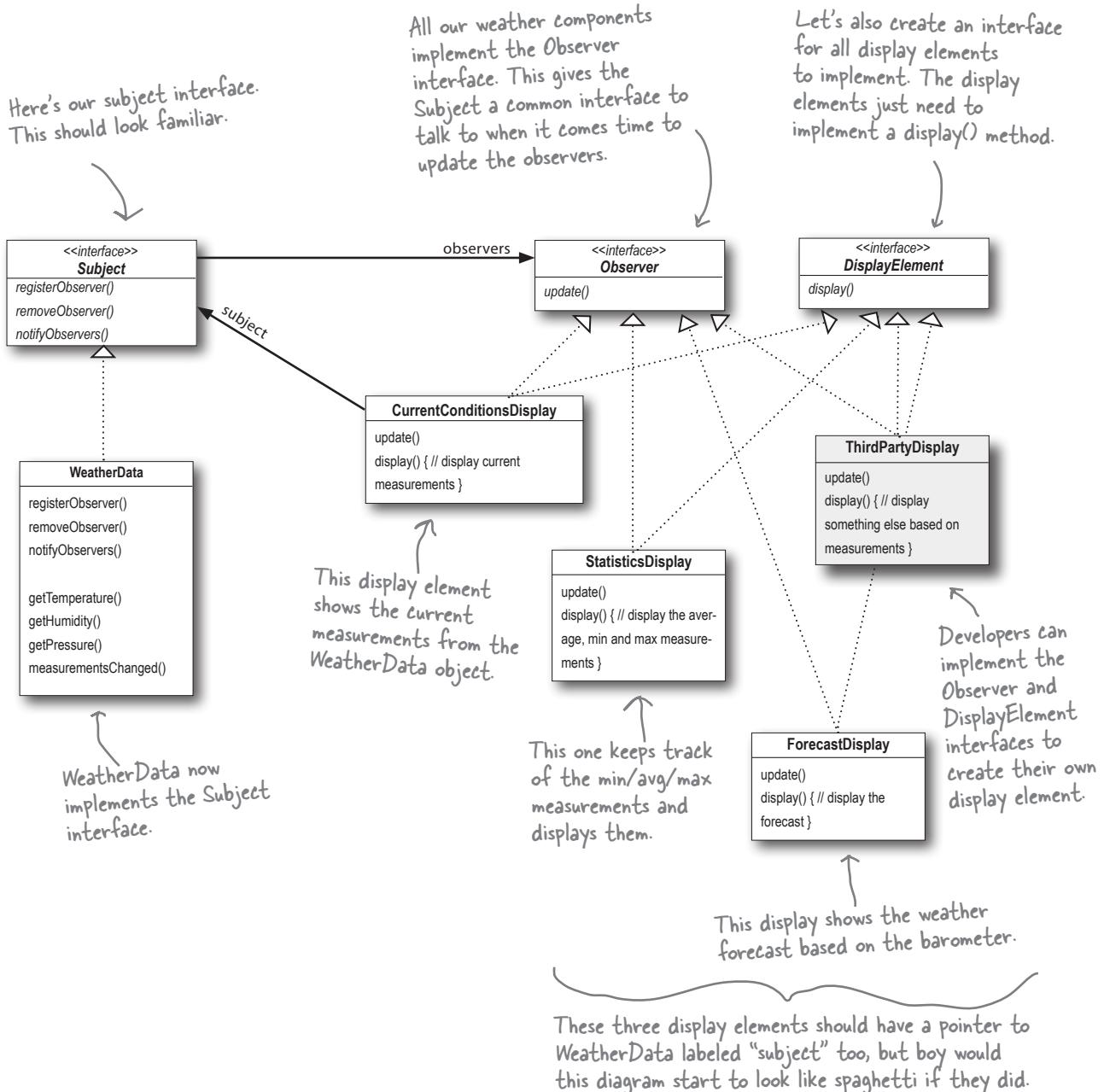
Sue: We gotta remember that every display element can be different... so I think that's where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

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

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

Designing the Weather Station

How does this diagram compare with yours?



Implementing the Weather Station

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

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

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

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

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

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

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

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

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



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

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

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

Implementing the Subject interface in WeatherData

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

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

Here we implement the Subject interface.

```

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

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

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

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

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

    public void measurementsChanged() {
        notifyObservers();
    }

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

    // other WeatherData methods here
}

```

- A brace on the left side of the code block is labeled "Here we implement the Subject interface."
- An annotation above the "observers" field says: "WeatherData now implements the Subject interface." with an arrow pointing to the "implements Subject" part of the class declaration.
- An annotation next to the constructor says: "We've added an ArrayList to hold the Observers, and we create it in the constructor." with an arrow pointing to the constructor assignment.
- An annotation next to the "registerObserver" method says: "When an observer registers, we just add it to the end of the list." with an arrow pointing to the "observers.add(o);" line.
- An annotation next to the "removeObserver" method says: "Likewise, when an observer wants to unregister, we just take it off the list." with an arrow pointing to the "observers.remove(i);" line.
- An annotation next to the "notifyObservers" method says: "Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them." with an arrow pointing to the loop body.
- An annotation next to the "measurementsChanged" method says: "We notify the Observers when we get updated measurements from the Weather Station." with an arrow pointing to the call to "notifyObservers".
- An annotation next to the "setMeasurements" method says: "Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the Web." with an arrow pointing to the method body.

Now, let's build those display elements

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

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

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

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

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

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

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

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

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

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

there are no
Dumb Questions

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

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

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

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

Power up the Weather Station



① First, let's create a test harness.

The Weather Station is ready to go. All we need is some code to glue everything together. Here's our first attempt. We'll come back later in the book and make sure all the components are easily pluggable via a configuration file. For now here's how it all works:

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

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

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

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

First, create the WeatherData object.

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

Simulate new weather measurements.

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

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



Sharpen your pencil

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

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

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

So get typing!

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

You can get heatindex.txt from wickedlysmart.com.

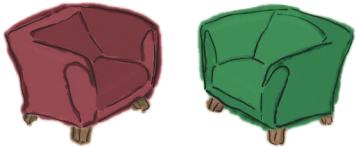
How does it work? You'd have to refer to *Head First Meteorology*, or try asking someone at the National Weather Service (or try a web search).

When you finish, your output should look like this:

Here's what changed in this output.

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

Fireside Chats



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

Subject:

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

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

Oh yeah, like what?

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

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

Observer:

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

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

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

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

Subject:

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

Observer:

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

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

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

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

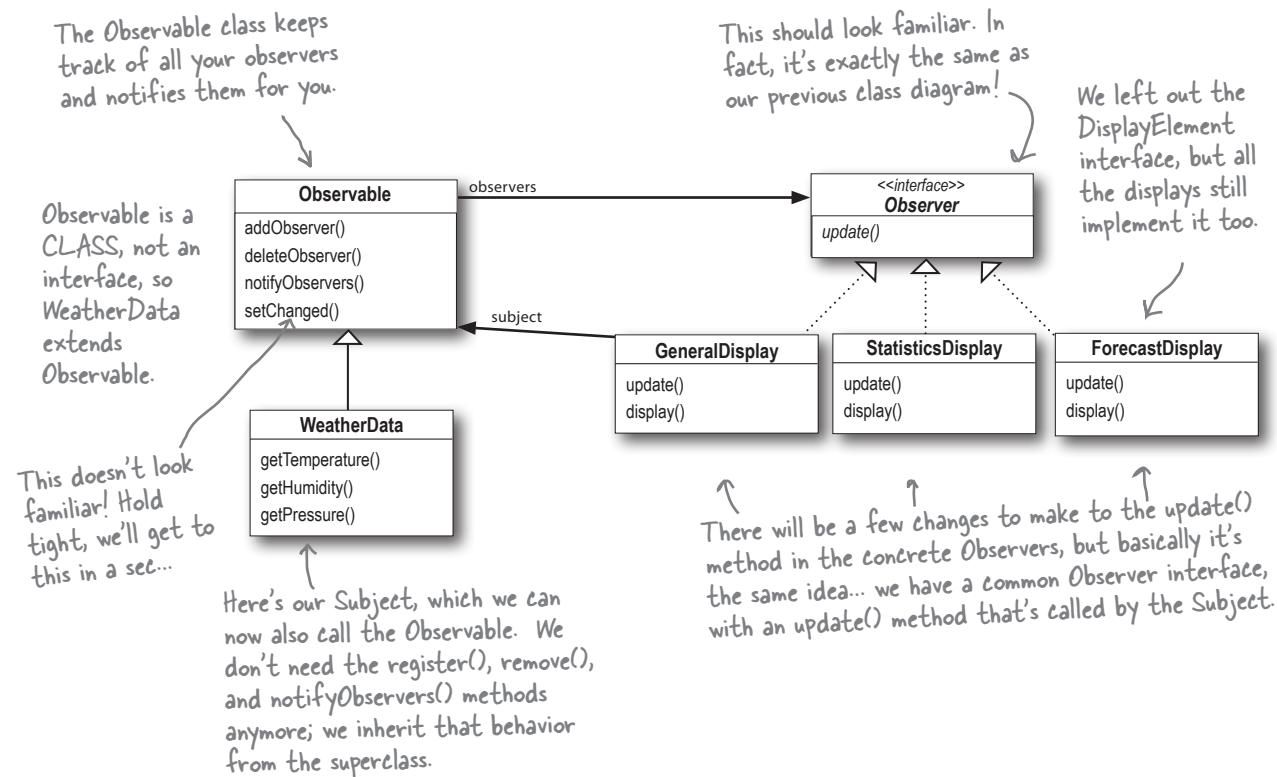
Oh really? I think we're going to look at that next....

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

Using Java's built-in Observer Pattern

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

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



How Java's built-in Observer Pattern works

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

For an Object to become an observer...

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

For the Observable to send notifications...

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

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

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

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

For an Observer to receive notifications...

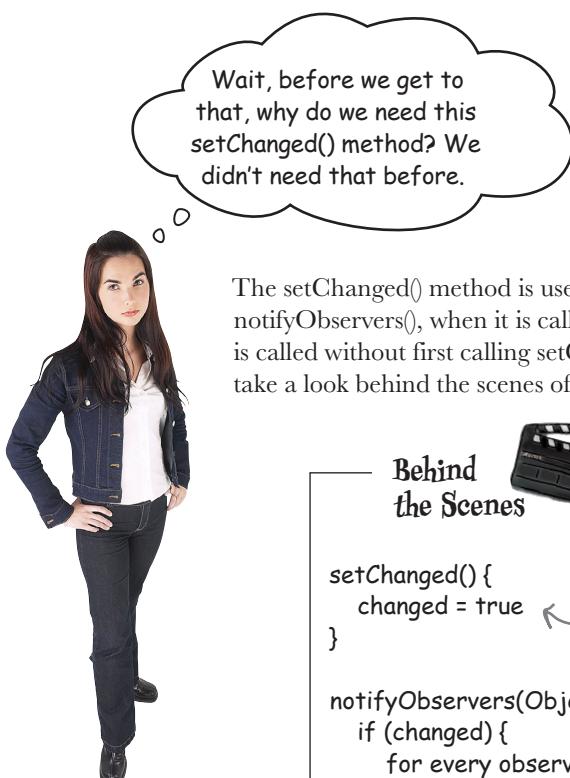
It implements the `update` method, as before, but the signature of the method is a bit different:

`update(Observable o, Object arg)`

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

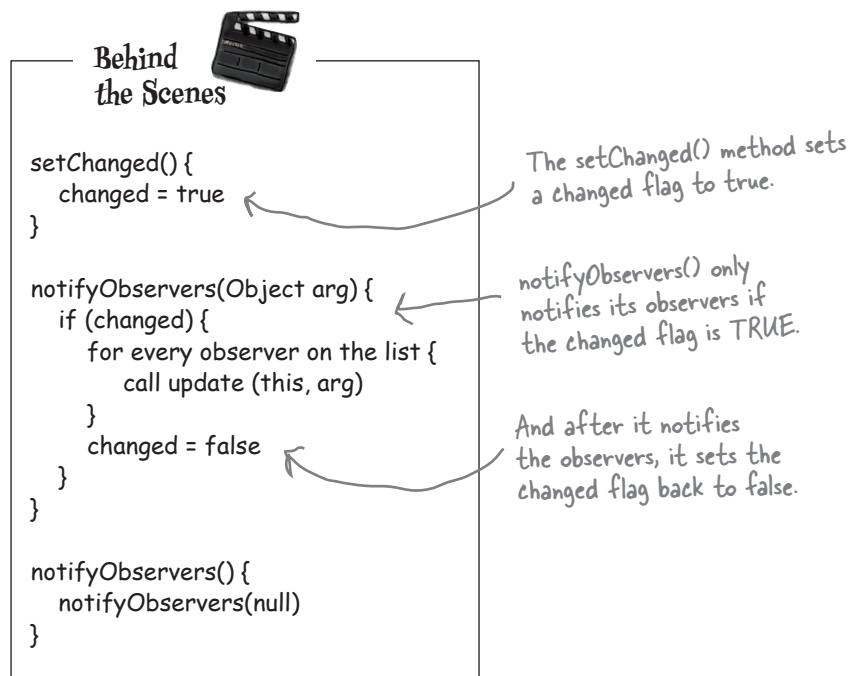
This will be the data object that was passed to `notifyObservers()`, or null if a data object wasn't specified.

If you want to “push” data to the observers, you can pass the data as a `data object` to the `notifyObservers(arg)` method. If not, then the `Observer` has to “pull” the data it wants from the `Observable` object passed to it. How? Let's rework the Weather Station and you'll see.



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

Pseudocode for the Observable class.



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

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

Reworking the Weather Station with the built-in support

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

```

import java.util.Observable;

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

    public WeatherData() { }

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

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

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}

```

1 Make sure we are importing the right Observable.

2 We are now subclassing Observable.

3 We don't need to keep track of our observers anymore, or manage their registration and removal (the superclass will handle that), so we've removed the `registerObserver()`, `removeObserver()` and `notifyObservers()` methods.

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

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

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

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

Now, let's rework the CurrentConditionsDisplay

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

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

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

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

```
    Observable observable;
```

```
    private float temperature;
```

```
    private float humidity;
```

```
    public CurrentConditionsDisplay(Observable observable) {
```

```
        this.observable = observable;
```

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

```
}
```

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

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

```
        if (obs instanceof WeatherData) {
```

```
            WeatherData weatherData = (WeatherData) obs;
```

```
            this.temperature = weatherData.getTemperature();
```

```
            this.humidity = weatherData.getHumidity();
```

```
            display();
```

```
}
```

```
}
```

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

```
    public void display() {
```

```
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
```

```
}
```

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



Code Magnets

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

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

```
    display();
```

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

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

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

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

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

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

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

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

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

Running the new code

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

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

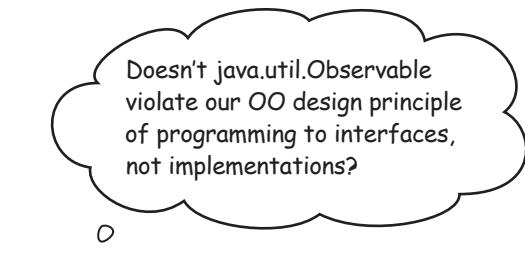
Hmm, do you notice anything different? Look again...

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

Never depend on order of evaluation of the Observer notifications

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

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



The dark side of `java.util.Observable`

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

Observable is a class

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

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

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

Observable protects crucial methods

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

What to do?

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

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

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

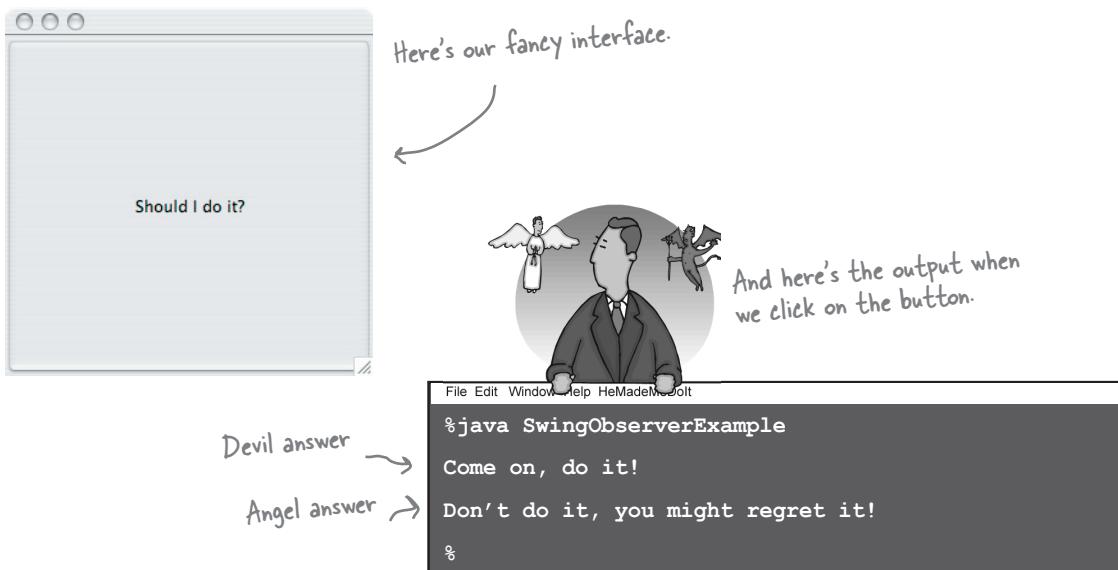
If you're curious about the Observer Pattern in JavaBeans, check out the `PropertyChangeListener` interface.

A little background...

Let's take a look at a simple part of the Swing API, the JButton. If you look under the hood at JButton's superclass, AbstractButton, you'll see that it has a lot of add/remove listener methods. These methods allow you to add and remove observers, or, as they are called in Swing, listeners, to listen for various types of events that occur on the Swing component. For instance, an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press. You'll find various types of listeners all over the Swing API.

A little life-changing application

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



And the code...

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

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

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

        // Set frame properties here
    }

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

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

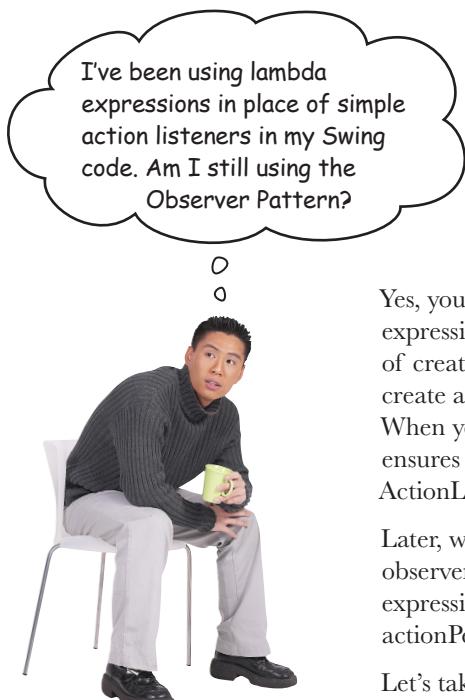
Simple Swing application that just creates a frame and throws a button in it.

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

Code to set up the frame goes here.

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

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



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

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

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

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

The updated code, using lambda expressions:

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();
        JButton button = new JButton("Should I do it?");
        button.addActionListener(event ->
            System.out.println("Don't do it, you might regret it!"));
        button.addActionListener(event ->
            System.out.println("Come on, do it!"));
        // Set frame properties here
    }
}
```

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

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

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

Using lambda expressions makes this code a lot more concise.

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



Tools for your Design Toolbox

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

OO Basics

- Abstraction
- Inheritance
- Encapsulation
- Polymorphism
- Interface

OO Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.

OO Patterns

- Strategy
- Encapsulation
- Interception
- Variability

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

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



BULLET POINTS

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



Design Principle Challenge

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

Design Principle

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

Design Principle

Program to an interface, not an implementation.

Design Principle

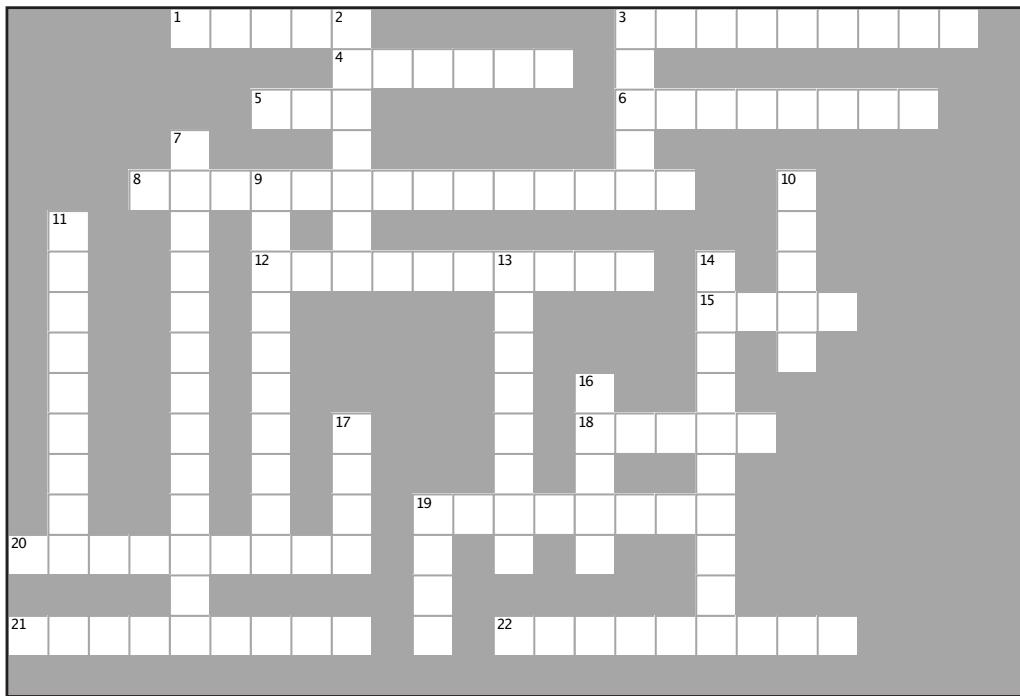
Favor composition over inheritance.

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



Design Patterns Crossword

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



ACROSS

1. Observable is a _____, not an interface.
3. Devil and Angel are _____ to the button.
4. Implement this method to get notified.
5. Jill got one of her own.
6. CurrentConditionsDisplay implements this interface.
8. How to get yourself off the Observer list.
12. You forgot this if you're not getting notified when you think you should be.
15. One Subject likes to talk to _____ observers.
18. Don't count on this for notification.
19. Temperature, humidity and _____.
20. Observers are _____ on the Subject.
21. Program to an _____ not an implementation.
22. A Subject is similar to a _____.

DOWN

2. Ron was both an Observer and a _____.
3. You want to keep your coupling _____.
7. He says you should go for it.
9. _____ can manage your observers for you.
10. Java framework with lots of Observers.
11. Weather-O-Rama's CEO named after this kind of storm.
13. Observers like to be _____ when something new happens.
14. The WeatherData class _____ the Subject interface.
16. He didn't want any more ints, so he removed himself.
17. CEO almost forgot the _____ index display
19. Subject initially wanted to _____ all the data to Observer.

Sharpen your pencil Solution

- 
- Based on our first implementation, which of the following apply? (Choose all that apply.)
- A. We are coding to concrete implementations, not interfaces.
 - B. For every new display element we need to alter code.
 - C. We have no way to add display elements at run time.
 - D. The display elements don't implement a common interface.
 - E. We haven't encapsulated what changes.
 - F. We are violating encapsulation of the WeatherData class.



Design Principle Challenge Solution

Design Principle

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

The thing that varies in the Observer Pattern

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

Design Principle

Program to an interface, not an implementation.

Both the Subject and Observer use interfaces.

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

Design Principle

Favor composition over inheritance.

The Observer Pattern uses composition to compose

any number of Observers with their Subjects.

These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!



Code Magnets Solution

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

```

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

public class ForecastDisplay implements
    Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

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

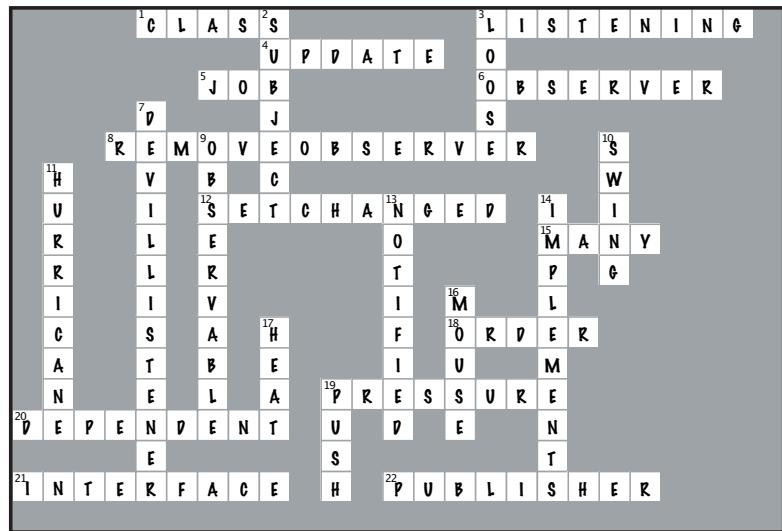
    public void update(Observable observable,
        Object arg) {
        if (observable instanceof WeatherData) {
            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();
            display();
        }
    }

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

```

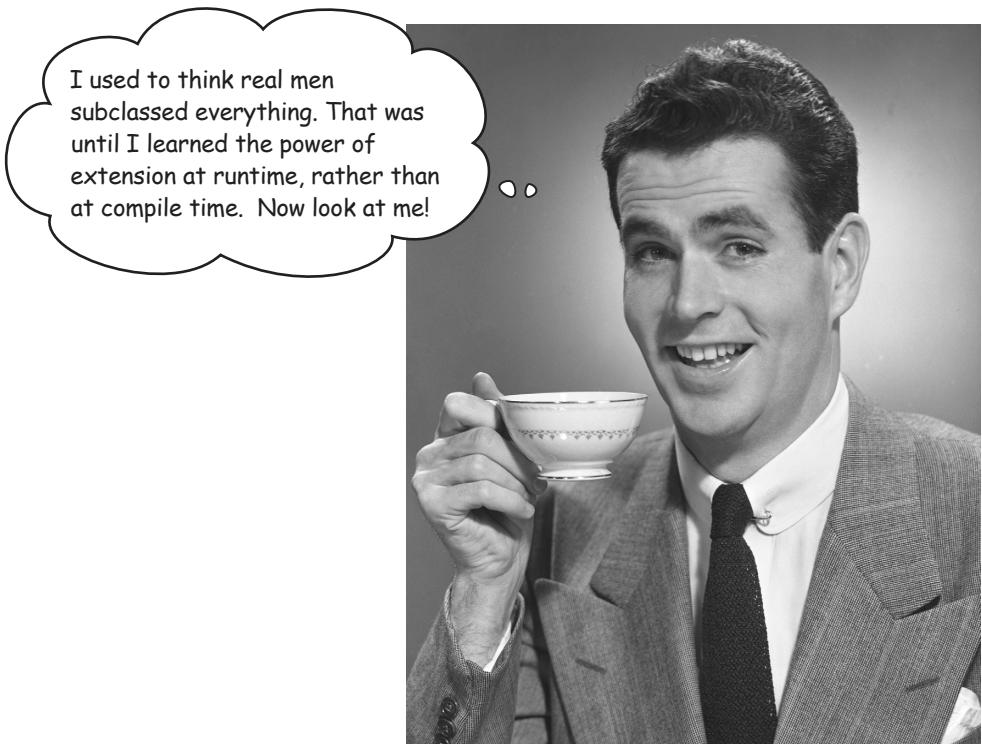


Design Patterns Crossword Solution



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*.

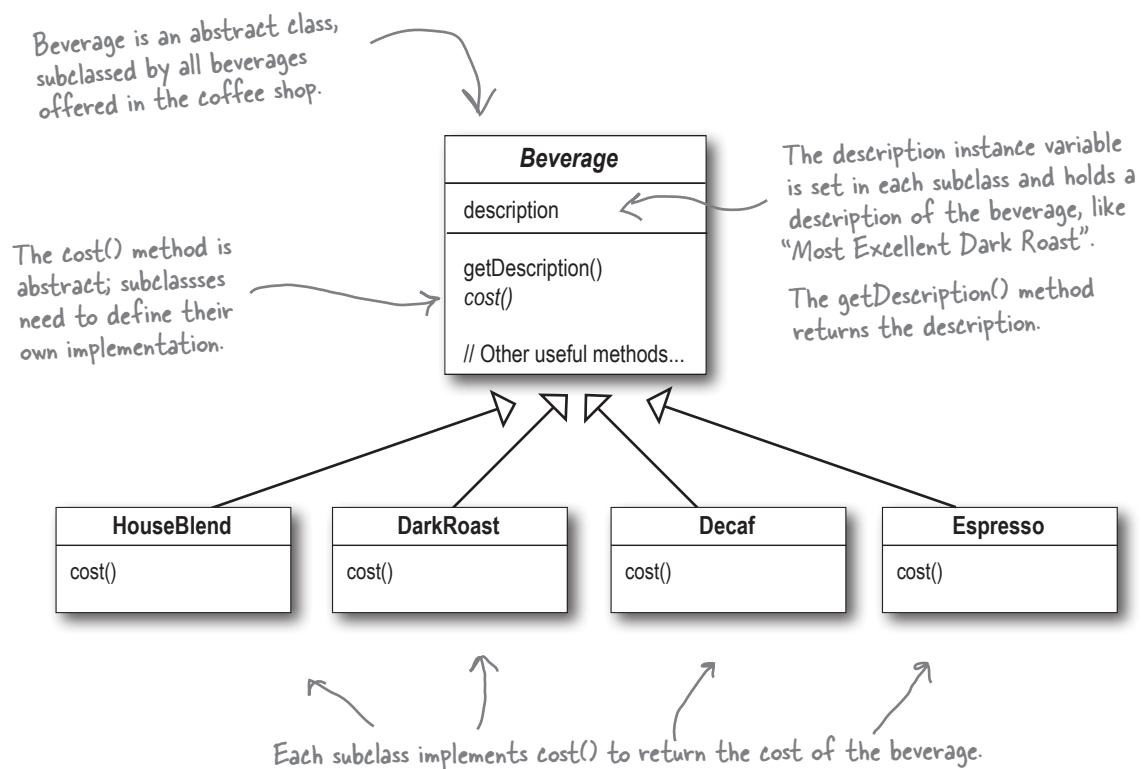
Welcome to Starbuzz Coffee

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.



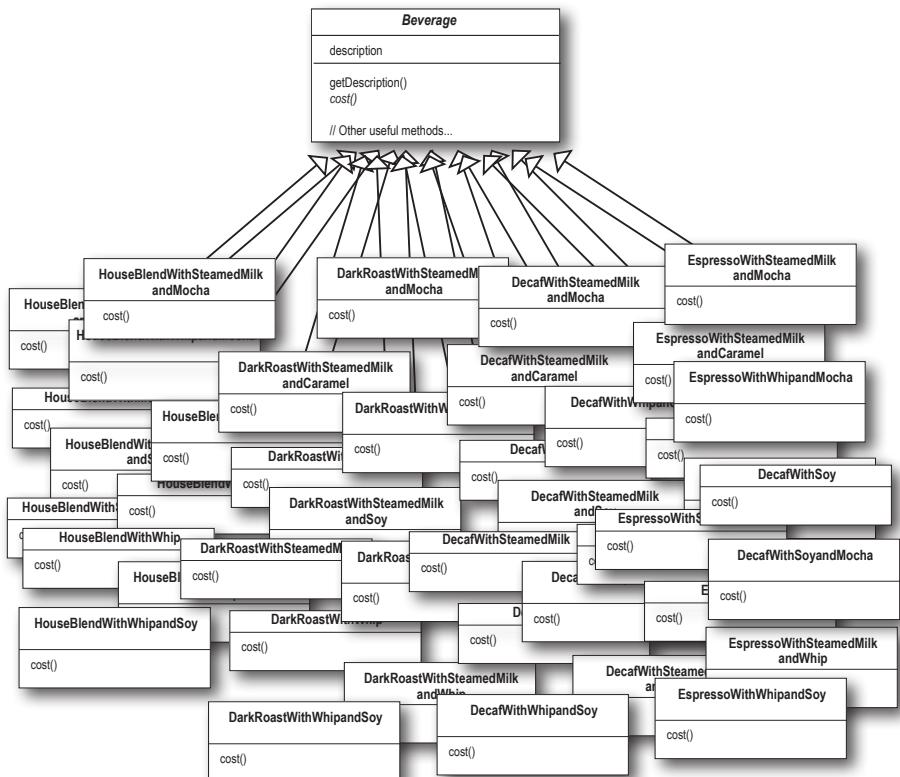
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

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



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...



Whoa!
Can you say
"class explosion"?

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



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?

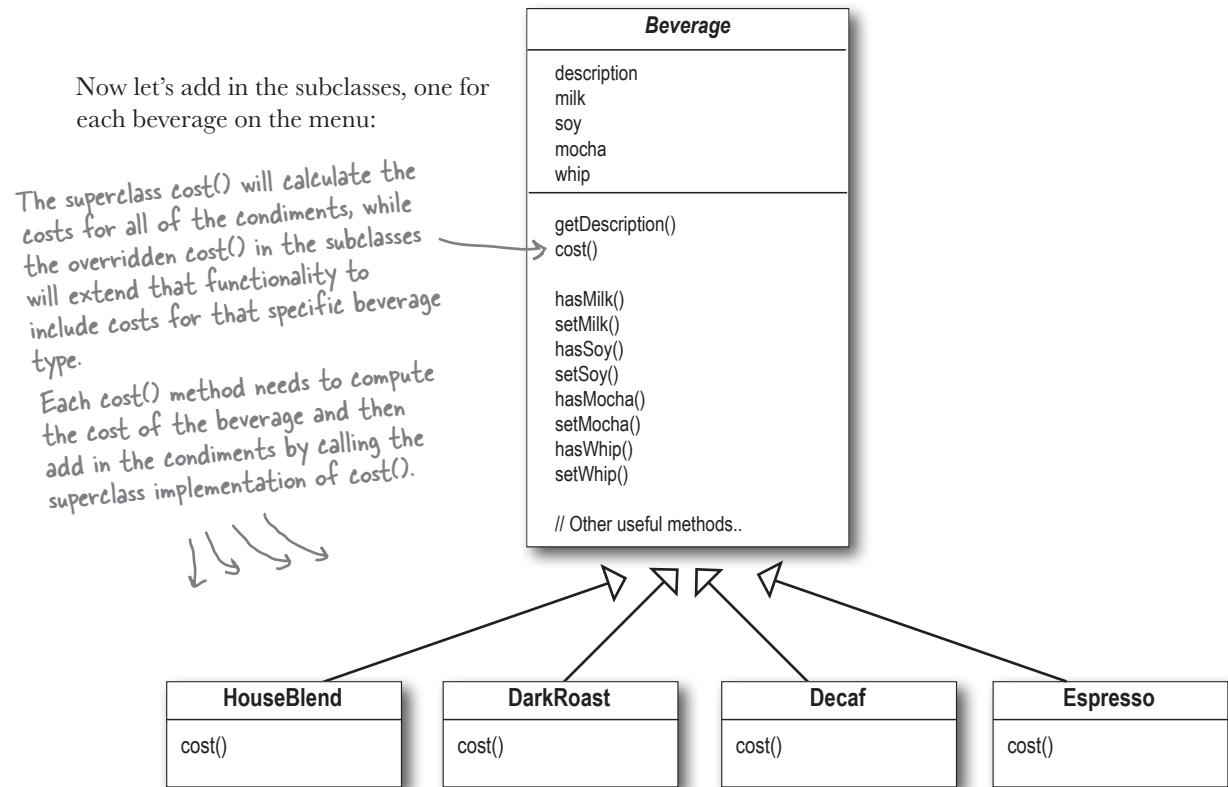
Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha, and whip...

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.



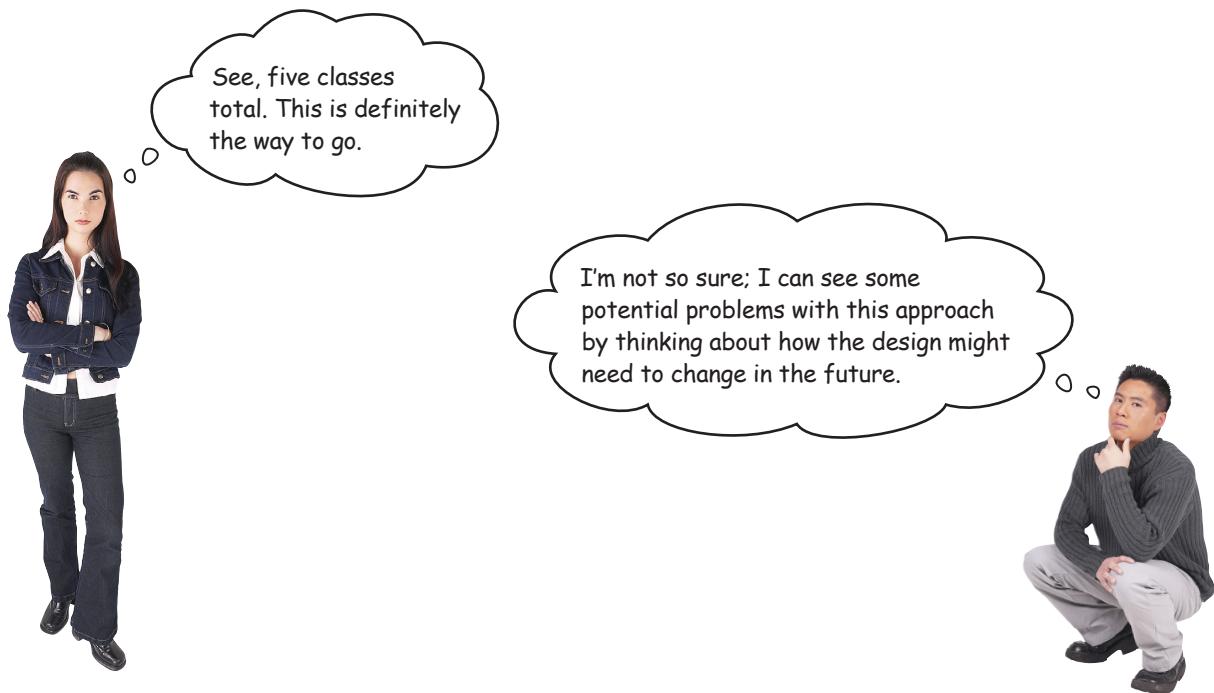
Sharpen your pencil

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

```

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

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



Sharpen your pencil

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

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

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

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

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

What if a customer wants a double mocha?

Your turn:



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!

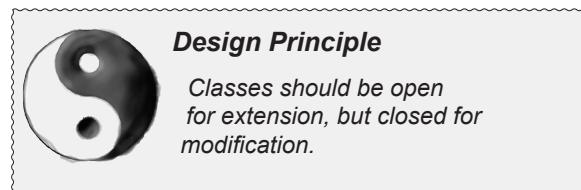
Master: What have you learned about the effect of composition on maintaining your code?

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:



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 knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

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

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

Meet the Decorator Pattern

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

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

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

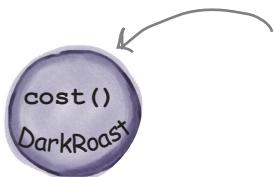
Okay, but how do you "decorate" an object, and how does delegation come into this? A hint: think of decorator objects as "wrappers." Let's see how this works...

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?



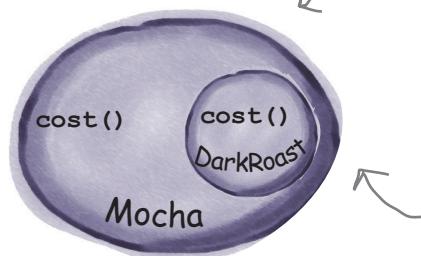
Constructing a drink order with Decorators

- 1 We start with our DarkRoast object.**



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

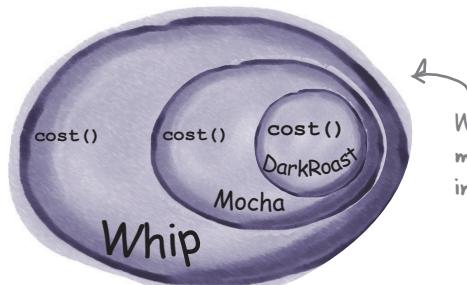
- 2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



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

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

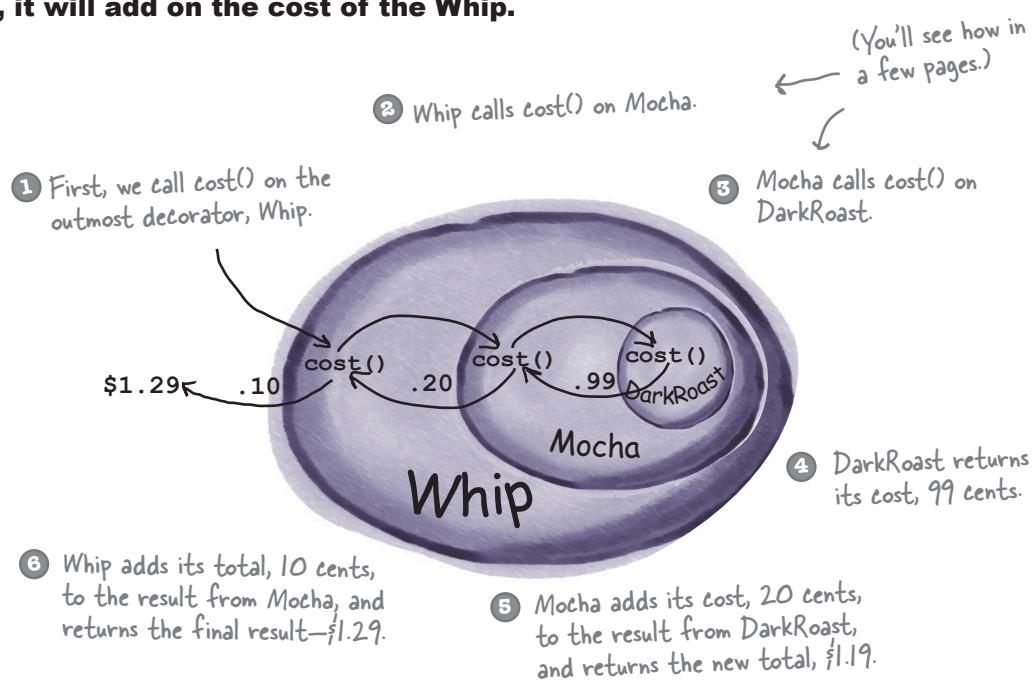
- 3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**



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

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

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



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

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

Key point!

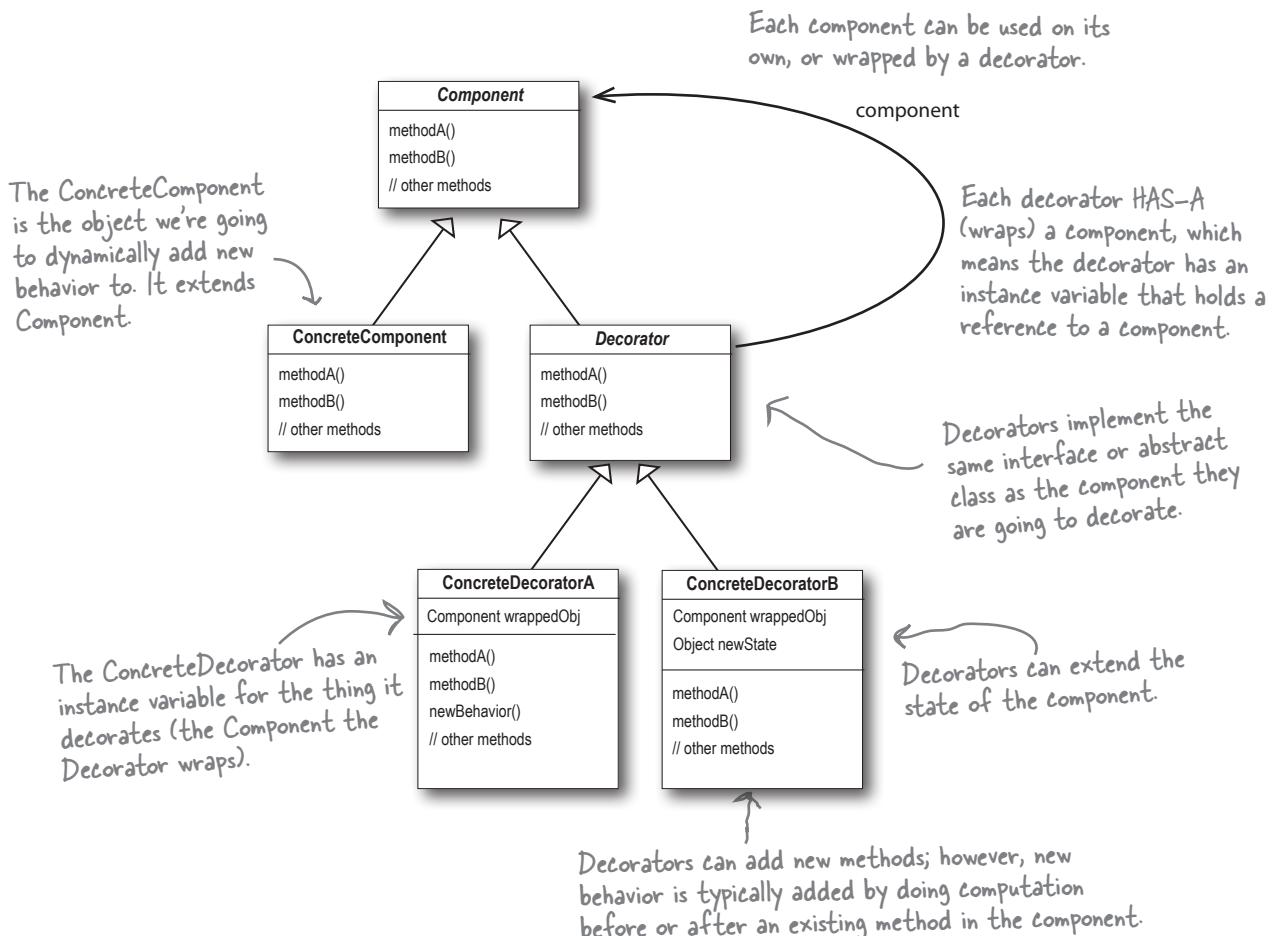
Now let's see how this all really works by looking at the **Decorator Pattern** definition and writing some code.

The Decorator Pattern defined

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

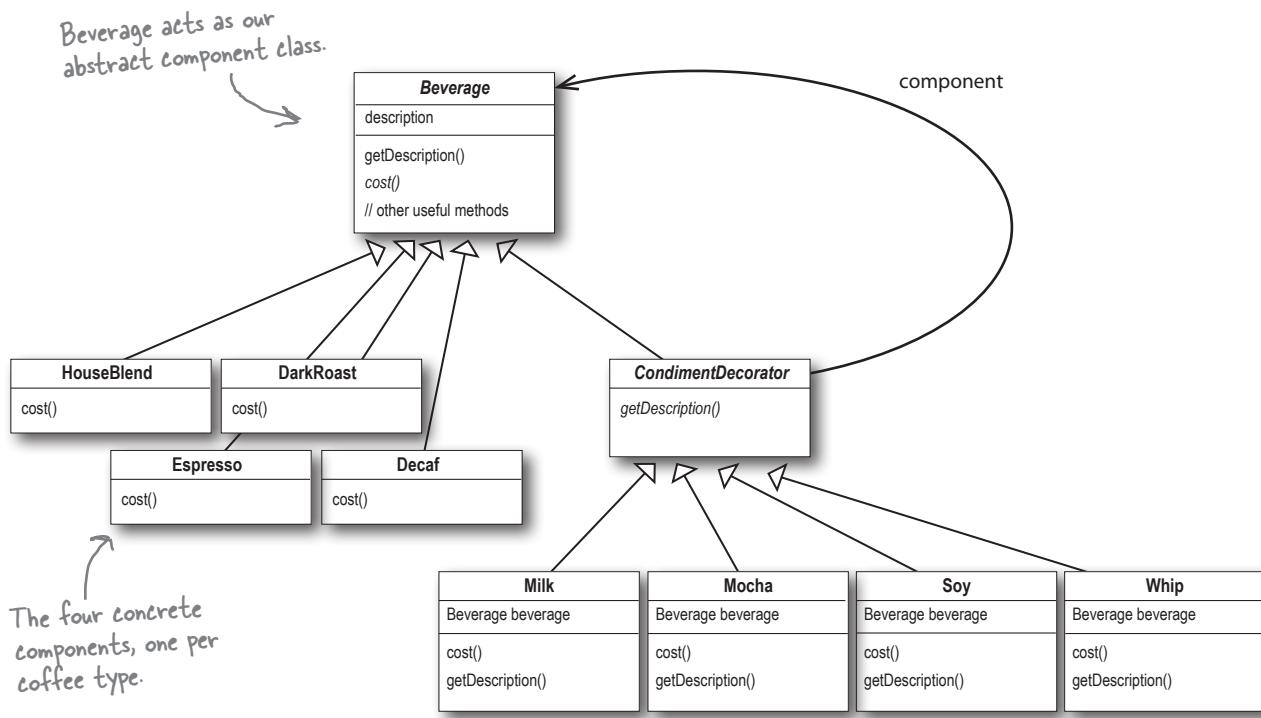
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

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



Decorating our Beverages

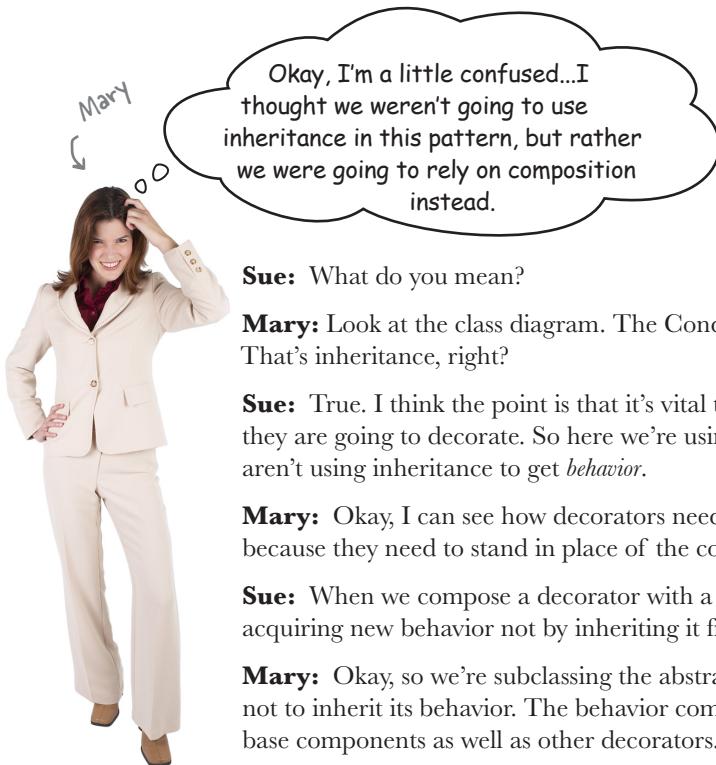
Okay, let's work our Starbuzz beverages into this framework...



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 `get>Description()` method of the condiments.

Cubicle Conversation

Some confusion over Inheritance versus Composition



Sue: What do you mean?

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

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

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

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

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

Sue: That's right.

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

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

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

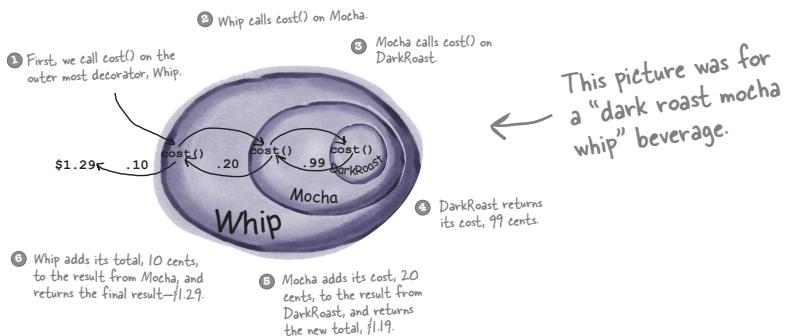
Sue: Exactly.

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

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

New barista training

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



This picture was for
a “dark roast mocha
whip” beverage.



Sharpen your pencil

Draw your picture here.

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



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

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.

```
public class Espresso extends Beverage {
```

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

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

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

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

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

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

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.

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

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

Starbuzz Coffee		
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.

public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

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

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'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.

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...



Exercise

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

Serving some coffees

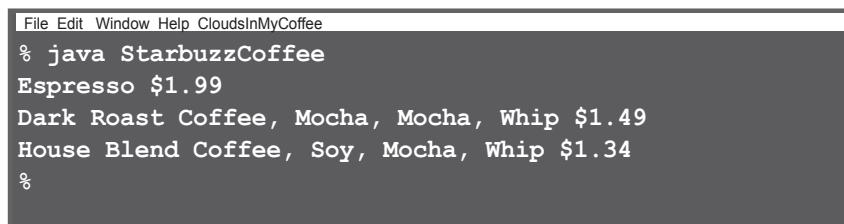
Congratulations. It's time to sit back, order a few coffees, and marvel at the flexible design you created with the Decorator Pattern.

Here's some test code* to make orders:

```
public class StarbuzzCoffee {  
  
    public static void main(String args[]) {  
        Beverage beverage = new Espresso();  
        System.out.println(beverage.getDescription()  
            + " $" + beverage.cost());  
  
        Beverage beverage2 = new DarkRoast(); ← Make a DarkRoast object.  
        beverage2 = new Mocha(beverage2); ← Wrap it with a Mocha.  
        beverage2 = new Mocha(beverage2); ← Wrap it in a second Mocha.  
        beverage2 = new Whip(beverage2); ← Wrap it in a Whip.  
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());  
  
        Beverage beverage3 = new HouseBlend(); ← Finally, give us a HouseBlend  
        beverage3 = new Soy(beverage3); with Soy, Mocha, and Whip.  
        beverage3 = new Mocha(beverage3);  
        beverage3 = new Whip(beverage3);  
  
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());  
    }  
}
```

Now, let's get those orders in:

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



The screenshot shows a terminal window with the following text:
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso \$1.99
Dark Roast Coffee, Mocha, Mocha, Whip \$1.49
House Blend Coffee, Soy, Mocha, Whip \$1.34
%

there are no
Dumb Questions

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

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

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

A: You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, 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 getDescription() method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha." That would require that my outermost decorator know all the decorators it is wrapping.

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



Sharpen your pencil

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

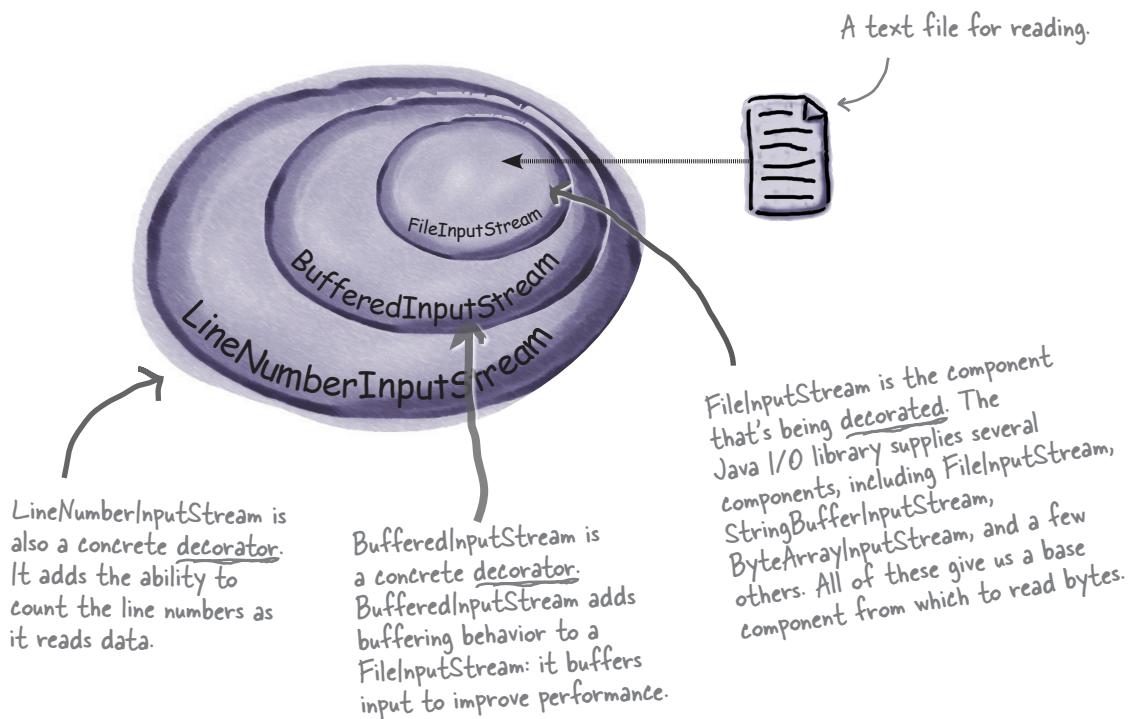
How would you alter the decorator classes to handle this change in requirements?

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

Real World Decorators: Java I/O

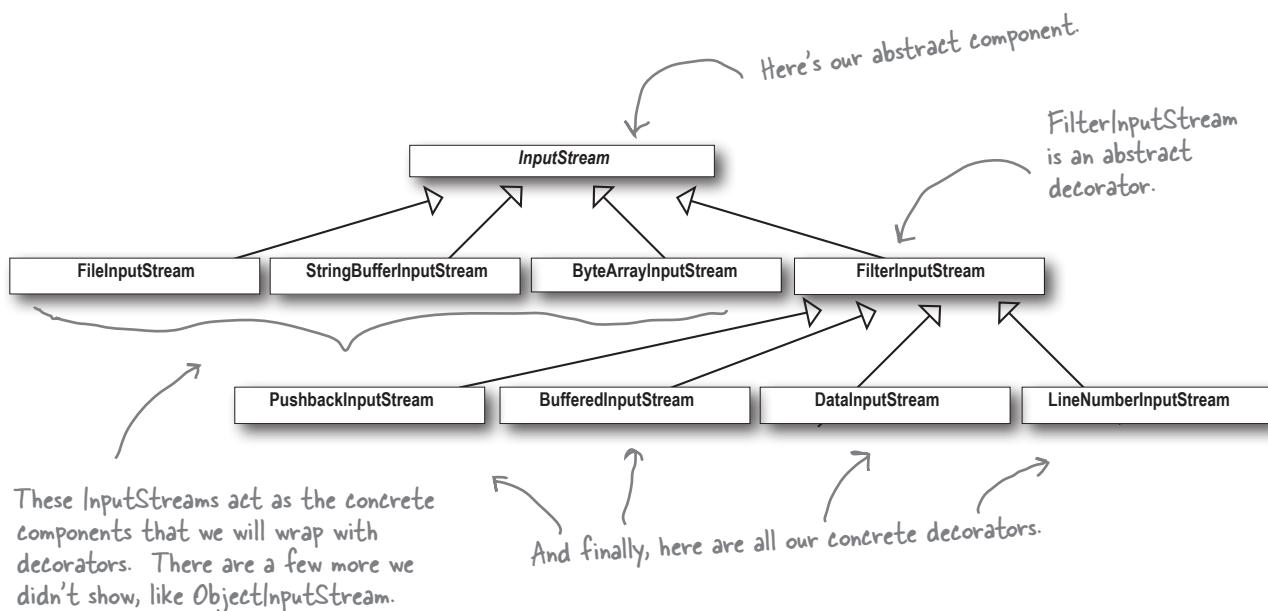
The large number of classes in the `java.io` package is... *overwhelming*. Don't feel alone if you said "whoa" the first (and second and third) time you looked at this API.

But now that you know the Decorator Pattern, the I/O classes should make more sense since the `java.io` package is largely based on Decorator. Here's a typical set of objects that use decorators to add functionality to reading data from a file:



BufferedInputStream and **LineNumberInputStream** both extend **FilterInputStream**, which acts as the abstract decorator class.

Decorating the java.io classes



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

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

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

Writing your own Java I/O Decorator

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

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

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



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

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

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

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

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.

Test out your new Java I/O Decorator

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

```

public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));
            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

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

I know the Decorator Pattern therefore I RULE!

test.txt file

You need to make this file.

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

Give it a spin:

```

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

```



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

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

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

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

HeadFirst: Can you give us an example?

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

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

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

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

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

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

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

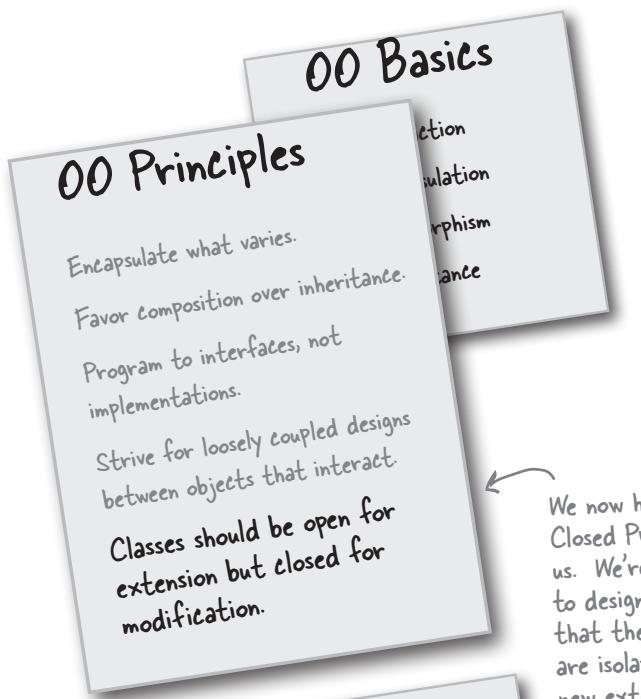
HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

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

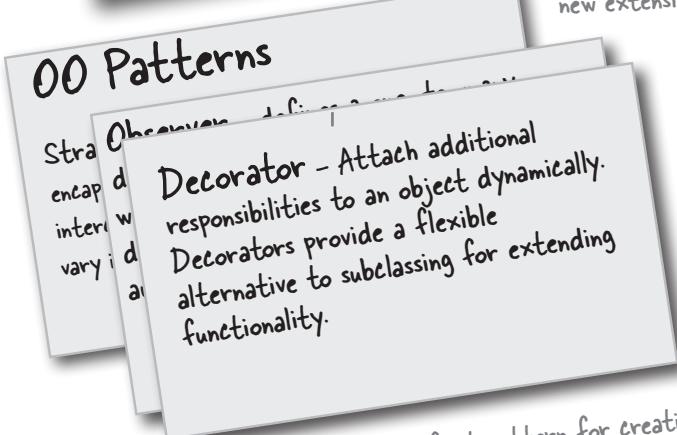


Tools for your Design Toolbox

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



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



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



BULLET POINTS

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



Sharpen your pencil

Solution

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

```

public class Beverage {

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

    public double cost() {

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

public class DarkRoast extends Beverage {

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

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

```

Sharpen your pencil

Solution

New barista training



"double mocha soy latte with whip"

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

- 2 Whip calls `cost()` on Mocha

- 3 Mocha calls `cost()` on another Mocha.

- 4 Next, Mocha calls `cost()` on Soy.

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

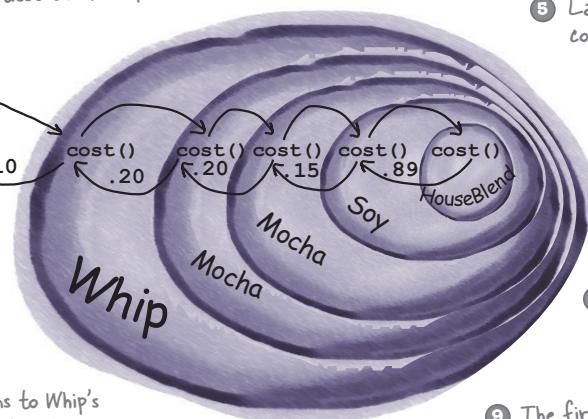
- 6 HouseBlend's `cost()` method returns .89 cents and pops off the stack.

- 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.

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





Sharpen your pencil

Solution

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

How would you alter the decorator classes to handle this change in requirements? Here's our solution.

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

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

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

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

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

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

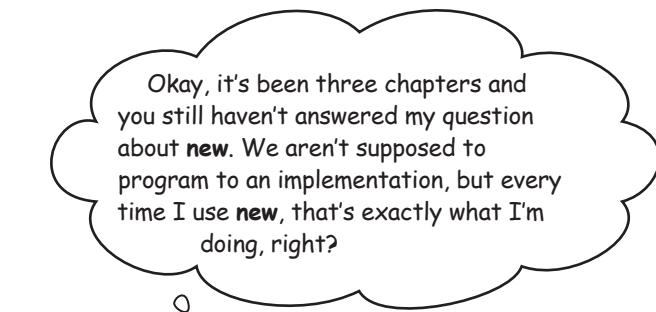
Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

4 the Factory Pattern

Baking with OO Goodness



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



Okay, it's been three chapters and you still haven't answered my question about `new`. We aren't supposed to program to an implementation, but every time I use `new`, that's exactly what I'm doing, right?

When you see “new,” think “concrete.”

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

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

We want to use interfaces
to keep code flexible.

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

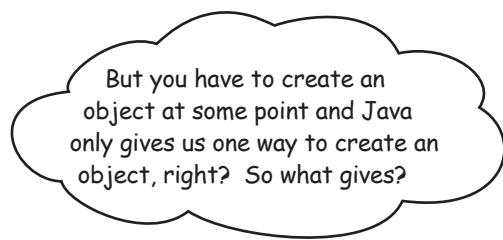
When you have a whole set of related concrete classes, often you're forced to write code like this:

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

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

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

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



What's wrong with “new”?

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

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

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

Remember that designs should be “open for extension but closed for modification” – see Chapter 3 for a review.



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

Identifying the aspects that vary

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

```
Pizza orderPizza() {  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```



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

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

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

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza();  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

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

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

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it! Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

You realize that all of your competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu:

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

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

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

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

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

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

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

Encapsulating object creation

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

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

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

What's going to go here?

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

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



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

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

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

Building a simple pizza factory

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

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

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

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

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

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

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

there are no Dumb Questions

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

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

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

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

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

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

Reworking the PizzaStore class

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

```
public class PizzaStore {
    SimplePizzaFactory factory;

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

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

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

        return pizza;
    }

    // other methods here
}
```

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

`PizzaStore` gets the factory passed to it in the constructor.

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

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



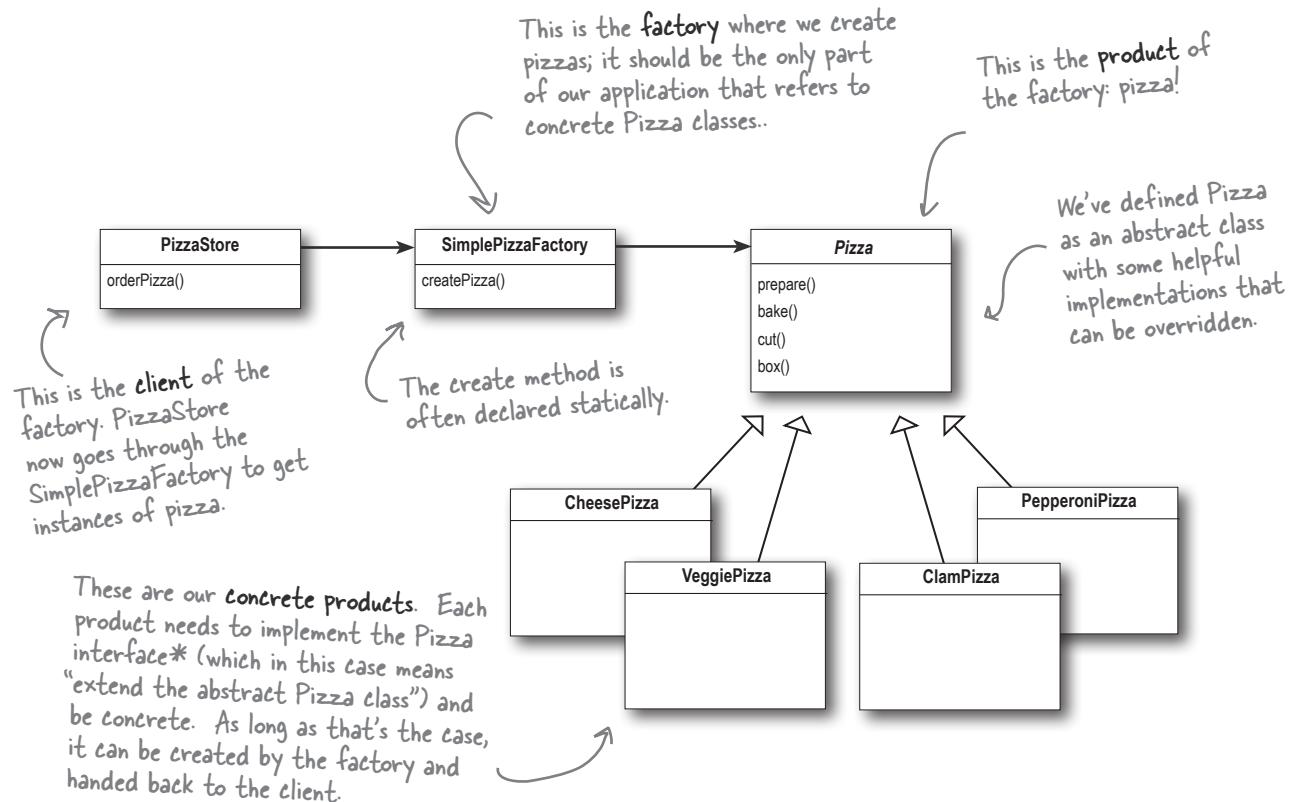
We know that object composition allows us to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to use that in the `PizzaStore`? What factory implementations might we be able to swap in and out?

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

The Simple Factory defined

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

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



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

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

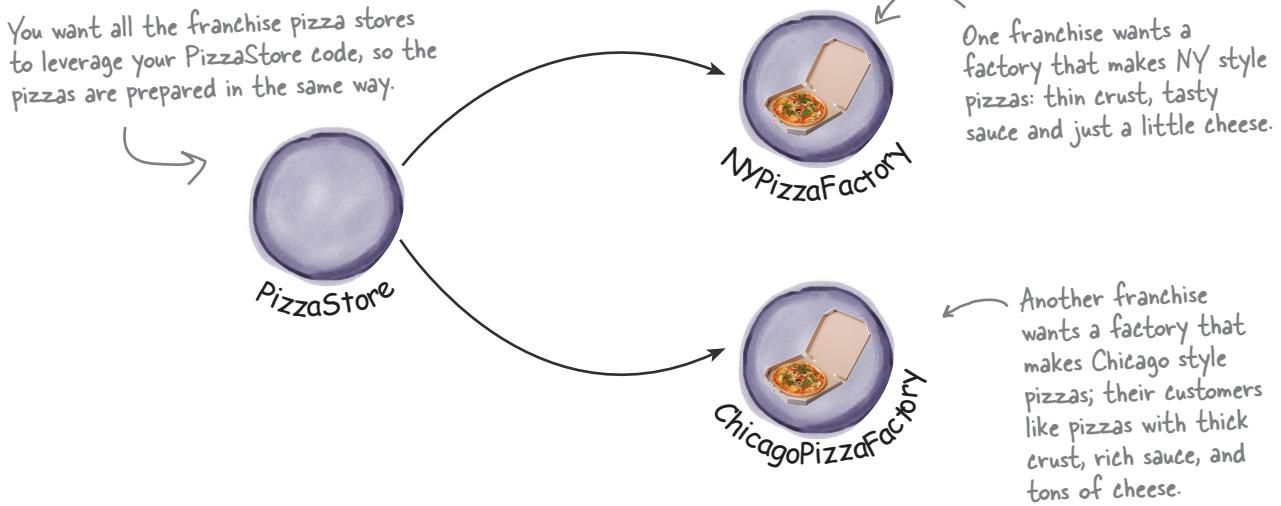


Pattern
Honorable
Mention

Franchising the pizza store

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

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



We've seen one approach...

If we take out `SimplePizzaFactory` and create three different factories—`NYPizzaFactory`, `ChicagoPizzaFactory` and `CaliforniaPizzaFactory`—then we can just compose the `PizzaStore` with the appropriate factory and a franchise is good to go. That's one approach.

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

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

Here we create a factory for making NY style pizzas.

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

...and when we make pizzas, we get NY style pizzas.

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

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

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

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

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

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

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



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

A framework for the pizza store

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

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

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

PizzaStore is now abstract (see why below).

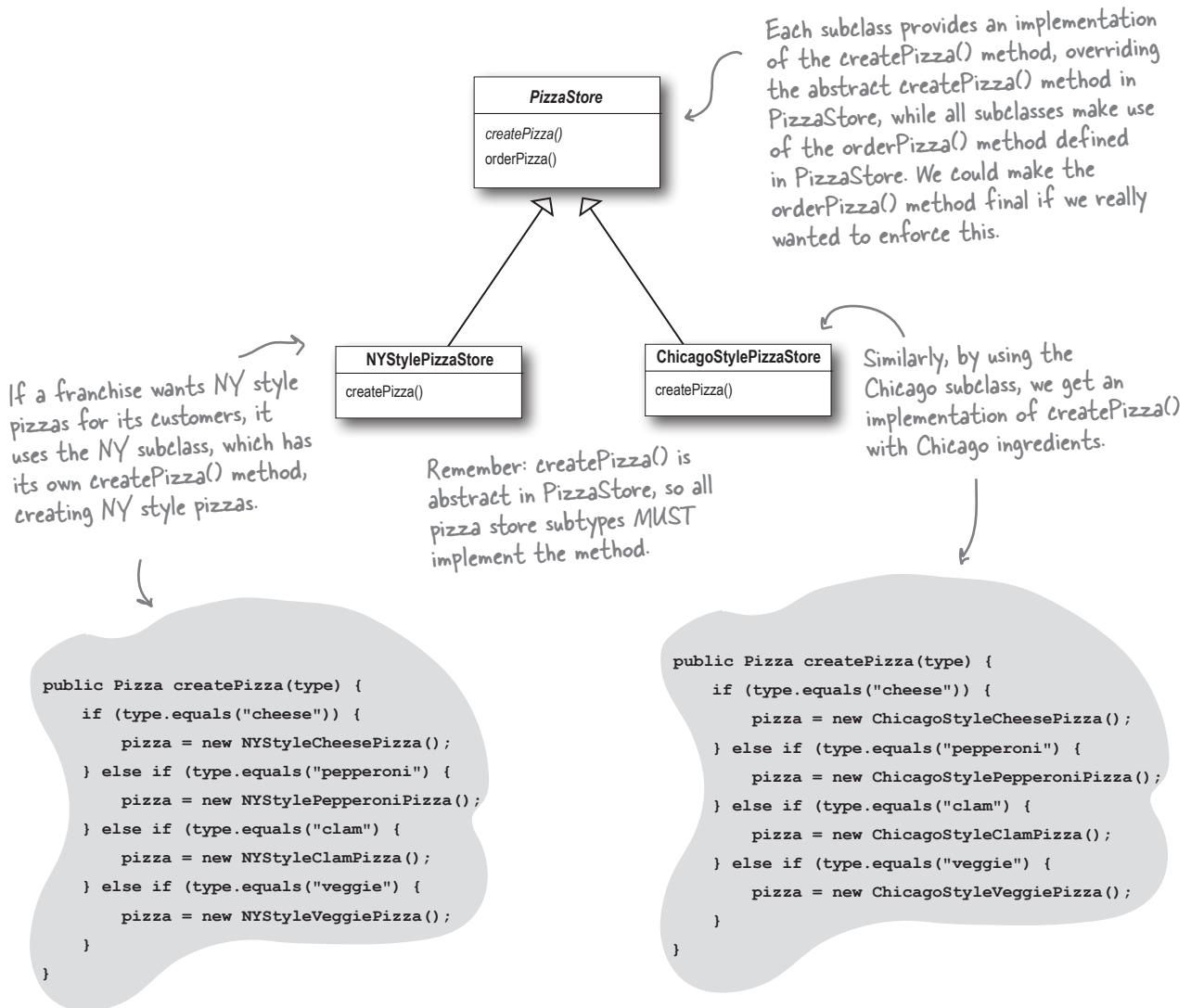
```
public abstract class PizzaStore {  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type); // ← Now createPizza is back to being a  
                                // call to a method in the PizzaStore  
                                // rather than on a factory object.  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box(); // ← All this looks just the same...  
  
        return pizza;  
    }  
  
    abstract Pizza createPizza(String type); // ← Now we've moved our factory  
                                            // object to this method.  
}  
  
Our "factory method" is now abstract in PizzaStore.
```

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

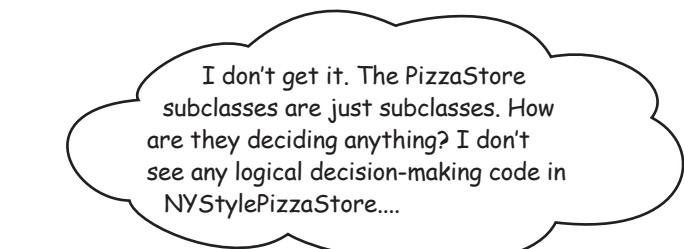
Allowing the subclasses to decide

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

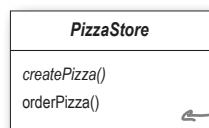
What varies among the regional PizzaStores is the style of pizzas they make—New York Pizza has thin crust, Chicago Pizza has thick, and so on—and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is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



how do subclasses decide?

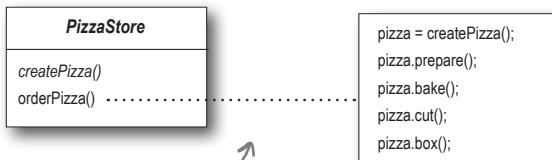


Well, think about it from the point of view of the PizzaStore's `orderPizza()` method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.



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

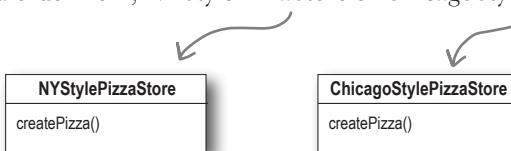
Now, to take this a little further, the `orderPizza()` method does a lot of things with a `Pizza` object (like `prepare`, `bake`, `cut`, `box`), but because `Pizza` is abstract, `orderPizza()` has no idea what real concrete classes are involved. In other words, it's decoupled!



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

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

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



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

Let's make a PizzaStore

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

Here's the New York regional style:

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

```
public class NYPizzaStore extends PizzaStore {

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

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

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

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

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

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



Sharpen your pencil

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

Declaring a factory method

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

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);
    // other methods here
}
```

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

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

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



Code Up Close

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

abstract Product factoryMethod(String type)

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

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

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

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

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



So how do they order?

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

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

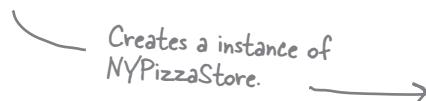
Behind the Scenes



1

Let's follow Ethan's order: first we need a NY PizzaStore:

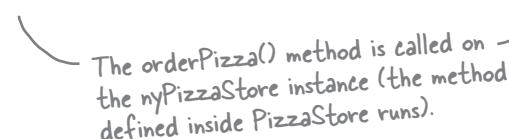
```
PizzaStore nyPizzaStore = new NYPizzaStore();
```



2

Now that we have a store, we can take an order:

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



3

The orderPizza() method then calls the createPizza() method:

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

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



4

Finally, we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

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

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

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

We're just missing one thing: PIZZA!

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



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

```
public abstract class Pizza {  
    String name;  
    String dough;  
    String sauce;  
    ArrayList<String> toppings = new ArrayList<String>();  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        System.out.println("Tossing dough...");  
        System.out.println("Adding sauce...");  
        System.out.println("Adding toppings: ");  
        for (String topping : toppings) {  
            System.out.println("    " + topping);  
        }  
    }  
  
    void bake() {  
        System.out.println("Bake for 25 minutes at 350");  
    }  
  
    void cut() {  
        System.out.println("Cutting the pizza into diagonal slices");  
    }  
  
    void box() {  
        System.out.println("Place pizza in official PizzaStore box");  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

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

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

Preparation follows a number of steps in a particular sequence.

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

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

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

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

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

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

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

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

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

```
public class PizzaTestDrive {  
  
    public static void main(String[] args) {  
        PizzaStore nyStore = new NYPizzaStore();  
        PizzaStore chicagoStore = new ChicagoPizzaStore();  
  
        Pizza pizza = nyStore.orderPizza("cheese");  
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");  
  
        pizza = chicagoStore.orderPizza("cheese");  
        System.out.println("Joel ordered a " + pizza.getName() + "\n");  
    }  
}
```

First we create two different stores.

Then use one store to make Ethan's order.

And the other for Joel's.

File Edit Window Help YouWantMootzOnThatPizza?

%java PizzaTestDrive

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

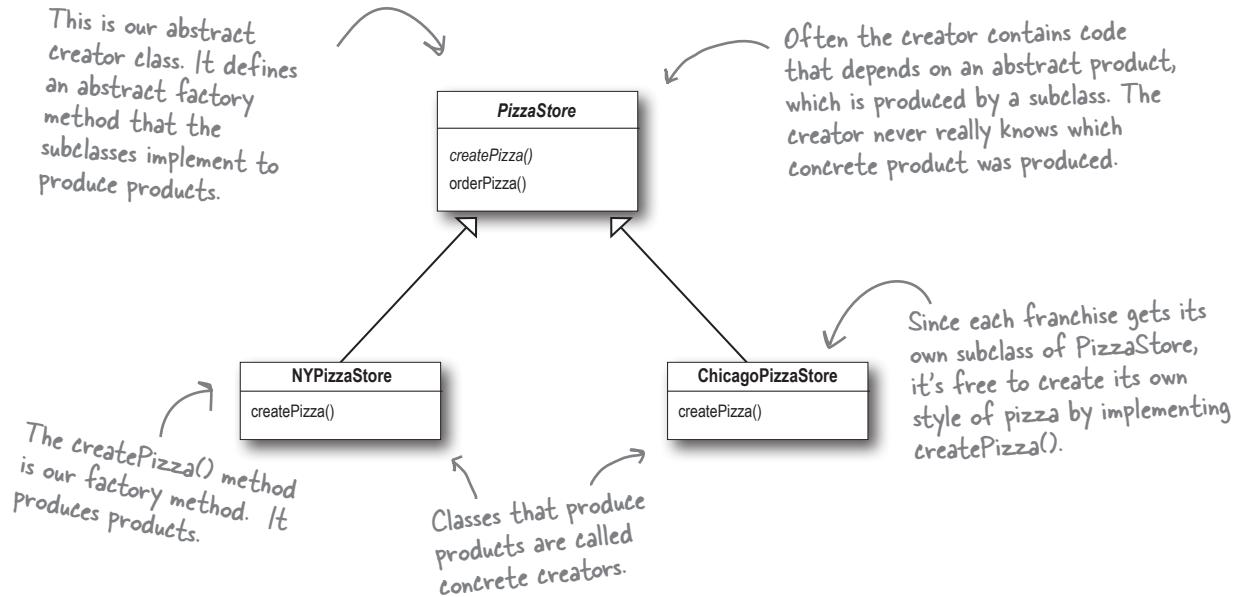
```
Preparing Chicago Style Deep Dish Cheese Pizza  
Tossing dough...  
Adding sauce...  
Adding toppings:  
    Shredded Mozzarella Cheese  
Bake for 25 minutes at 350  
Cutting the pizza into square slices  
Place pizza in official PizzaStore box  
Joel ordered a Chicago Style Deep Dish Cheese Pizza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

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

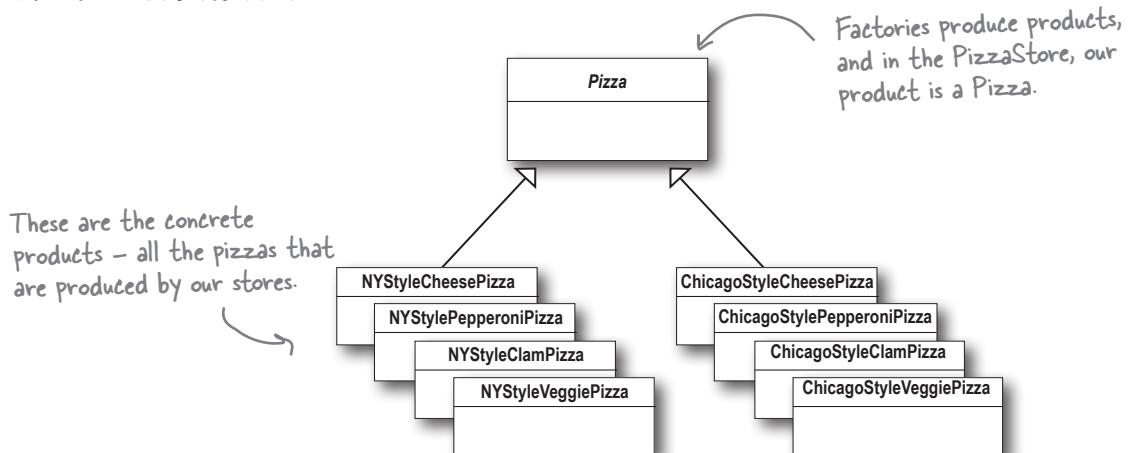
It's finally time to meet the Factory Method Pattern

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

The Creator classes



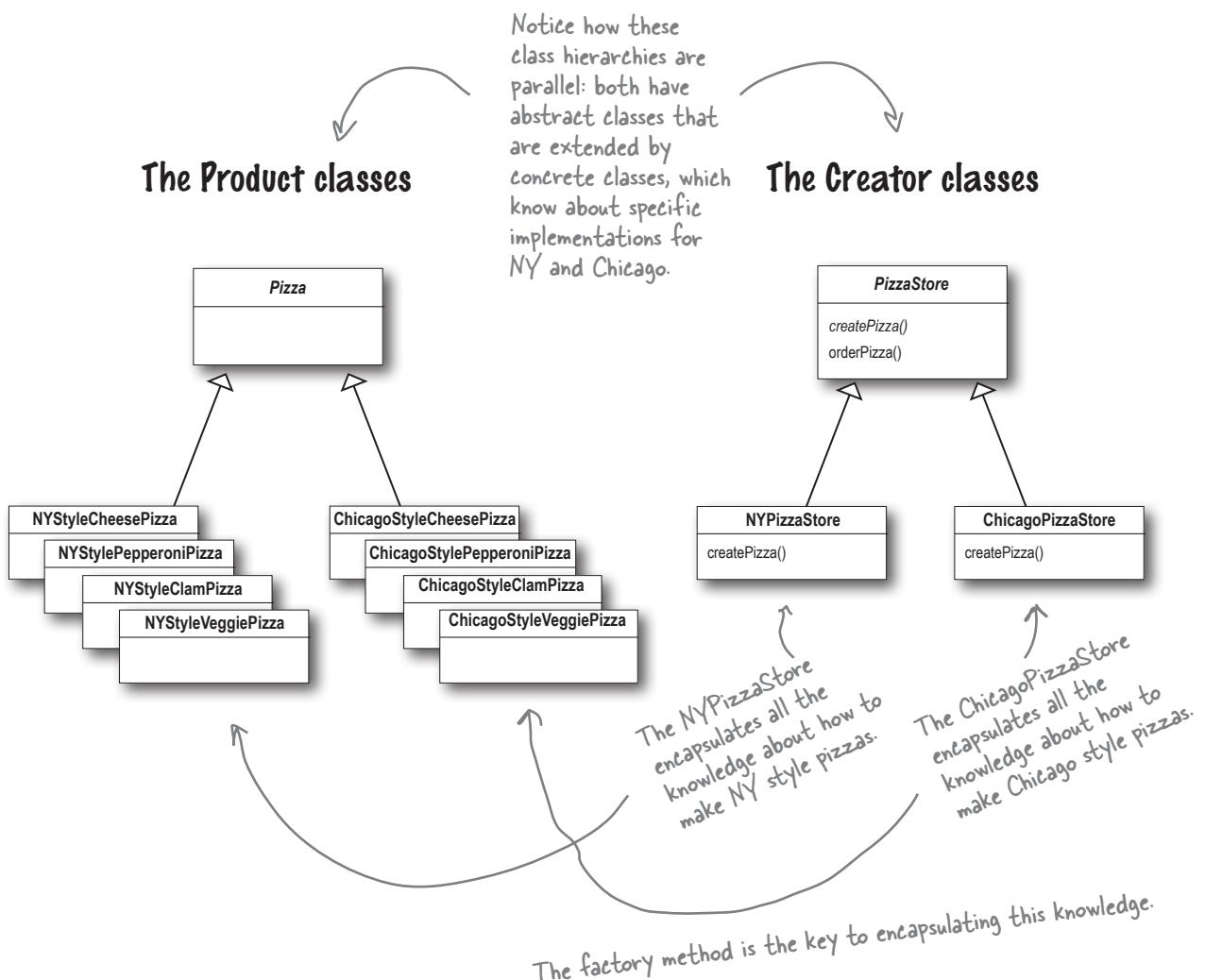
The Product classes



Another perspective: parallel class hierarchies

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

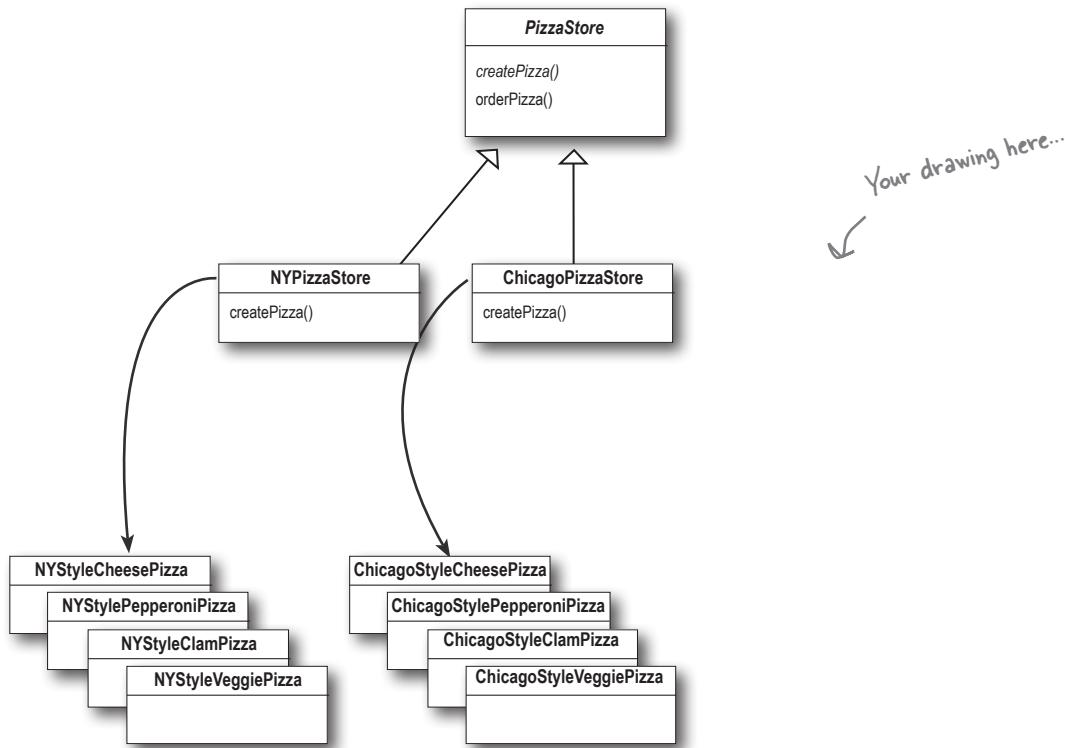
Let's look at the two parallel class hierarchies and see how they relate:





Design Puzzle

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



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

Factory Method Pattern defined

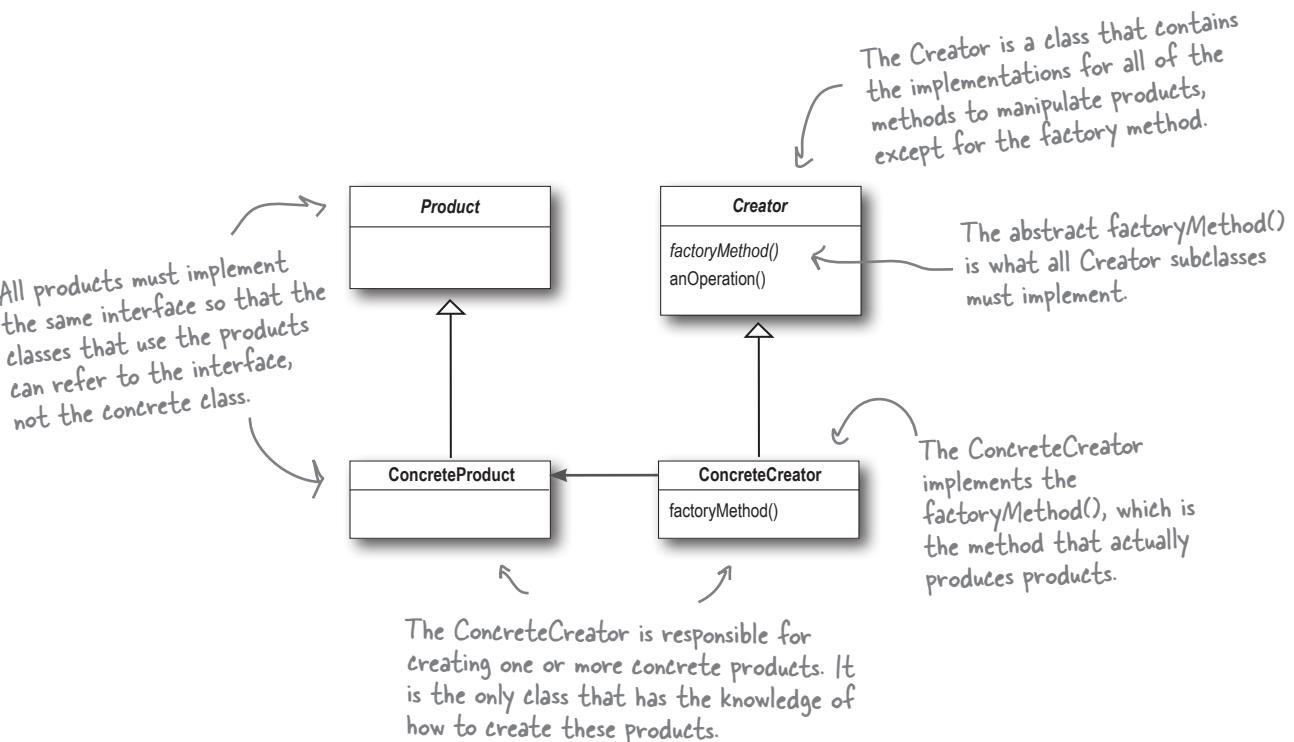
It's time to roll out the official definition of the Factory Method Pattern:

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

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

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

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



there are no Dumb Questions

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

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

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

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

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

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

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

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

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

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

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

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



Master and Student...

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

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

Master: Go on...

Student: I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I've learned a technique called “factories” that allows you to encapsulate this behavior of instantiation.

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

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

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

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

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

Student: Master, I see the truth in this.

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

A very dependent PizzaStore



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

```
public class DependentPizzaStore {

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

Handles all the NY style pizzas

Handles all the Chicago style pizzas

You can write your answers here:

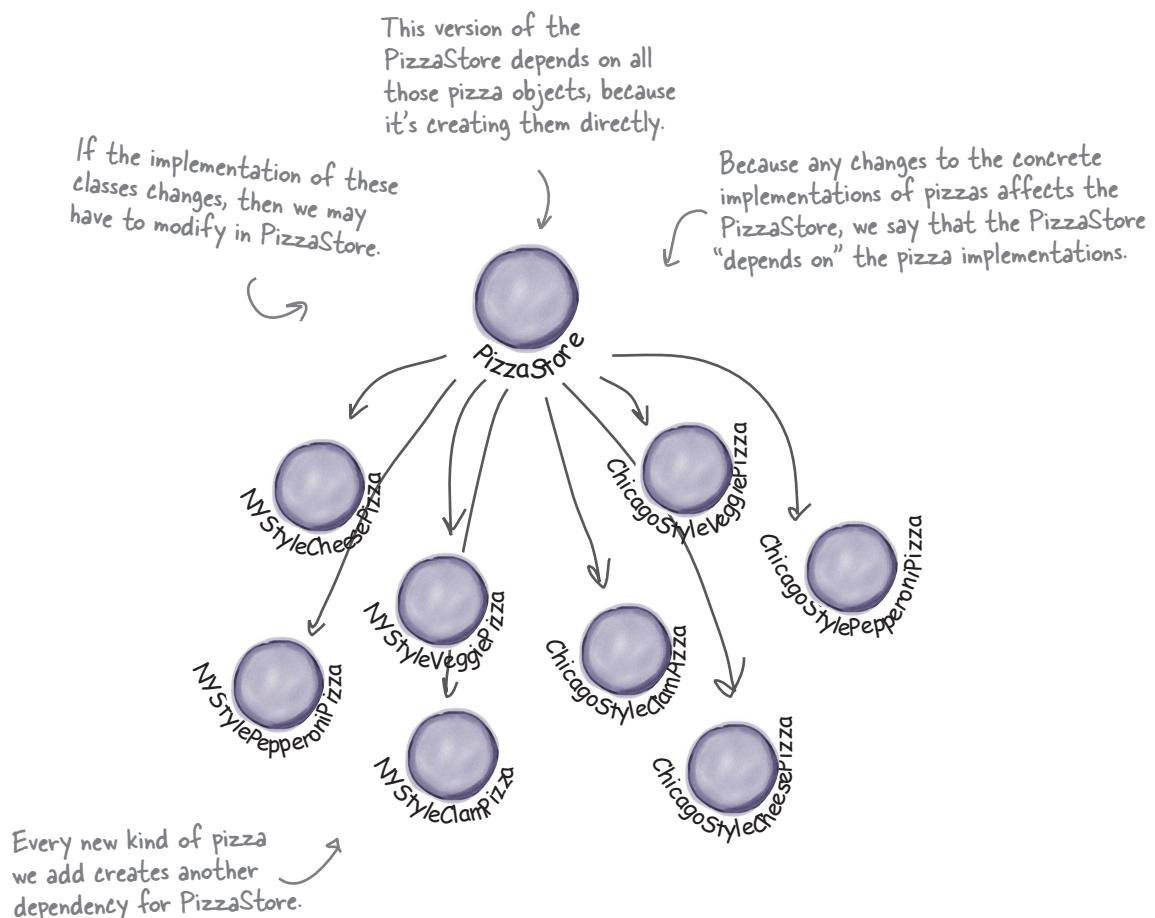
number

number with California too

Looking at object dependencies

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

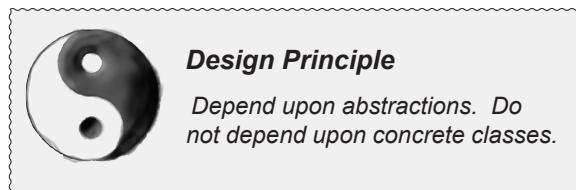
If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here's what it looks like:



The Dependency Inversion Principle

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

Here’s the general principle:



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

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

But what the heck does that mean?

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

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

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

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

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

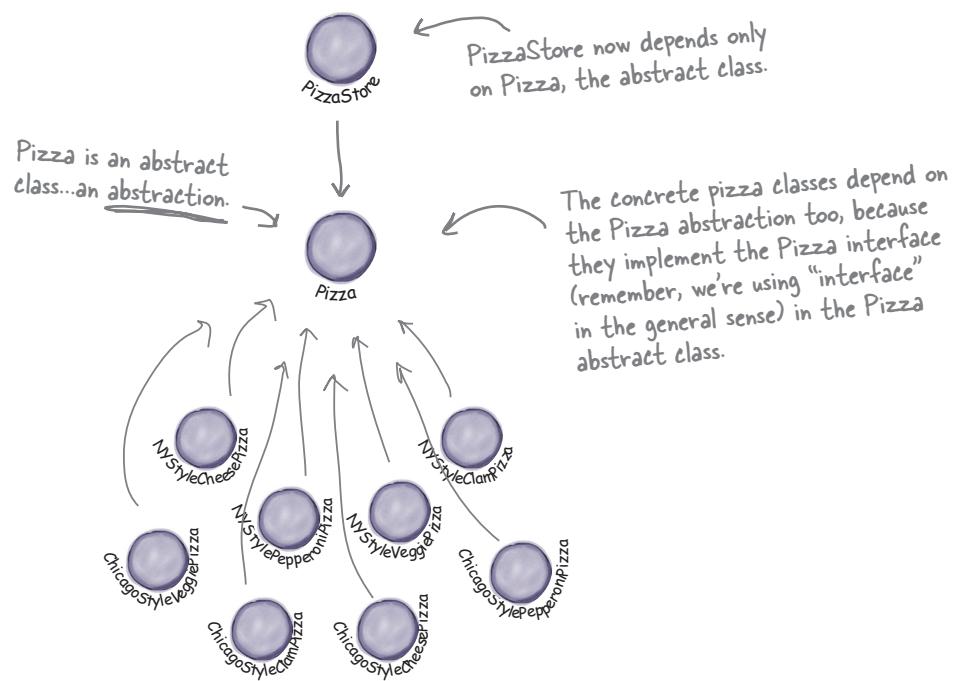
Applying the Principle

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

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

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

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



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



Where's the “inversion” in Dependency Inversion Principle?

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

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

Inverting your thinking...



Okay, so you need to implement a *PizzaStore*. What's the first thought that pops into your head?

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

Now, let's "invert" your thinking... instead of starting at the top, start at the Pizzas and think about what you can abstract.

Right! You are thinking about the abstraction *Pizza*. So now, go back and think about the design of the *Pizza Store* again.

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

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

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.

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

- No class should derive from a concrete class.

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

- No method should override an implemented method of any of its base classes.

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

But wait, aren't these guidelines impossible to follow? If I follow these, I'll never be able to write a single program!

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

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

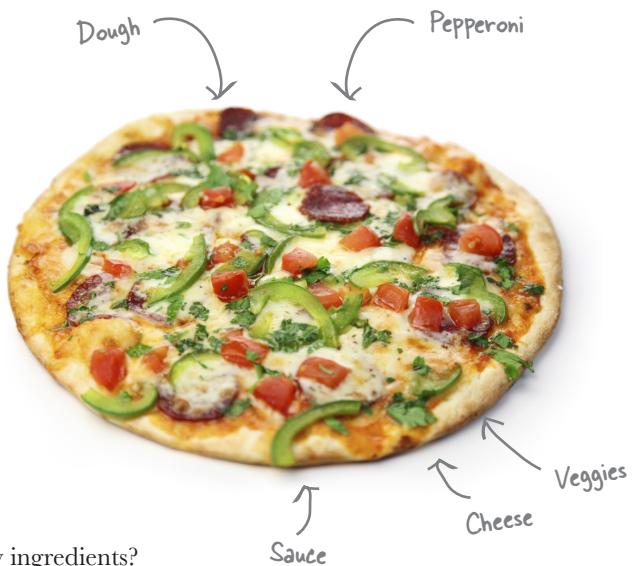
If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.



Meanwhile, back at the PizzaStore...

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

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



Ensuring consistency in your ingredients

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

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

Chicago Pizza Menu

- Cheese Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Oregano
- Veggie Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives
- Clam Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Clams
- Pepperoni Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

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

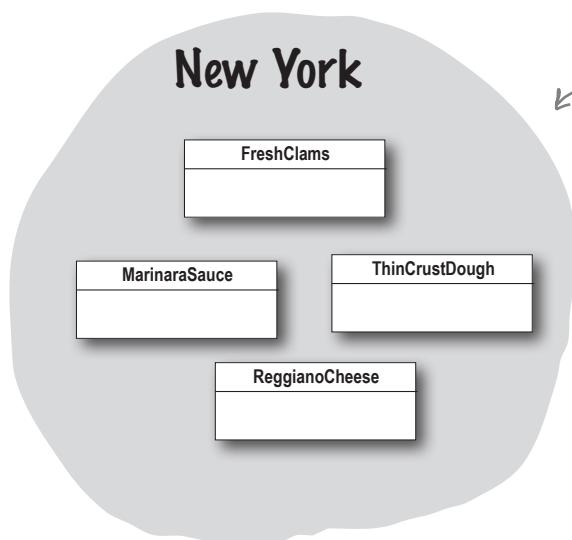
New York Pizza Menu

- Cheese Pizza
Marinara Sauce, Reggiano, Garlic
- Veggie Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers
- Clam Pizza
Marinara Sauce, Reggiano, Fresh Clams
- Pepperoni Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

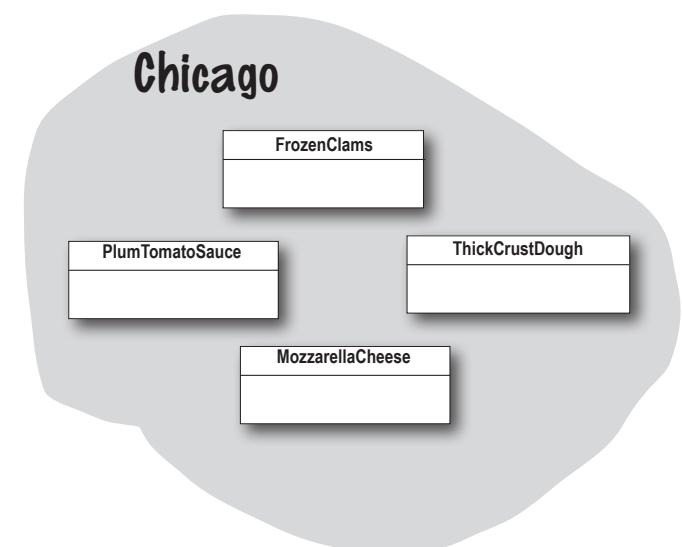
Families of ingredients...

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza, it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Seattle?

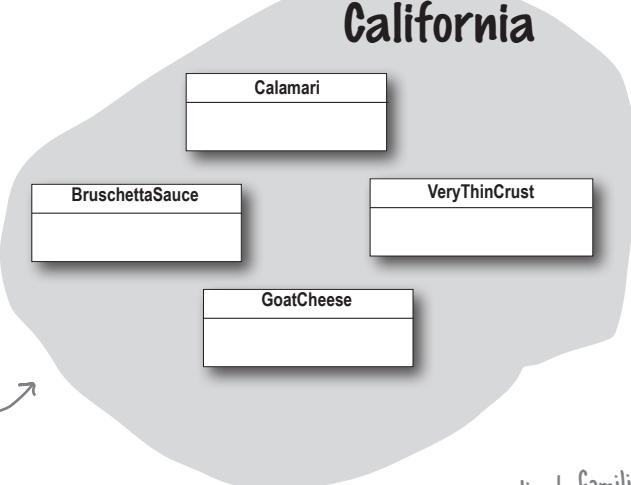
For this to work, you are going to have to figure out how to handle families of ingredients.



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



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



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

Building the ingredient factories

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

Let's start by defining an interface for the factory that is going to create all our ingredients:

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

Lots of new classes here,
one per ingredient.



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

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

Here's what we're going to do:

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

Building the New York ingredient factory

Okay, here's the implementation for the New York ingredient factory. This factory specializes in Marinara Sauce, Reggiano Cheese, Fresh Clams...

The NY ingredient factory implements the interface for all ingredient factories

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

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

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

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

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

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

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

For each ingredient in the ingredient family, we create the New York version.

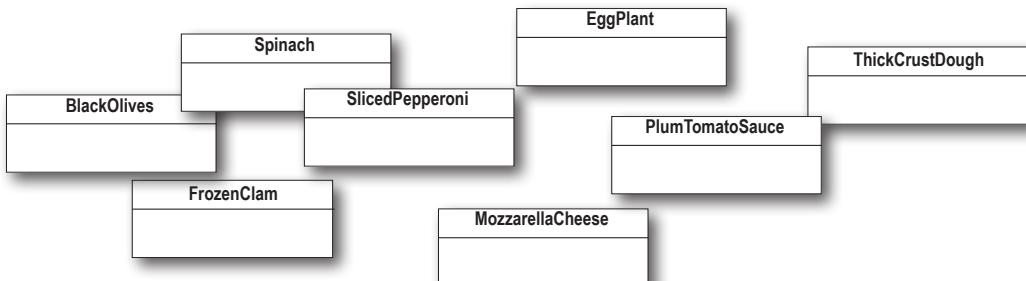
For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself



Sharpen your pencil

Write the ChicagoPizzaIngredientFactory. You can reference the classes below in your implementation:



Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract Pizza class:

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;
    abstract void prepare();
    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }
    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }
    void setName(String name) {
        this.name = name;
    }
    String getName() {
        return name;
    }
    public String toString() {
        // code to print pizza here
    }
}
```

The code is annotated with several handwritten notes and arrows:

- An arrow points from the line "abstract void prepare();" to the text: "We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory."
- An arrow points from the line "Dough dough;" to the text: "Each pizza holds a set of ingredients that are used in its preparation."
- An arrow points from the line "Our other methods remain the same, with the exception of the prepare method." to the "setName" and "getName" methods.

Reworking the pizzas, continued...

Now that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas—only this time around they will get their ingredients straight from the factory. The franchisees' days of skimping on ingredients are over!

When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the Cheese Pizza:

```
public class CheesePizza extends Pizza {  
    PizzaIngredientFactory ingredientFactory;  
  
    public CheesePizza(PizzaIngredientFactory ingredientFactory) {  
        this.ingredientFactory = ingredientFactory;  
    }  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        dough = ingredientFactory.createDough();  
        sauce = ingredientFactory.createSauce();  
        cheese = ingredientFactory.createCheese();  
    }  
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.



Code Up Close

The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Rockies, the Pacific Northwest, and beyond.

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Let's check out the ClamPizza as well:

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

ClamPizza also stashes an ingredient factory.

To make a clam pizza, the prepare method collects the right ingredients from its local factory.

If it's a New York factory, the clams will be fresh; if it's Chicago, they'll be frozen.

Revisiting our pizza stores

We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```
public class NYPizzaStore extends PizzaStore {  
  
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;  
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory();  
  
        if (item.equals("cheese")) {  
  
            pizza = new CheesePizza(ingredientFactory);  
            pizza.setName("New York Style Cheese Pizza");  
  
        } else if (item.equals("veggie")) {  
  
            pizza = new VeggiePizza(ingredientFactory);  
            pizza.setName("New York Style Veggie Pizza");  
  
        } else if (item.equals("clam")) {  
  
            pizza = new ClamPizza(ingredientFactory);  
            pizza.setName("New York Style Clam Pizza");  
  
        } else if (item.equals("pepperoni")) {  
  
            pizza = new PepperoniPizza(ingredientFactory);  
            pizza.setName("New York Style Pepperoni Pizza");  
  
        }  
        return pizza;  
    }  
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.



We now pass each pizza the factory that should be used to produce its ingredients.



Look back one page and make sure you understand how the pizza and the factory work together!



For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.



Compare this version of the createPizza() method to the one in the Factory Method implementation earlier in the chapter.

What have we done?

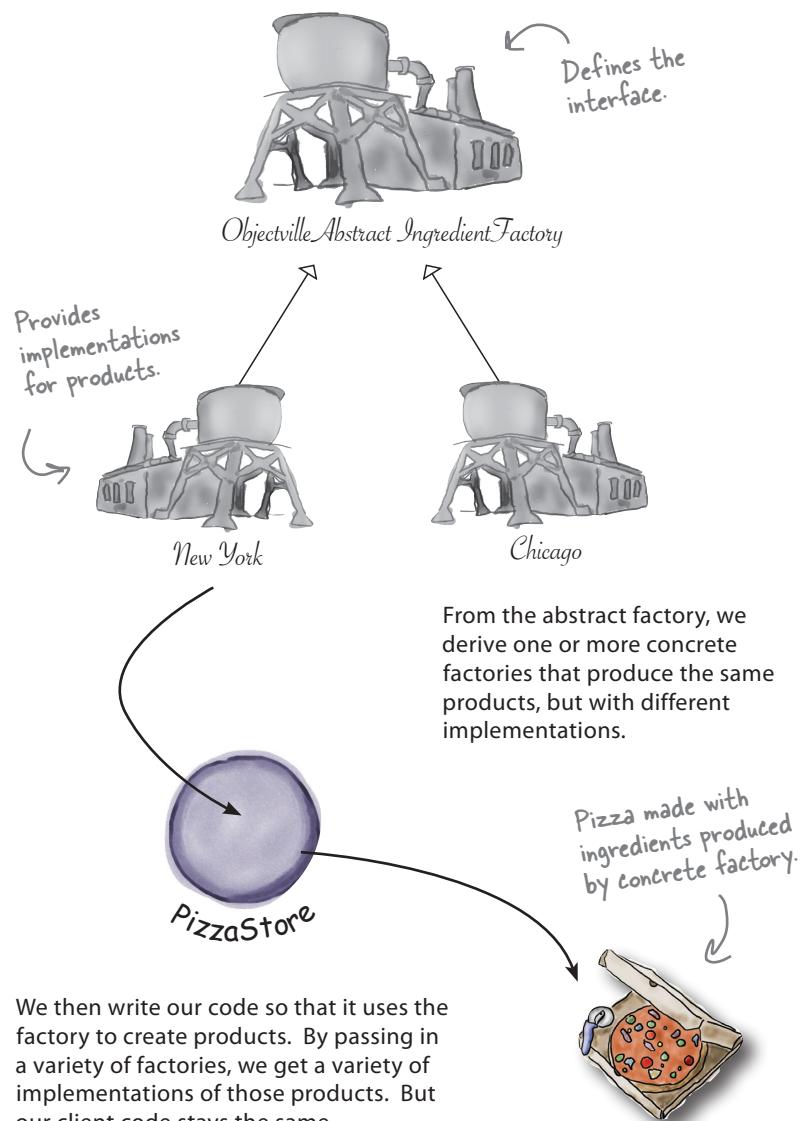
That was quite a series of code changes; what exactly did we do?

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory.

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts—such as different regions, different operating systems, or different look and feels.

Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like getting marinara instead of plum tomatoes).

An Abstract Factory provides an interface for a family of products. What's a family? In our case, it's all the things we need to make a pizza: dough, sauce, cheese, meats, and veggies.



More pizza for Ethan and Joel...

Ethan and Joel can't get enough Objectville Pizza! What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...

Behind
the Scenes



The first part of the order process hasn't changed at all.
Let's follow Ethan's order again:

1 First we need a NY PizzaStore:

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

Creates an instance
of NYPizzaStore.

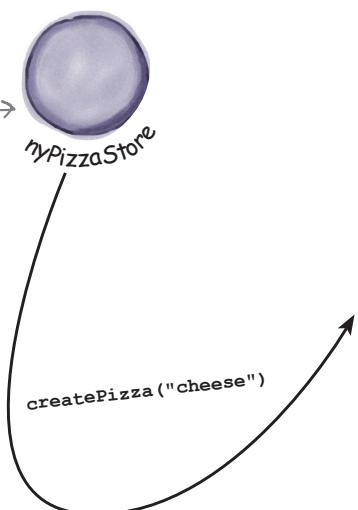
2 Now that we have a store, we can take an order:

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

The orderPizza() method is called
on the nyPizzaStore instance.

3 The orderPizza() method first calls the createPizza() method:

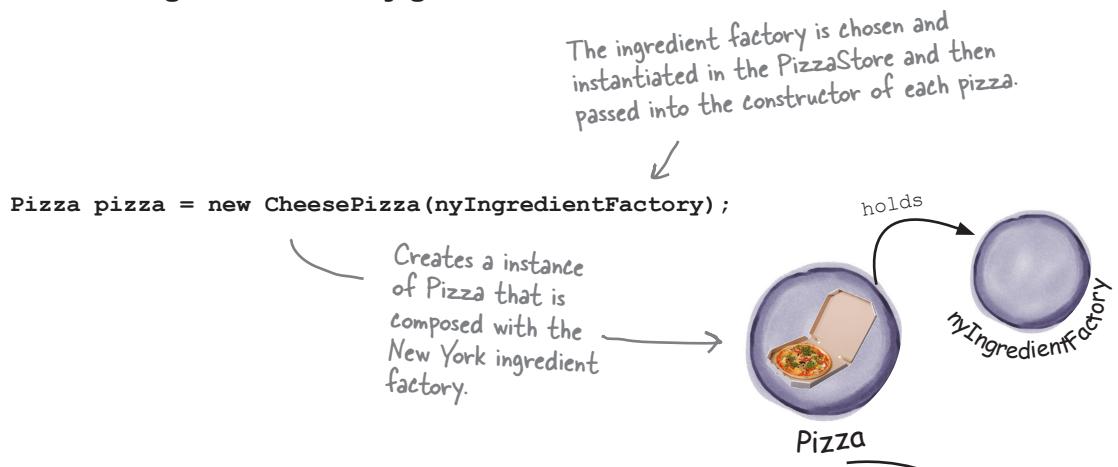
```
Pizza pizza = createPizza("cheese");
```



From here things change, because we are using an ingredient factory



- 4 When the `createPizza()` method is called, that's when our ingredient factory gets involved:**



- 5 Next we need to prepare the pizza. Once the `prepare()` method is called, the factory is asked to prepare ingredients:**

```

void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
  
```

Annotations point from the `factory` variable to three lines of code: `createDough()`, `createSauce()`, and `createCheese()`. Arrows point from these lines to the corresponding pizza toppings: `Thin crust`, `Marinara`, and `Reggiano`. A large circle labeled `prepare()` encloses the entire `prepare()` method block.

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

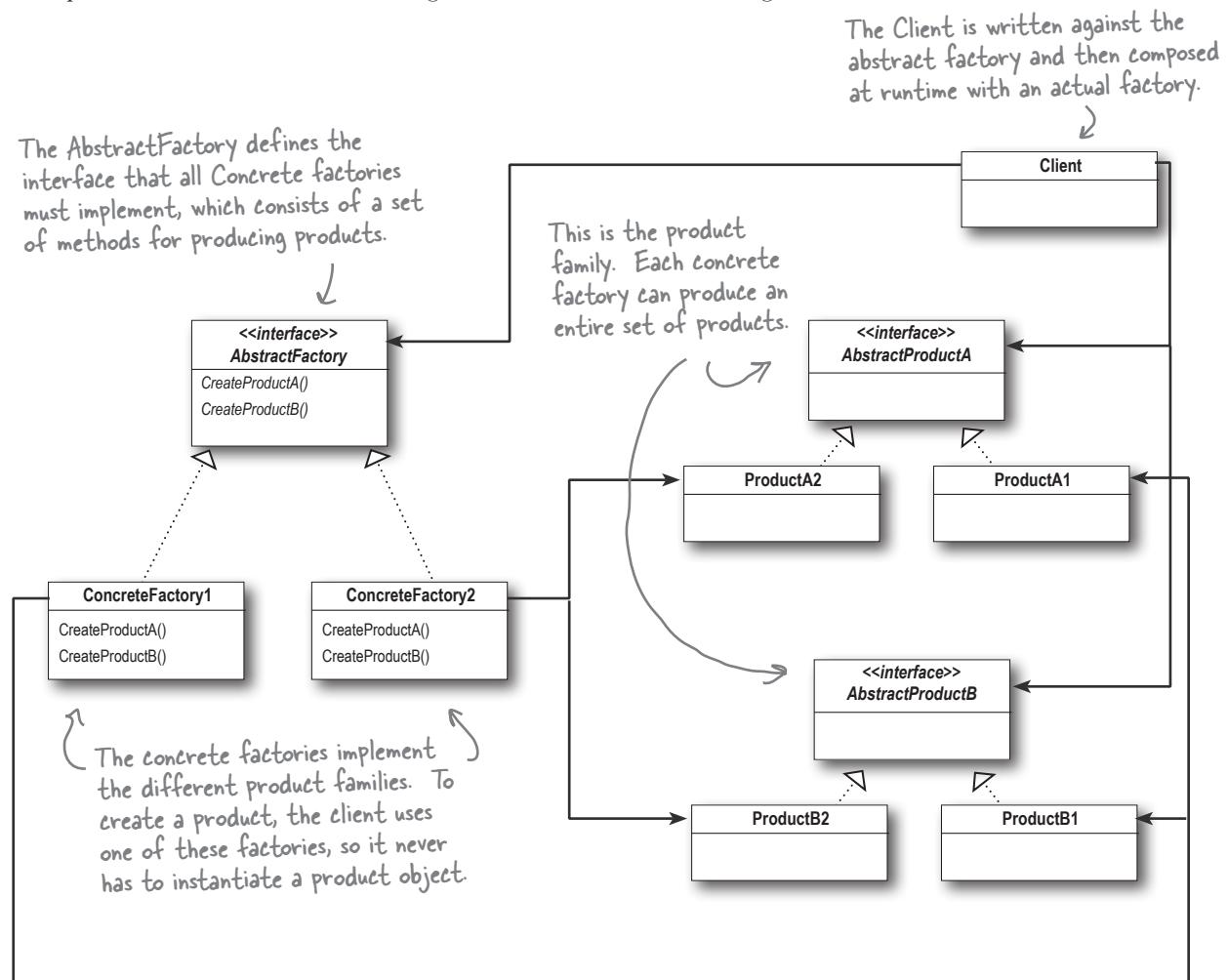
- 6 Finally, we have the prepared pizza in hand and the `orderPizza()` method bakes, cuts, and boxes the pizza.**

Abstract Factory Pattern defined

We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:



That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:

The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza.

NYPizzalnredientFactory

```
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
```

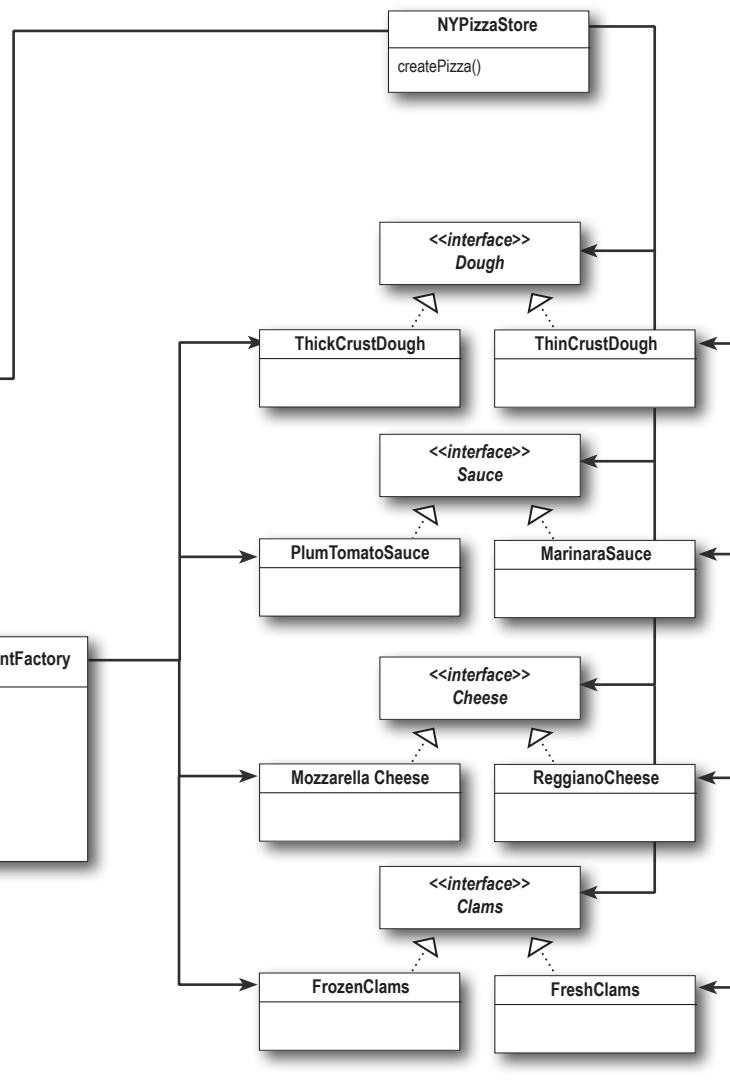
ChicagoPizzalnredientFactory

```
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
```

<<interface>> PizzaIngredientFactory

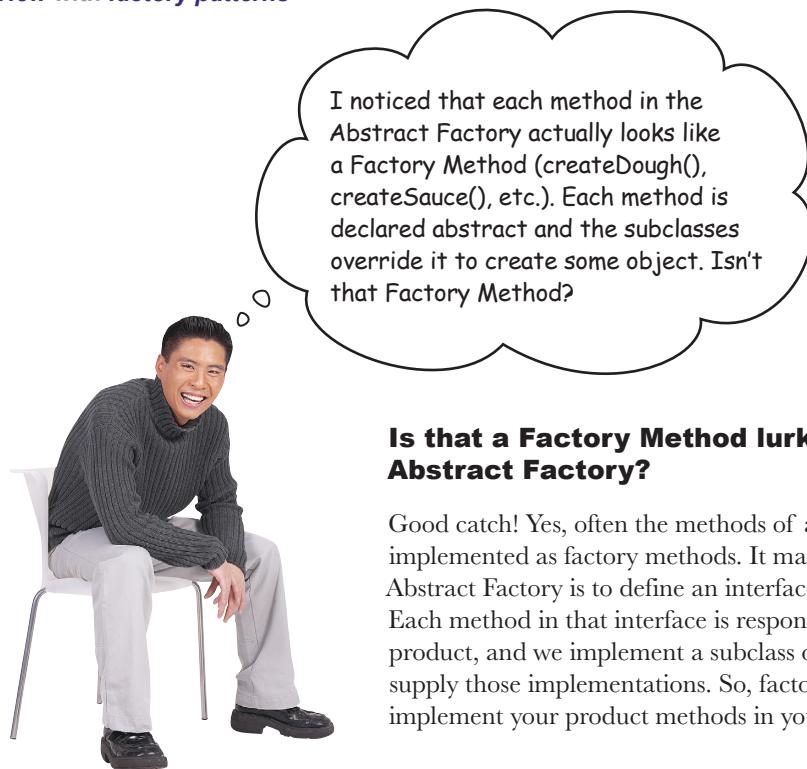
```
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
```

The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.



The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.

Each factory produces a different implementation for the family of products.



I noticed that each method in the Abstract Factory actually looks like a Factory Method (`createDough()`, `createSauce()`, etc.). Each method is declared abstract and the subclasses override it to create some object. Isn't that Factory Method?

Is that a Factory Method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.



Patterns Exposed

This week's interview:
Factory Method and Abstract Factory, on each other

HeadFirst: Wow, an interview with two patterns at once! This is a first for us.

Factory Method: Yeah, I'm not so sure I like being lumped in with Abstract Factory, you know. Just because we're both factory patterns doesn't mean we shouldn't get our own interviews.

HeadFirst: Don't be miffed, we wanted to interview you together so we could help clear up any confusion about who's who for the readers. You do have similarities, and I've heard that people sometimes get you confused.

Abstract Factory: It is true, there have been times I've been mistaken for Factory Method, and I know you've had similar issues, Factory Method. We're both really good at decoupling applications from specific implementations; we just do it in different ways. So I can see why people might sometimes get us confused.

Factory Method: Well, it still ticks me off. After all, I use classes to create and you use objects; that's totally different!

HeadFirst: Can you explain more about that, Factory Method?

Factory Method: Sure. Both Abstract Factory and I create objects—that's our jobs. But I do it through inheritance...

Abstract Factory: ...and I do it through object composition.

Factory Method: Right. So that means, to create objects using Factory Method, you need to extend a class and provide an implementation for a factory method.

HeadFirst: And that factory method does what?

Factory Method: It creates objects, of course! I mean, the whole point of the Factory Method Pattern is that you're using a subclass to do your creation for you. In that way, clients only need to know the abstract type they are using, the subclass worries about the concrete type. So, in other words, I keep clients decoupled from the concrete types.

Abstract Factory: And I do too, only I do it in a different way.

HeadFirst: Go on, Abstract Factory... you said something about object composition?

Abstract Factory: I provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type. So, like Factory Method, my clients are decoupled from the actual concrete products they use.

HeadFirst: Oh, I see, so another advantage is that you group together a set of related products.

Abstract Factory: That's right.

HeadFirst: What happens if you need to extend that set of related products to, say, add another one? Doesn't that require changing your interface?

Abstract Factory: That's true; my interface has to change if new products are added, which I know people don't like to do....

Factory Method: <snicker>

Abstract Factory: What are you snickering at, Factory Method?

Factory Method: Oh, come on, that's a big deal! Changing your interface means you have to go in and change the interface of every subclass! That sounds like a lot of work.

Abstract Factory: Yeah, but I need a big interface because I am used to creating entire families of products. You're only creating one product, so you don't really need a big interface, you just need one method.

HeadFirst: Abstract Factory, I heard that you often use factory methods to implement your concrete factories?

Abstract Factory: Yes, I'll admit it, my concrete factories often implement a factory method to create their products. In my case, they are used purely to create products...

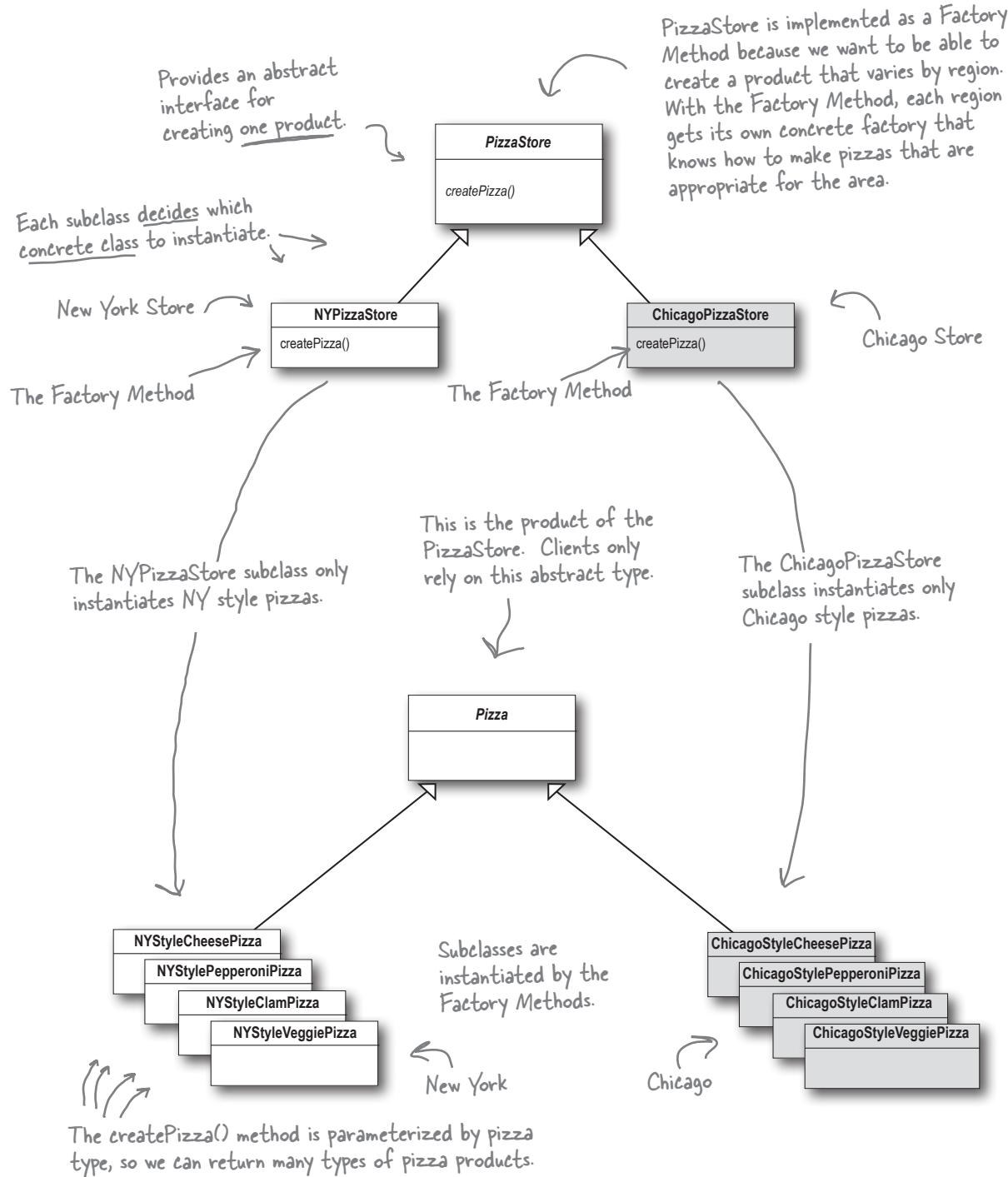
Factory Method: ...while in my case I usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

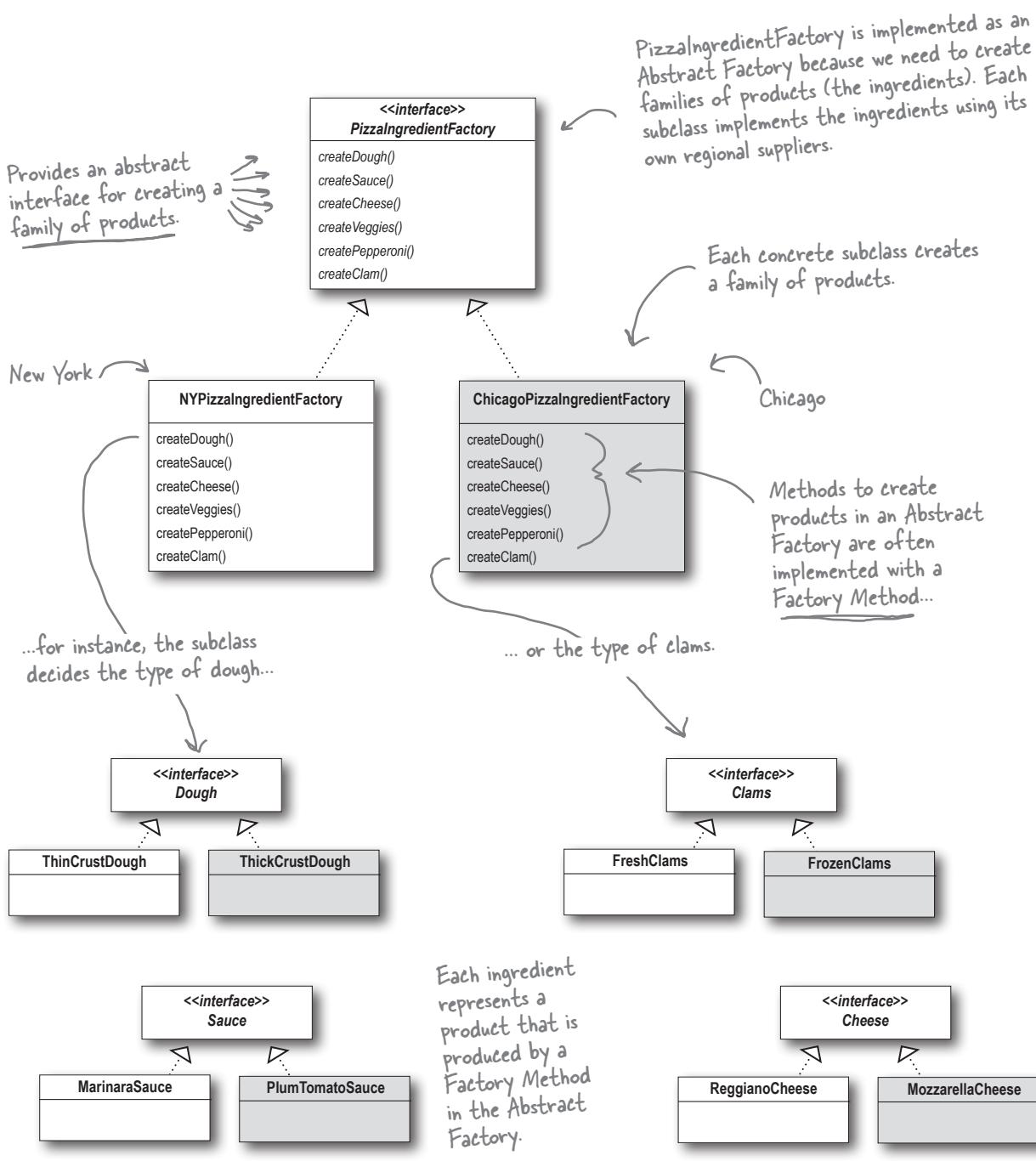
HeadFirst: It sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. You both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, which is really great, whether you're using Factory Method or Abstract Factory. May I allow you each a parting word?

Abstract Factory: Thanks. Remember me, Abstract Factory, and use me whenever you have families of products you need to create and you want to make sure your clients create products that belong together.

Factory Method: And I'm Factory Method; use me to decouple your client code from the concrete classes you need to instantiate, or if you don't know ahead of time all the concrete classes you are going to need. To use me, just subclass me and implement my factory method!

Factory Method and Abstract Factory compared





The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.



Tools for your Design Toolbox

In this chapter, we added two more tools to your toolbox: Factory Method and Abstract Factory. Both patterns encapsulate object creation and allow you to decouple your code from concrete types.



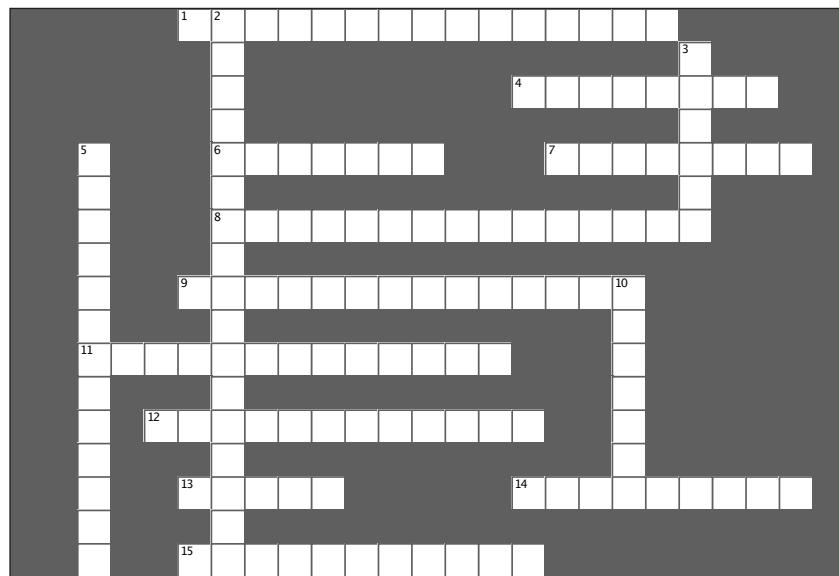
BULLET POINTS

- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.



Design Patterns Crossword

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter.



ACROSS

1. In Factory Method, each franchise is a _____.
4. In Factory Method, who decides which class to instantiate?
6. Role of PizzaStore in Factory Method Pattern.
7. All New York style pizzas use this kind of cheese.
8. In Abstract Factory, each ingredient factory is a _____.
9. When you use new, you are programming to an _____.
11. createPizza() is a _____ (two words).
12. Joel likes this kind of pizza.
13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction.
14. When a class instantiates an object from a concrete class, it's _____ on that object.
15. All factory patterns allow us to _____ object creation.

DOWN

2. We used _____ in Simple Factory and Abstract Factory, and inheritance in Factory Method.
3. Abstract Factory creates a _____ of products.
5. Not a REAL factory pattern, but handy nonetheless.
10. Ethan likes this kind of pizza.



Sharpen your pencil

Solution

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

Both of these stores are almost exactly like the New York store... they just create different kinds of pizzas.

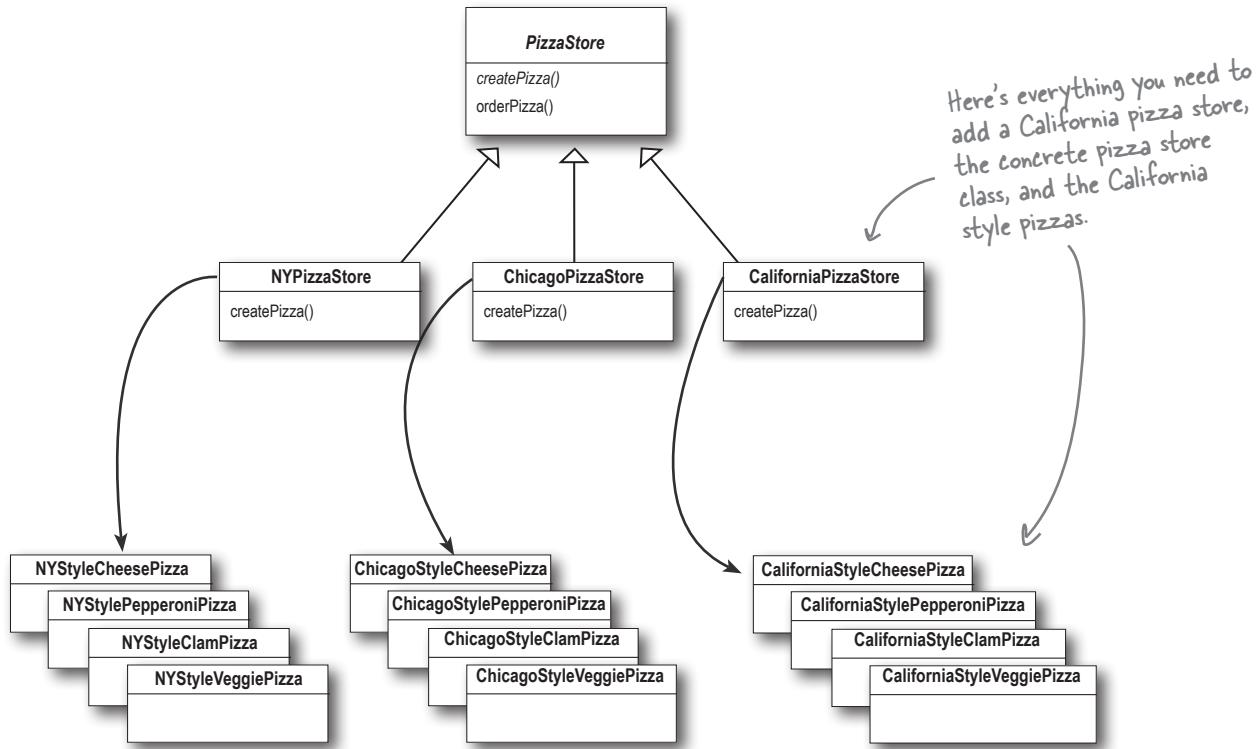
```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza(); ← For the Chicago pizza
        } else if (item.equals("veggie")) { ← store, we just have to
            return new ChicagoStyleVeggiePizza(); ← make sure we create
        } else if (item.equals("clam")) { ← Chicago style pizzas...
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}

public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza(); ← and for the California
        } else if (item.equals("veggie")) { ← pizza store, we create
            return new CaliforniaStyleVeggiePizza(); ← California style pizzas.
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```



Design Puzzle Solution

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



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

Here
are our
suggestions...

Mashed Potatoes with Roasted Garlic

BBQ Sauce

Artichoke Hearts

M&M's

Peanuts

A very dependent PizzaStore



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

```
public class DependentPizzaStore {

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

Handles all the NY style pizzas

Handles all the Chicago style pizzas

You can write your answers here:

8 number

12 number with California too

Sharpen your pencil

Solution

Go ahead and write the ChicagoPizzaIngredientFactory; you can reference the classes below in your implementation:

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

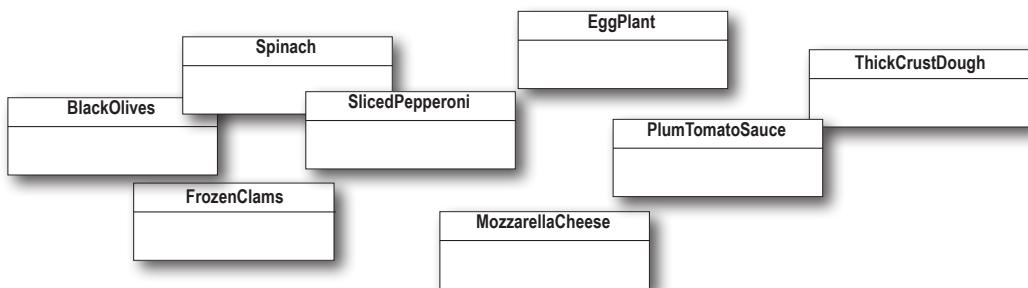
    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

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

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
            new Spinach(),
            new Eggplant() };
        return veggies;
    }

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

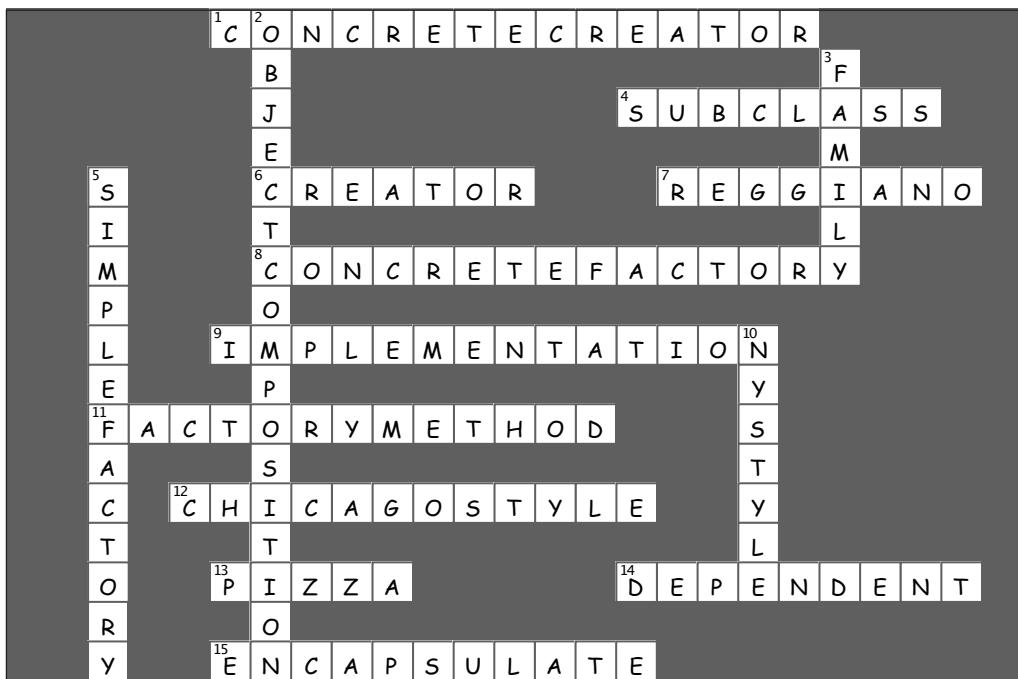
    public Clams createClam() {
        return new FrozenClams();
    }
}
```





Design Patterns Crossword Solution

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter. Here's the solution.

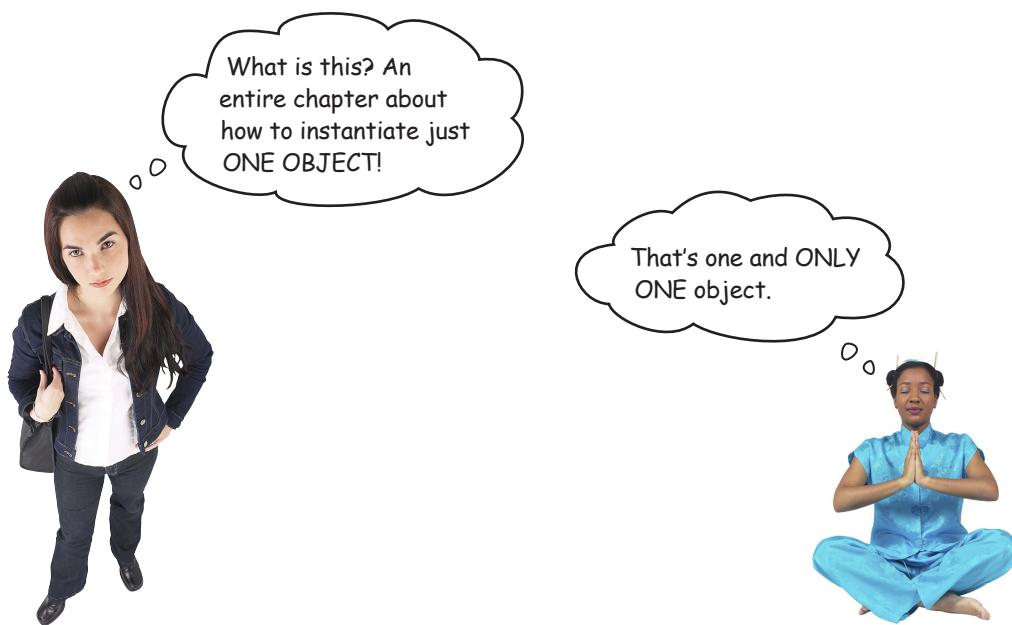


5 the Singleton Pattern

* One of a Kind Objects *



Our next stop is the Singleton Pattern, our ticket to creating **one-of-a-kind objects for which there is only one instance**. 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.



Developer: What use is that?

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

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

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

Developer: What downsides?

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

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

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

The Little Singleton

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

How would you create a single object?

`new MyObject();`

And, what if another object wanted to create a `MyObject`? Could it call `new` on `MyObject` again?

Yes, of course.

So as long as we have a class, can we always instantiate it one or more times?

Yes. Well, only if it's a public class.

And if not?

Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.

Hmm, interesting.

No, I'd never thought of it, but I guess it makes sense because it is a legal definition.

Did you know you could do this?

```
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.

Wow, you sure can.

Now can I instantiate a MyClass?

```
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

```

public class Singleton {
    private static Singleton uniqueInstance; ← We have a static variable to hold our one instance of the class Singleton.

    Let's rename MyClass to Singleton. ←

    // other useful instance variables here

    private Singleton() {} ← Our constructor is declared private; only Singleton can instantiate this class!

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

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

```

Watch it!

If you're just flipping through the book, don't blindly type in this code; you'll see it has a few issues later in the chapter.



Code Up Close

```

uniqueInstance holds our ONE instance; remember, it is a static variable. ← If uniqueInstance is null, then we haven't created the instance yet... ← ...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
}

return uniqueInstance; ← If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement.

By the time we hit this code, we have an instance and we return it. ←

```



Watch it!

If you're just flipping through the book, don't blindly type in this code; you'll see it has a few issues later in the chapter.



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 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!

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

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

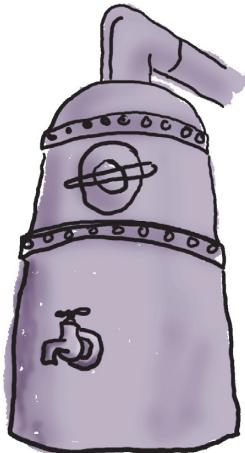
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    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.



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?

```
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.

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

The uniqueInstance class variable holds our one and only instance of Singleton.

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

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

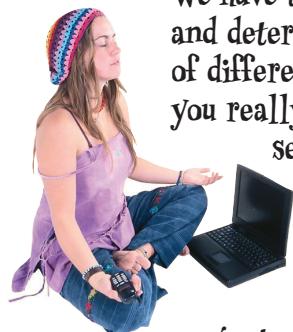
It looks like the Chocolate Boiler has let us down; despite the fact we improved the code using Classic Singleton, somehow the ChocolateBoiler's fill() method was able to start filling the boiler even though a batch of milk and chocolate was already boiling! That's 500 gallons of spilled milk (and chocolate)!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

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

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

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

```
}
```

```
    return uniqueInstance;
```

```
}
```

Make sure you check your answer on page 190 before continuing!

Thread
One

Thread
Two

Value of
`uniqueInstance`

Dealing with multithreading

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

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

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

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



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

Can we improve multithreading?

For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads. But, it is 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:

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

    private Singleton() {}

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

Annotations for the eager initialization code:

- A callout points to the line `private static Singleton uniqueInstance = new Singleton();` with the text: "Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!"
- A callout points to the line `return uniqueInstance;` with the text: "We've already got an instance, so just return it."

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

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

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

Let's check out the code:

```
public class Singleton {  
    private volatile* static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

*The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

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.

If performance is an issue in your use of the getInstance() method then this method of implementing the Singleton can drastically reduce the overhead.



Watch it!

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

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

Meanwhile, back at the Chocolate Factory...

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



Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

Use eager instantiation:

Double-checked locking:

Congratulations!

At this point, the Chocolate Factory is a happy customer and Choc-O-Holic was glad to have some expertise applied to their boiler code. No matter which multithreading solution you applied, the boiler should be in good shape with no more mishaps. Congratulations. 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.

there are no Dumb Questions

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 class loaders, 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.



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()` produces 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.

there are no Dumb Questions

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 you could argue it is taking on two responsibilities. Nevertheless, it isn't hard to see that there is utility in a class managing its own instance; it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

A: One problem with subclassing a Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected. But then, it's not really a Singleton anymore, because other classes can instantiate it.

If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing a registry of sorts is required in the base class.

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

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions. Do not depend on concrete classes.

OO Patterns

Factory Method Define a family of objects in a class or interface, and let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Singleton - Ensure a class only has one instance and provide a global point of access to it.

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.



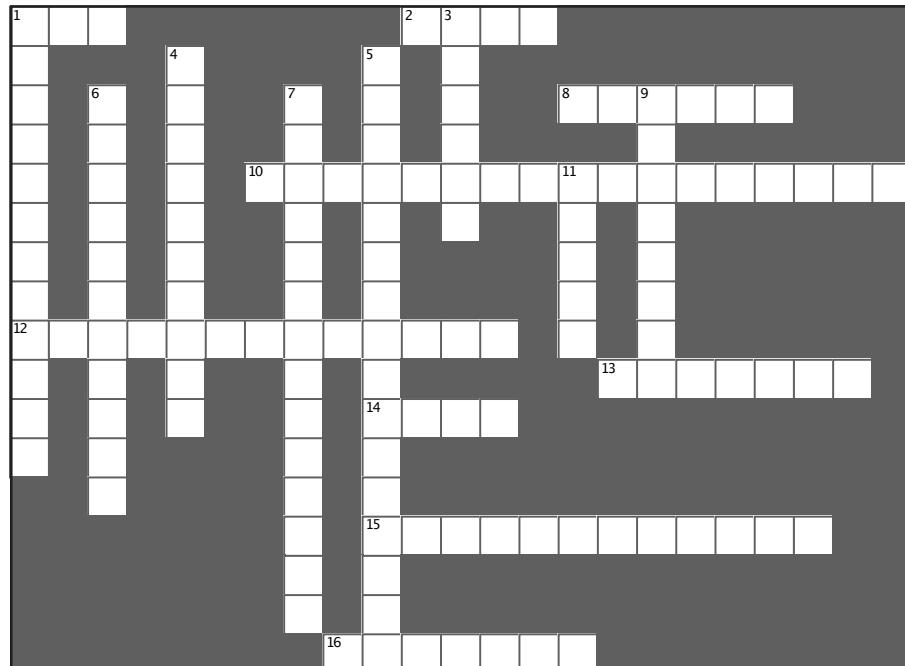
BULLET POINTS

- 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.



Design Patterns Crossword

Sit back, open that case of chocolate that you were sent for solving the multithreading problem, and have some downtime working on this little crossword puzzle; all of the solution words are from this chapter.



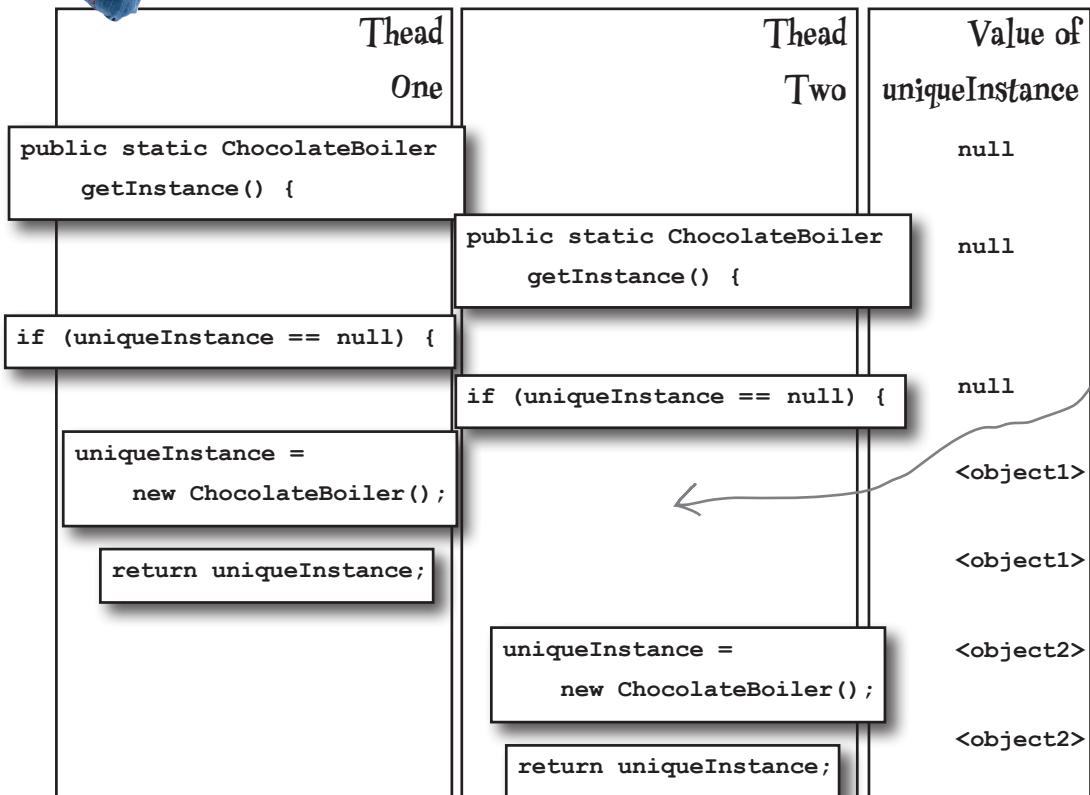
ACROSS

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 multi-threading approach if not using Java 5 or later.
13. Chocolate capital of the USA.
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 Java 1.2, this can eat your Singletons (two words).
6. The Singleton was embarrassed it had no public _____.
7. The classic implementation doesn’t handle this.
9. Singleton ensures only one of these exists.
11. The Singleton Pattern has one.

BE the JVM Solution





Sharpen your pencil Solution

Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

```
public class ChocolateBoiler {  
    private boolean empty;  
    private boolean boiled;  
  
    private static ChocolateBoiler uniqueInstance;  
  
    private ChocolateBoiler() {  
        empty = true;  
        boiled = false;  
    }  
  
    public static ChocolateBoiler getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new ChocolateBoiler();  
        }  
        return uniqueInstance;  
    }  
  
    public void fill() {  
        if (isEmpty()) {  
            empty = false;  
            boiled = false;  
            // fill the boiler with a milk/chocolate mixture  
        }  
    }  
    // rest of ChocolateBoiler code...  
}
```



Sharpen your pencil Solution

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

A straightforward technique that is guaranteed to work. We don't seem to have any performance concerns with the chocolate boiler, so this would be a good choice.

Use eager instantiation:

We are always going to instantiate the chocolate boiler in our code, so statically initializing the instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familiar 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.

A crossword puzzle grid with the following words filled in:

- Across:
 - 1. CAR
 - 4. S
 - 5. G
 - 7. M
 - 8. B O I L E R
 - 9. N
 - 10. G L O B A L A C
 - 11. C E S S P O I N T
 - 12. D O U B L E C H E C K E D
 - 13. H E R S H E Y
 - 14. L A Z Y
 - 15. C H O C - O - H O L I C
 - 16. P R I V A T E
- Down:
 - 1. L
 - 2. M I L K
 - 3. T
 - 4. A S
 - 5. S
 - 6. C
 - 7. U R E
 - 8. L T
 - 9. A A
 - 10. S N
 - 11. C E S S
 - 12. O R
 - 13. S N C
 - 14. R T Y E E L
 - 15. S O A D I N G
 - 16. C H O C - O - H O L I C



Design Patterns Crossword Solution

6 the Command Pattern

Encapsulating Invocation

These top secret drop boxes have revolutionized the spy industry. I just drop in my request and people disappear, governments change overnight and my dry cleaning gets done. I don't have to worry about when, where, or how; it just happens!



In this chapter, we take encapsulation to a whole new level: we're going to encapsulate method invocation. That's right; by encapsulating method invocation, we can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things, it just uses our crystallized method to get it done. We can also do some wickedly smart things with these encapsulated method invocations, like save them away for logging or reuse them to implement undo in our code.



Home Automation or Bust, Inc.
1221 Industrial Avenue, Suite 2000
Future City, IL 62914

Greetings!

I recently received a demo and briefing from Johnny Hurricane, CEO of Weather-O-Rama, on their new expandable weather station. I have to say, I was so impressed with the software architecture that I'd like to ask you to design the API for our new Home Automation Remote Control. In return for your services we'd be happy to handsomely reward you with stock options in Home Automation or Bust, Inc.

I'm enclosing a prototype of our ground-breaking remote control for your perusal. The remote control features seven programmable slots (each can be assigned to a different household device) along with corresponding on/off buttons for each. The remote also has a global undo button.

I'm also enclosing a set of Java classes on CD-R that were created by various vendors to control home automation devices such as lights, fans, hot tubs, audio equipment, and other similar controllable appliances.

We'd like you to create an API for programming the remote so that each slot can be assigned to control a device or set of devices. Note that it is important that we be able to control the current devices on the disc, and also any future devices that the vendors may supply.

Given the work you did on the Weather-O-Rama weather station, we know you'll do a great job on our remote control!

We look forward to seeing your design.

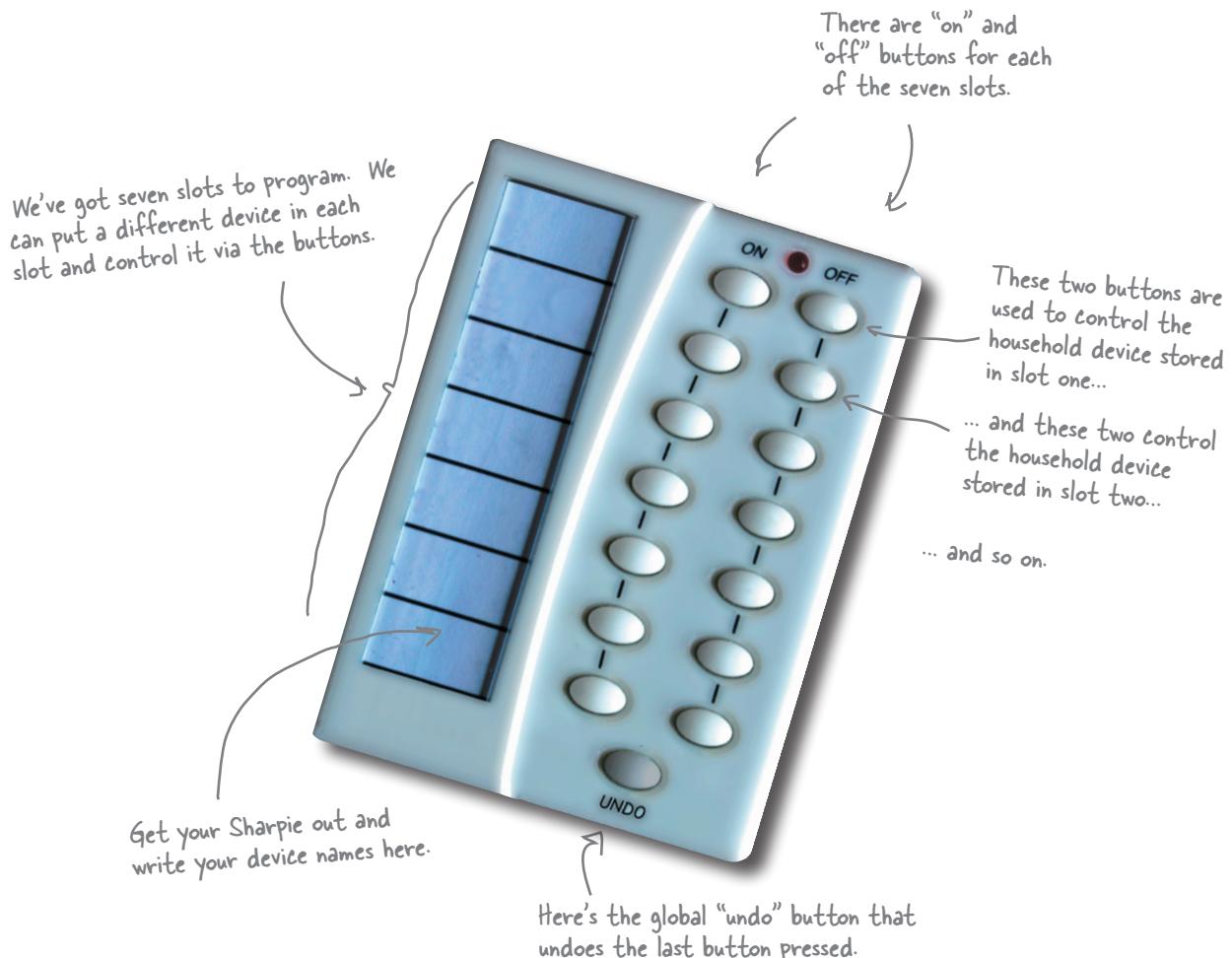
Sincerely,

Billy Thompson

Bill "X-10" Thompson, CEO

HOME AUTOMATION
VENDOR CLASSES

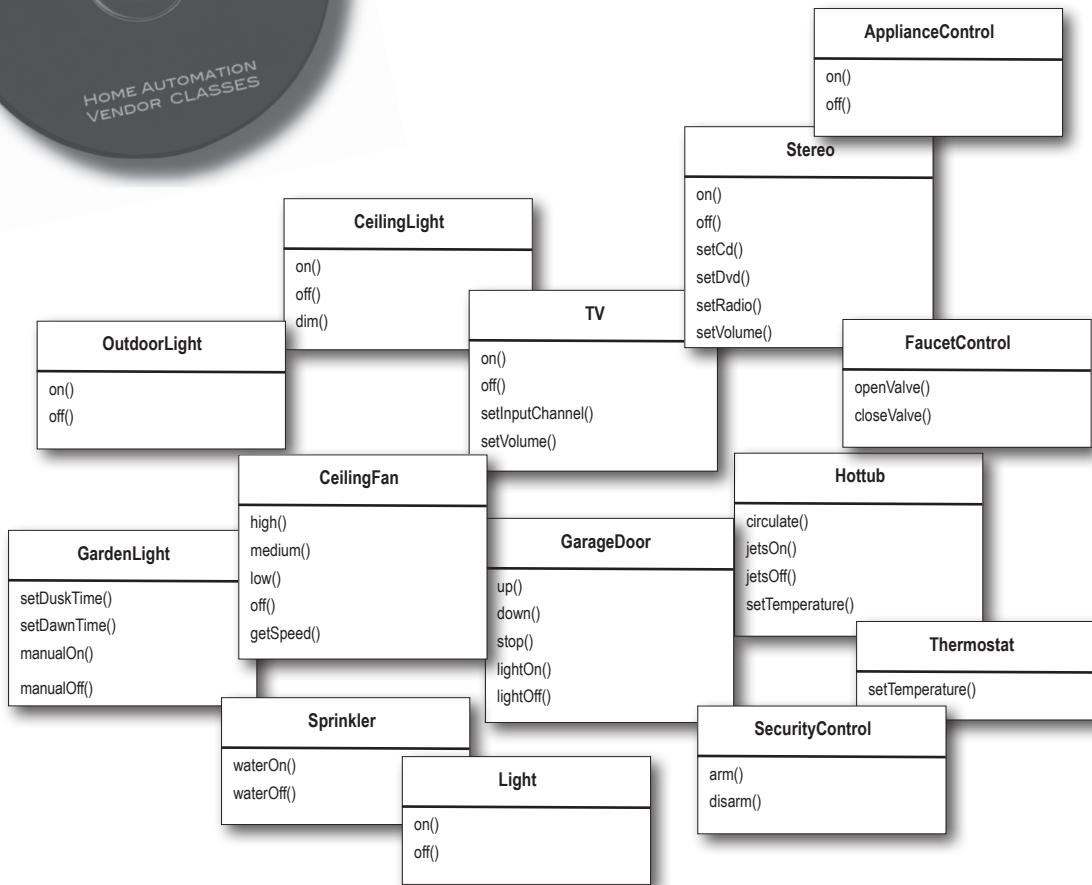
Free hardware! Let's check out the Remote Control...





Taking a look at the vendor classes

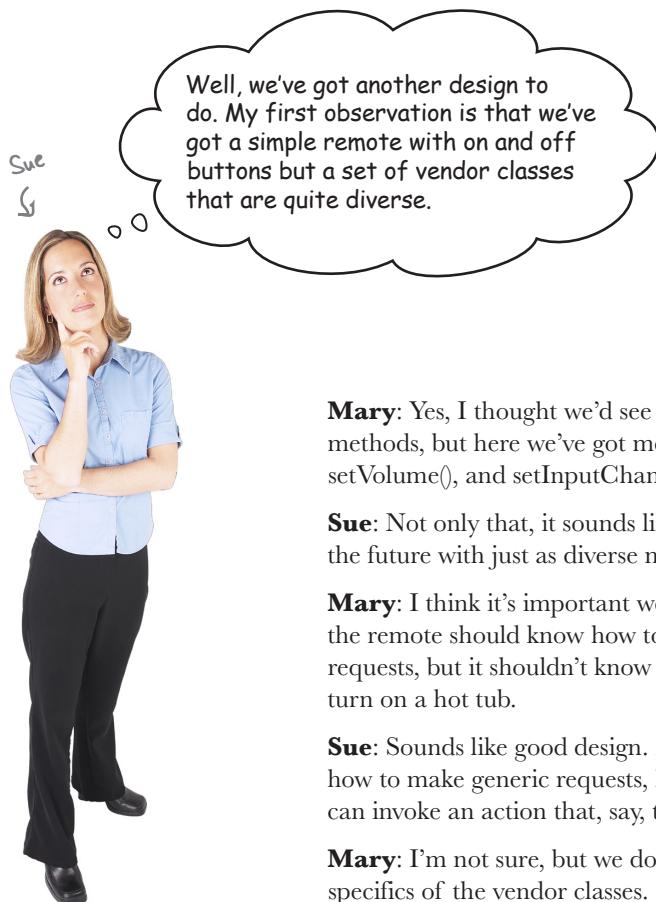
Check out the vendor classes on the CD-R. These should give you some idea of the interfaces of the objects we need to control from the remote.



It looks like we have quite a set of classes here, and not a lot of industry effort to come up with a set of common interfaces. Not only that, it sounds like we can expect more of these classes in the future. Designing a remote control API is going to be interesting. Let's get on to the design.

Cubicle Conversation

Your teammates are already discussing how to design the remote control API...



Mary: Yes, I thought we'd see a bunch of classes with on() and off() methods, but here we've got methods like dim(), setTemperature(), setVolume(), and setInputChannel().

Sue: Not only that, it sounds like we can expect more vendor classes in the future with just as diverse methods.

Mary: I think it's important we view this as a separation of concerns: the remote should know how to interpret button presses and make requests, but it shouldn't know a lot about home automation or how to turn on a hot tub.

Sue: Sounds like good design. But if the remote is dumb and just knows how to make generic requests, how do we design the remote so that it can invoke an action that, say, turns on a light or opens a garage door?

Mary: I'm not sure, but we don't want the remote to have to know the specifics of the vendor classes.

Sue: What do you mean?

Mary: We don't want the remote to consist of a set of if statements, like "if slot1 == Light, then light.on()", else if slot1 == Hottub then hottub.jetsOn()". We know that is a bad design.

Sue: I agree. Whenever a new vendor class comes out, we'd have to go in and modify the code, potentially creating bugs and more work for ourselves!



Mary: Yeah? Tell us more.

Joe: The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action. So, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

Sue: How is that possible? How can we decouple them? After all, when I press a button, the remote has to turn on a light.

Joe: You can do that by introducing “command objects” into your design. A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object). So, if we store a command object for each button, when the button is pressed we ask the command object to do some work. The remote doesn’t have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done. So, you see, the remote is decoupled from the light object!

Sue: This certainly sounds like it’s going in the right direction.

Mary: Still, I’m having a hard time wrapping my head around the pattern.

Joe: Given that the objects are so decoupled, it’s a little difficult to picture how the pattern actually works.

Mary: Let me see if I at least have the right idea: using this pattern, we could create an API in which these command objects can be loaded into button slots, allowing the remote code to stay very simple. And, the command objects encapsulate how to do a home automation task along with the object that needs to do it.

Joe: Yes, I think so. I also think this pattern can help you with that undo button, but I haven’t studied that part yet.

Mary: This sounds really encouraging, but I think I have a bit of work to do to really “get” the pattern.

Sue: Me too.

Meanwhile, back at the Diner... or, A brief introduction to the Command Pattern

As Joe said, it is a little hard to understand the Command Pattern by just hearing its description. But don't fear, we have some friends ready to help: remember our friendly diner from Chapter 1? It's been a while since we visited Alice, Flo, and the short-order cook, but we've got good reason for returning (well, beyond the food and great conversation): the diner is going to help us understand the Command Pattern.

So, let's take a short detour back to the diner and study the interactions between the customers, the waitress, the orders and the short-order cook. Through these interactions, you're going to understand the objects involved in the Command Pattern and also get a feel for how the decoupling works. After that, we're going to knock out that remote control API.

Checking in at the Objectville Diner...

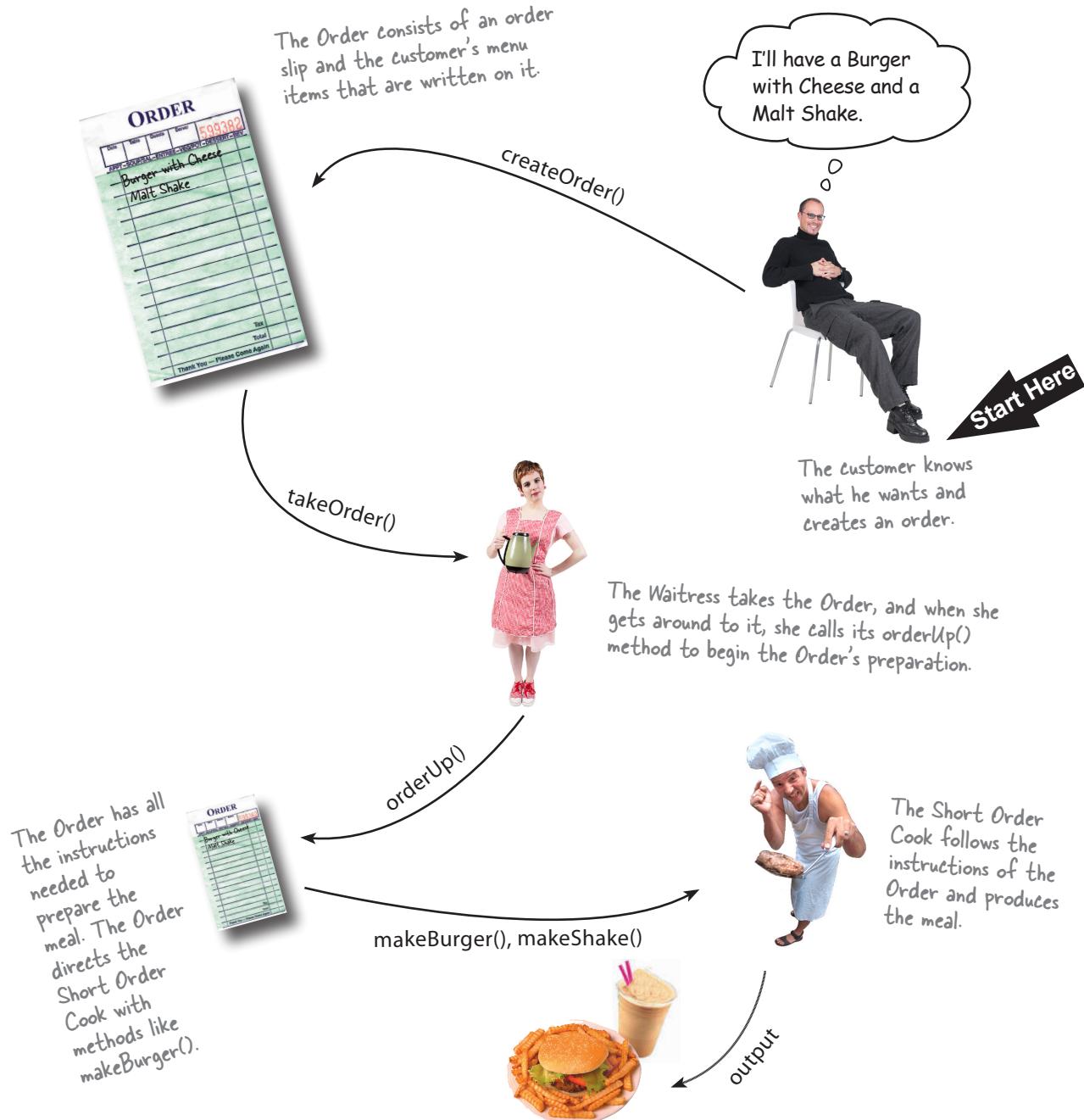


Okay, we all know how the Diner operates:



Let's study the interaction in a little more detail...

...and given this Diner is in Objectville, let's think about the object and method calls involved, too!



The Objectville Diner roles and responsibilities

An Order Slip encapsulates a request to prepare a meal.

Think of the Order Slip as an object, an object that acts as a request to prepare a meal. Like any object, it can be passed around—from the Waitress to the order counter, or to the next Waitress taking over her shift. It has an interface that consists of only one method, `orderUp()`, that encapsulates the actions needed to prepare the meal. It also has a reference to the object that needs to prepare it (in our case, the Cook). It's encapsulated in that the Waitress doesn't have to know what's in the order or even who prepares the meal; she only needs to pass the slip through the order window and call “Order up!”

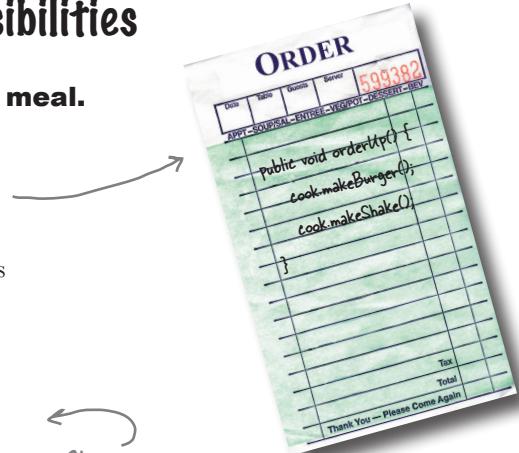
The Waitress's job is to take Order Slips and invoke the `orderUp()` method on them.

The Waitress has it easy: take an order from the customer, continue helping customers until she makes it back to the order counter, then invoke the `orderUp()` method to have the meal prepared. As we've already discussed, in Objectville, the Waitress really isn't worried about what's on the order or who is going to prepare it; she just knows Order Slips have an `orderUp()` method she can call to get the job done.

Now, throughout the day, the Waitress's `takeOrder()` method gets parameterized with different Order Slips from different customers, but that doesn't faze her; she knows all Order Slips support the `orderUp()` method and she can call `orderUp()` any time she needs a meal prepared.

The Short Order Cook has the knowledge required to prepare the meal.

The Short Order Cook is the object that really knows how to prepare meals. Once the Waitress has invoked the `orderUp()` method; the Short Order Cook takes over and implements all the methods that are needed to create meals. Notice the Waitress and the Cook are totally decoupled: the Waitress has Order Slips that encapsulate the details of the meal; she just calls a method on each order to get it prepared. Likewise, the Cook gets his instructions from the Order Slip; he never needs to directly communicate with the Waitress.



Okay, in real life a waitress would probably care what is on the Order Slip and who cooks it, but this is Objectville... work with us here!

Don't ask me to cook,
I just take orders and
yell "Order up!"





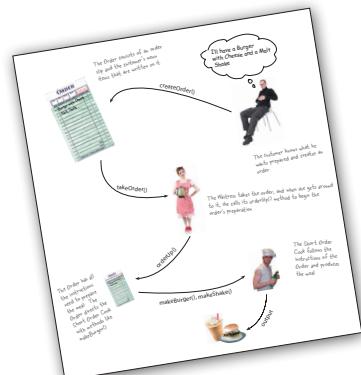
Patience, we're getting there...

Think of the Diner as a model for an OO design pattern that allows us to separate an object making a request from the objects that receive and execute those requests. For instance, in our remote control API, we need to separate the code that gets invoked when we press a button from the objects of the vendor-specific classes that carry out those requests. What if each slot of the remote held an object like the Diner's Order Slip object? Then, when a button is pressed, we could just call the equivalent of the "orderUp()" method on this object and have the lights turn on without the remote knowing the details of how to make those things happen or what objects are making them happen.

Now, let's switch gears a bit and map all this Diner talk to the Command Pattern...

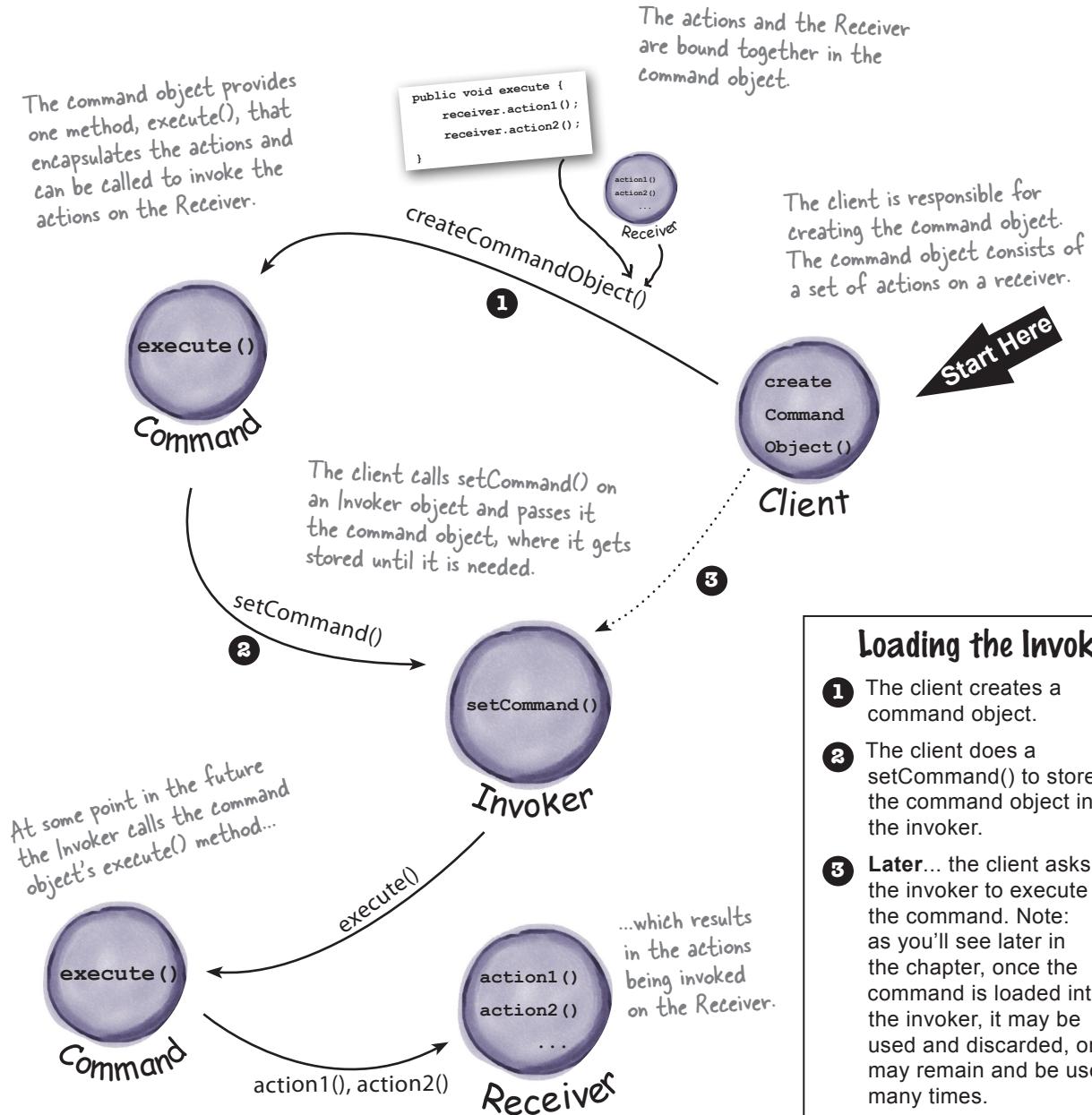
BRAIN POWER

Before we move on, spend some time studying the diagram two pages back along with Diner roles and responsibilities until you think you've got a handle on the Objectville Diner objects and relationships. Once you've done that, get ready to nail the Command Pattern!



From the Diner to the Command Pattern

Okay, we've spent enough time in the Objectville Diner that we know all the personalities and their responsibilities quite well. Now we're going to rework the Diner diagram to reflect the Command Pattern. You'll see that all the players are the same; only the names have changed.



Loading the Invoker

- 1 The client creates a command object.
- 2 The client does a `setCommand()` to store the command object in the invoker.
- 3 Later... the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.



Match the diner objects and methods with the corresponding names from the Command Pattern.

Diner

Waitress

Short Order Cook

orderUp()

Order

Customer

takeOrder()

Command Pattern

Command

execute()

Client

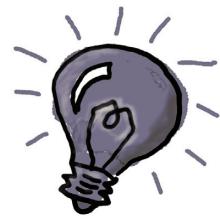
Invoker

Receiver

setCommand()

Our first command object

Isn't it about time we build our first command object? Let's go ahead and write some code for the remote control. While we haven't figured out how to design the remote control API yet, building a few things from the bottom up may help us...



Implementing the Command interface

First things first: all command objects implement the same interface, which consists of one method. In the Diner we called this method `orderUp()`; however, we typically just use the name `execute()`.

Here's the Command interface:

```
public interface Command {
    public void execute();
}
```

Simple. All we need is one method called `execute()`.

Implementing a command to turn a light on

Now, let's say you want to implement a command for turning a light on. Referring to our set of vendor classes, the Light class has two methods: `on()` and `off()`. Here's how you can implement this as a command:

Light
on()
off()

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

This is a command, so we need to implement the Command interface.

The constructor is passed the specific light that this command is going to control – say the living room light – and stashes it in the `light` instance variable. When `execute` gets called, this is the light object that is going to be the Receiver of the request.

The `execute` method calls the `on()` method on the receiving object, which is the light we are controlling.

Now that you've got a `LightOnCommand` class, let's see if we can put it to use...

Using the command object

Okay, let's make things simple: say we've got a remote control with only one button and corresponding slot to hold a device to control:

```
public class SimpleRemoteControl {
    Command slot;
    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control

Here's just a bit of code to test out the simple remote control. Let's take a look and we'll point out how the pieces match the Command Pattern diagram:

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

This is our Client in Command Pattern-speak.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

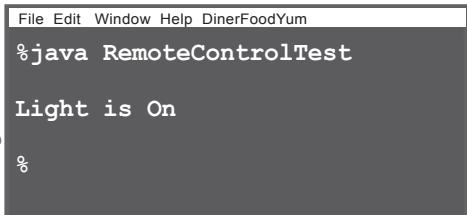
Now we create a Light object. This will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

Here, pass the command to the Invoker.

And then we simulate the button being pressed.

Here's the output of running this test code.

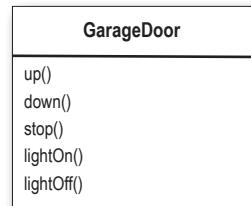


```
File Edit Window Help DinerFoodYum
%java RemoteControlTest
Light is On
%
```



Okay, it's time for you to implement the GarageDoorOpenCommand class. First, supply the code for the class below. You'll need the GarageDoor class diagram.

```
public class GarageDoorOpenCommand
    implements Command {
```



↖ Your code here

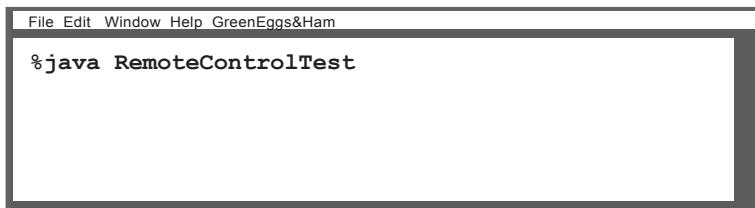
}

Now that you've got your class, what is the output of the following code? (Hint: the GarageDoor up() method prints out "Garage Door is Open" when it is complete.)

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

Your output here. ↗



The Command Pattern defined

You've done your time in the Objectville Diner, you've partly implemented the remote control API, and in the process you've got a fairly good picture of how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.

Let's start with its official definition:

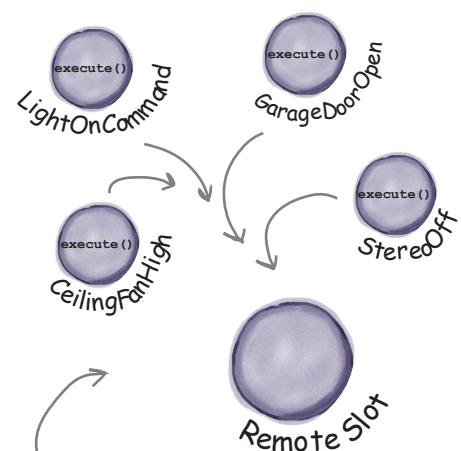
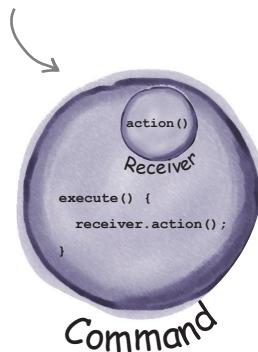
The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

Let's step through this. We know that a command object *encapsulates a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, `execute()`. When called, `execute()` causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the `execute()` method, their request will be serviced.

We've also seen a couple examples of *parameterizing an object* with a command. Back at the diner, the Waitress was parameterized with multiple orders throughout the day. In the simple remote control, we first loaded the button slot with a "light on" command and then later replaced it with a "garage door open" command. Like the Waitress, your remote slot didn't care what command object it had, as long as it implemented the Command interface.

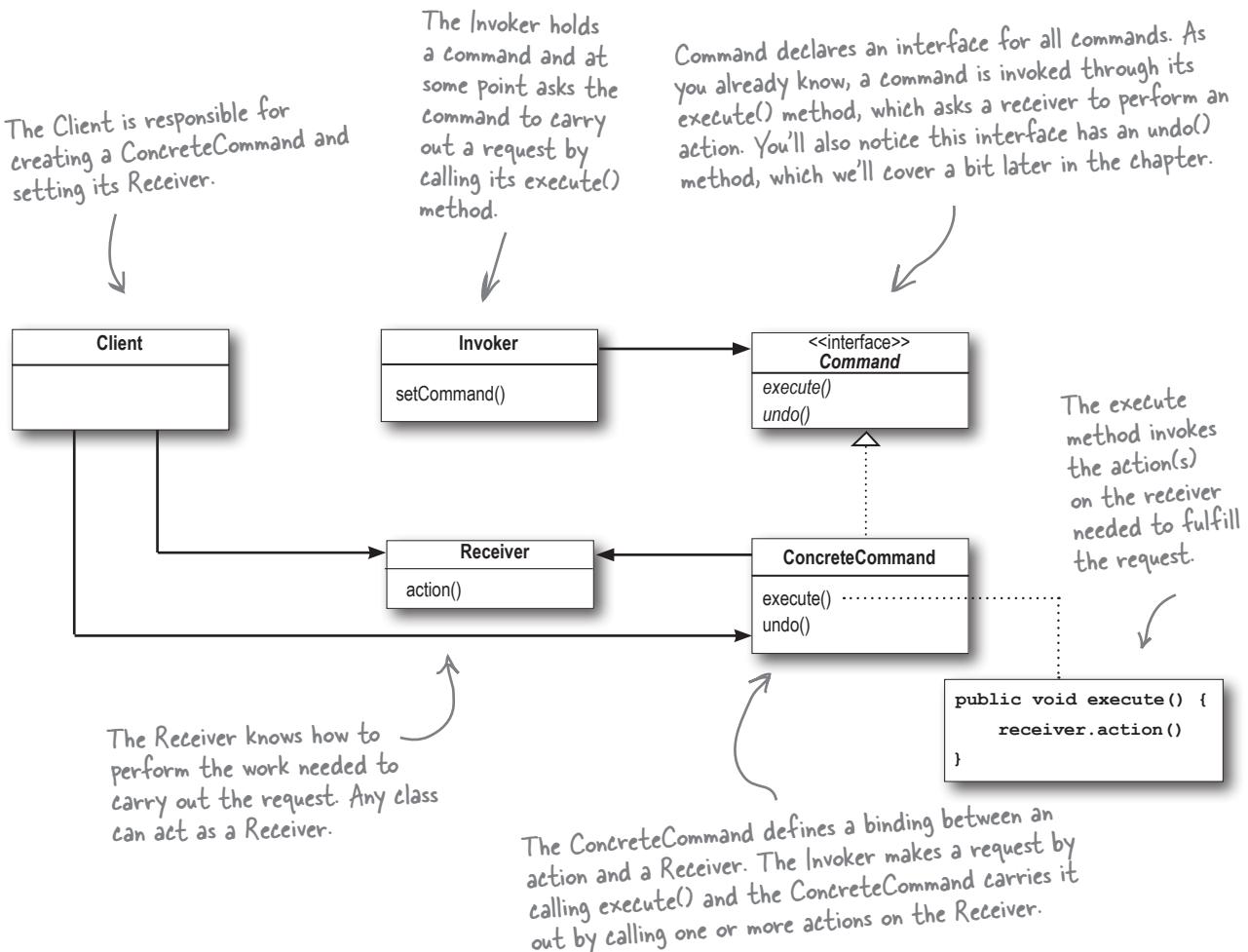
What we haven't encountered yet is using commands to implement *queues and logs and support undo operations*. Don't worry, those are pretty straightforward extensions of the basic Command Pattern and we will get to them soon. We can also easily support what's known as the Meta Command Pattern once we have the basics in place. The Meta Command Pattern allows you to create macros of commands so that you can execute multiple commands at once.

An encapsulated request.

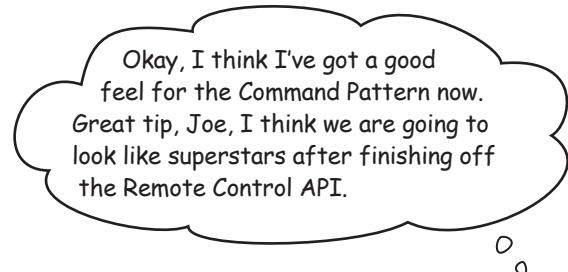


An invoker – for instance, one slot of the remote – can be parameterized with different requests.

The Command Pattern defined: the class diagram



How does the design of the Command Pattern support the decoupling of the invoker of a request and the receiver of the request?



Mary: Me too. So where do we begin?

Sue: Like we did in the SimpleRemote, we need to provide a way to assign commands to slots. In our case we have seven slots, each with an “on” and “off” button. So we might assign commands to the remote something like this:

```
onCommands[0] = onCommand;  
offCommands[0] = offCommand;
```

and so on for each of the seven command slots.

Mary: That makes sense, except for the Light objects. How does the remote know the living room from the kitchen light?

Sue: Ah, that's just it, it doesn't! The remote doesn't know anything but how to call execute() on the corresponding command object when a button is pressed.

Mary: Yeah, I sorta got that, but in the implementation, how do we make sure the right objects are turning on and off the right devices?

Sue: When we create the commands to be loaded into the remote, we create one LightCommand that is bound to the living room light object and another that is bound to the kitchen light object. Remember, the receiver of the request gets bound to the command it's encapsulated in. So, by the time the button is pressed, no one cares which light is which; the right thing just happens when the execute() method is called.

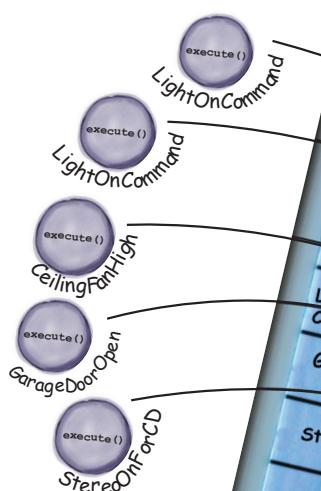
Mary: I think I've got it. Let's implement the remote and I think this will get clearer!

Sue: Sounds good. Let's give it a shot...

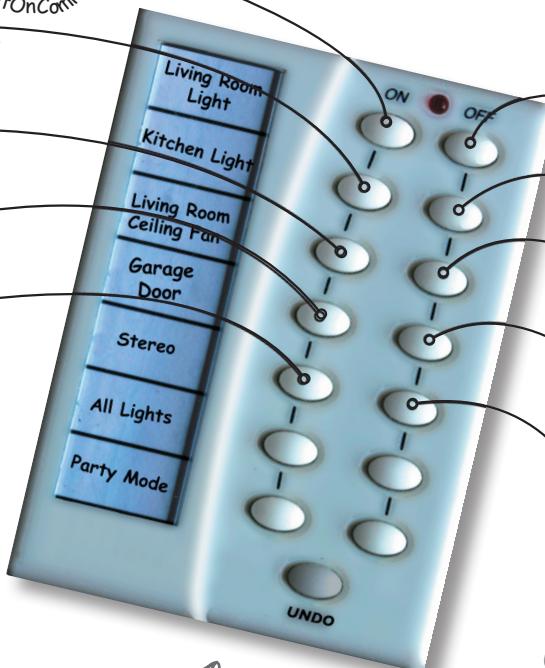
Assigning Commands to slots

So we have a plan: we're going to assign each slot to a command in the remote control. This makes the remote control our *invoker*. When a button is pressed the `execute()` method is going to be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans, and stereos).

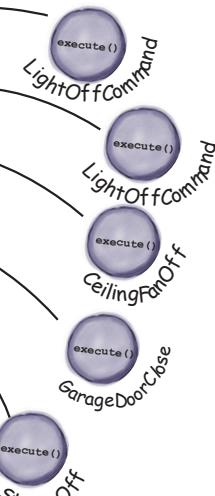
(1) Each slot gets a command.



We'll worry about the remaining slots in a bit.



(2) When the button is pressed, the `execute()` method is called on the corresponding command.



(3) In the `execute()` method actions are invoked on the receiver.



Implementing the Remote Control

```

public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;
}

public RemoteControl() {
    onCommands = new Command[7];
    offCommands = new Command[7];
}

Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
}

public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}

public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
}

public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
}

public String toString() {
    StringBuffer stringBuff = new StringBuffer();
    stringBuff.append("\n----- Remote Control -----");
    for (int i = 0; i < onCommands.length; i++) {
        stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
            + "      " + offCommands[i].getClass().getName() + "\n");
    }
    return stringBuff.toString();
}

```

This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot.

It puts these commands in the on and off arrays for later use.

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

We've overwritten `toString()` to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

Implementing the Commands

Well, we've already gotten our feet wet implementing the LightOnCommand for the SimpleRemoteControl. We can plug that same code in here and everything works beautifully. Off commands are no different; in fact, the LightOffCommand looks like this:

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Let's try something a little more challenging: how about writing on and off commands for the Stereo? Okay, off is easy, we just bind the Stereo to the off() method in the StereoOffCommand. On is a little more complicated; let's say we want to write a StereoOnWithCDCCommand...

Stereo
on()
off()
setCd()
setDvd()
setRadio()
setVolume()

```
public class StereoOnWithCDCCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

Not too bad. Take a look at the rest of the vendor classes; by now, you can definitely knock out the rest of the Command classes we need for those.

Putting the Remote Control through its paces

Our job with the remote is pretty much done; all we need to do is run some tests and get some documentation together to describe the API. Home Automation or Bust, Inc. sure is going to be impressed, don't ya think? We've managed to come up with a design that is going to allow them to produce a remote that is easy to maintain and they're going to have no trouble convincing the vendors to write some simple command classes in the future since they are so easy to write.

Let's get to testing this code!

```
public class RemoteLoader {  
  
    public static void main(String[] args) {  
        RemoteControl remoteControl = new RemoteControl();  
  
        Light livingRoomLight = new Light("Living Room");  
        Light kitchenLight = new Light("Kitchen");  
        CeilingFan ceilingFan = new CeilingFan("Living Room");  
        GarageDoor garageDoor = new GarageDoor("");  
        Stereo stereo = new Stereo("Living Room");  
  
        LightOnCommand livingRoomLightOn =  
            new LightOnCommand(livingRoomLight);  
        LightOffCommand livingRoomLightOff =  
            new LightOffCommand(livingRoomLight);  
        LightOnCommand kitchenLightOn =  
            new LightOnCommand(kitchenLight);  
        LightOffCommand kitchenLightOff =  
            new LightOffCommand(kitchenLight);  
  
        CeilingFanOnCommand ceilingFanOn =  
            new CeilingFanOnCommand(ceilingFan);  
        CeilingFanOffCommand ceilingFanOff =  
            new CeilingFanOffCommand(ceilingFan);  
  
        GarageDoorUpCommand garageDoorUp =  
            new GarageDoorUpCommand(garageDoor);  
        GarageDoorDownCommand garageDoorDown =  
            new GarageDoorDownCommand(garageDoor);  
  
        StereoOnWithCDCCommand stereoOnWithCD =  
            new StereoOnWithCDCCommand(stereo);  
        StereoOffCommand stereoOff =  
            new StereoOffCommand(stereo);  
    }  
}
```

Diagram illustrating the grouping of code blocks by their purpose:

- Group 1: `Light`, `CeilingFan`, `GarageDoor`, `Stereo` objects. Bracketed note: Create all the devices in their proper locations.
- Group 2: `LightOnCommand`, `LightOffCommand`, `LightOnCommand`, `LightOffCommand`. Bracketed note: Create all the Light Command objects.
- Group 3: `CeilingFanOnCommand`, `CeilingFanOffCommand`. Bracketed note: Create the On and Off for the ceiling fan.
- Group 4: `GarageDoorUpCommand`, `GarageDoorDownCommand`. Bracketed note: Create the Up and Down commands for the Garage.
- Group 5: `StereoOnWithCDCCommand`, `StereoOffCommand`. Bracketed note: Create the stereo On and Off commands.

```
remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

System.out.println(remoteControl);
```

Now that we've got all our commands, we can load them into the remote slots.

```
remoteControl.onButtonWasPushed(0);
remoteControl.offButtonWasPushed(0);
remoteControl.onButtonWasPushed(1);
remoteControl.offButtonWasPushed(1);
remoteControl.onButtonWasPushed(2);
remoteControl.offButtonWasPushed(2);
remoteControl.onButtonWasPushed(3);
remoteControl.offButtonWasPushed(3);
```

Here's where we use our `toString()` method to print each remote slot and the command that it is assigned to.

All right, we are ready to roll! Now, we step through each slot and push its On and Off button.

Now, let's check out the execution of our remote control test...

```
File Edit Window Help CommandsGetThingsDone

% java RemoteLoader
----- Remote Control -----
[slot 0] LightOnCommand           LightOffCommand
[slot 1] LightOnCommand           LightOffCommand
[slot 2] CeilingFanOnCommand     CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand   StereoOffCommand
[slot 4] NoCommand                NoCommand
[slot 5] NoCommand                NoCommand
[slot 6] NoCommand                NoCommand




On slots      Off slots


```

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off

← Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living Room light is on."



Wait a second, what is with that NoCommand that is loaded in slots four through six? Trying to pull a fast one?

Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the `onButtonWasPushed()` method, we would need code like this:

```
public void onButtonWasPushed(int slot) {  
    if (onCommands[slot] != null) {  
        onCommands[slot].execute();  
    }  
}
```

So, how do we get around that? Implement a command that does nothing!

```
public class NoCommand implements Command {  
    public void execute() {}  
}
```

Then, in our `RemoteControl` constructor, we assign every slot a `NoCommand` object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();  
for (int i = 0; i < 7; i++) {  
    onCommands[i] = noCommand;  
    offCommands[i] = noCommand;  
}
```

So in the output of our test run, you are seeing only slots that have been assigned to a command other than the default `NoCommand` object, which we assigned when we created the `RemoteControl`.



Pattern Honorable Mention

The `NoCommand` object is an example of a *null object*. A *null object* is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling `null` from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a `NoCommand` object that acts as a surrogate and does nothing when its `execute` method is called.

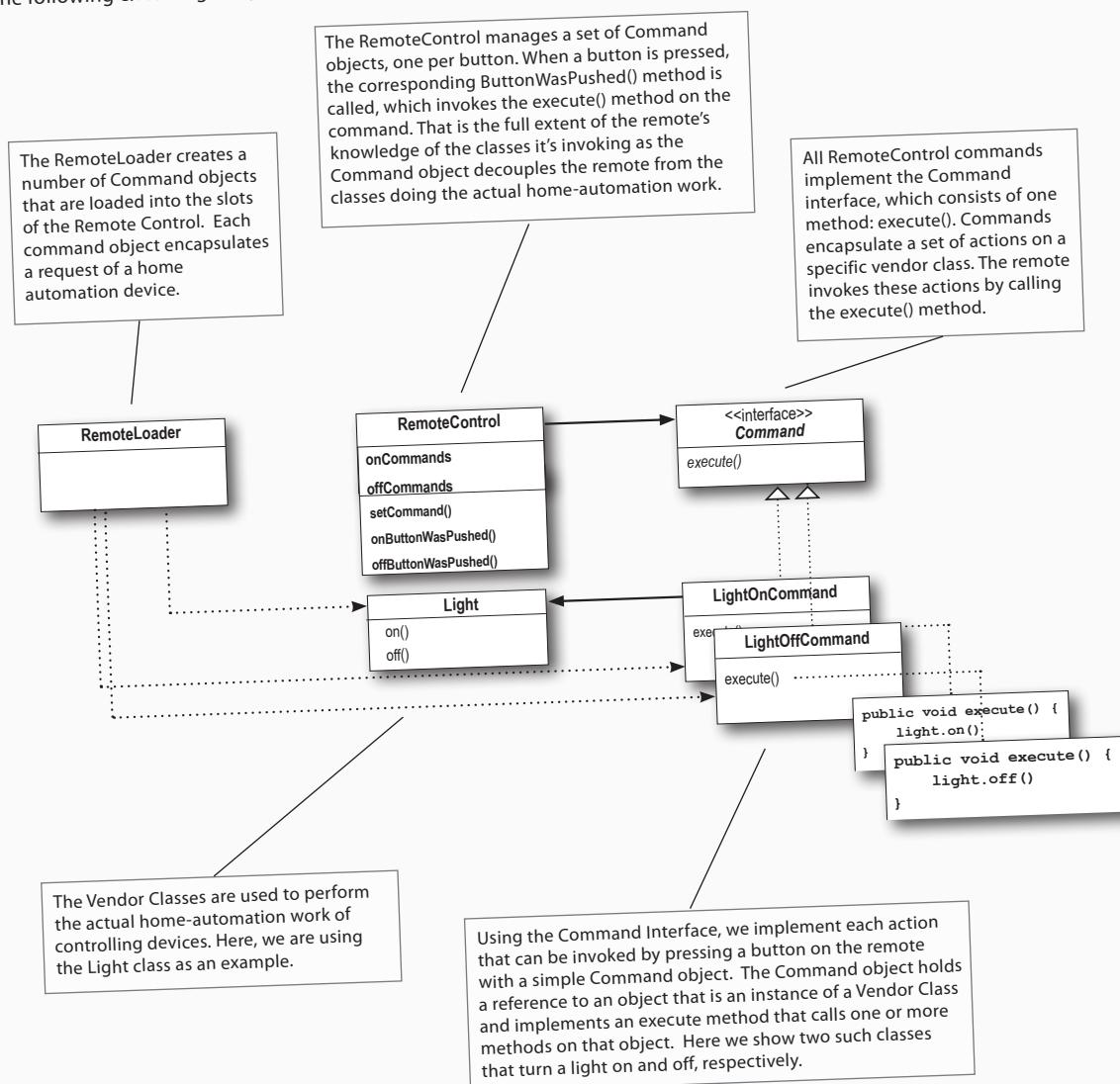
You'll find uses for Null Objects in conjunction with many Design Patterns and sometimes you'll even see Null Object listed as a Design Pattern.

Time to write that documentation...

Remote Control API Design for Home Automation or Bust, Inc.

We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the Command Pattern to logically decouple the RemoteControl class from the Vendor Classes. We believe this will reduce the cost of producing the remote as well as drastically reduce your ongoing maintenance costs.

The following class diagram provides an overview of our design:





Whoops! We almost forgot... luckily, once we have our basic Command classes, undo is easy to add. Let's step through adding undo to our commands and to the remote control...

What are we doing?

Okay, we need to add functionality to support the undo button on the remote. It works like this: say the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button then the last action will be reversed—in this case, the light will turn off. Before we get into more complex examples, let's get the light working with the undo button:

- ➊ When commands support undo, they have an `undo()` method that mirrors the `execute()` method. Whatever `execute()` last did, `undo()` reverses. So, before we can add undo to our commands, we need to add an `undo()` method to the Command interface:

```
public interface Command {  
    public void execute();  
    public void undo();  
}
```

↗ Here's the new `undo()` method.

That was simple enough.

Now, let's dive into the Light command and implement the `undo()` method.

- 2 Let's start with the LightOnCommand: if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() method.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();    ←
    }
}
```

execute() turns the light
on, so undo() simply turns
the light back off.

Piece of cake! Now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();    ←
    }
}
```

And here, undo() turns
the light back on.

Could this be any easier? Okay, we aren't done yet; we need to work a little support into the Remote Control to handle tracking the last button pressed and the undo button press.

- 3 To add support for the undo button we only have to make a few small changes to the Remote Control class. Here's how we're going to do it: we'll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.

```

public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // toString code here...
    }
}

```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

Time to QA that Undo button!

Okay, let's rework the test harness a bit to test the undo button:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room"); ← Create a Light, and our new undo()
        ← enabled Light On and Off Commands.

        LightOnCommand livingRoomLightOn =
            new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
            new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

And here are the test results...

```
File Edit Window Help UndoCommandsDefyEntropy
% java RemoteLoader
Light is on ← Turn the light on, then off.
Light is off

----- Remote Control -----
[slot 0] LightOnCommand      LightOffCommand ← Here are the Light commands.
[slot 1] NoCommand           NoCommand
[slot 2] NoCommand           NoCommand
[slot 3] NoCommand           NoCommand
[slot 4] NoCommand           NoCommand
[slot 5] NoCommand           NoCommand
[slot 6] NoCommand           NoCommand
[undo] LightOffCommand

Light is on ← Undo was pressed... the LightOffCommand
← undo() turns the light back on.
Light is off ← Then we turn the light off then back on.

----- Remote Control -----
[slot 0] LightOnCommand      LightOffCommand
[slot 1] NoCommand           NoCommand
[slot 2] NoCommand           NoCommand
[slot 3] NoCommand           NoCommand
[slot 4] NoCommand           NoCommand
[slot 5] NoCommand           NoCommand
[slot 6] NoCommand           NoCommand
[undo] LightOnCommand

Light is off ← Undo was pressed, the light is back off. ← Now undo holds the LightOnCommand, the last
← command invoked.
```

Using state to implement Undo

Okay, implementing undo on the Light was instructive but a little too easy. Typically, we need to manage a bit of state to implement undo. Let's try something a little more interesting, like the CeilingFan from the vendor classes. The CeilingFan allows a number of speeds to be set along with an off method.

Here's the source code for the CeilingFan:

```
public class CeilingFan {  
    public static final int HIGH = 3;  
    public static final int MEDIUM = 2;  
    public static final int LOW = 1;  
    public static final int OFF = 0;  
    String location;  
    int speed;  
  
    public CeilingFan(String location) {  
        this.location = location;  
        speed = OFF;  
    }  
  
    public void high() {  
        speed = HIGH;  
        // code to set fan to high  
    }  
  
    public void medium() {  
        speed = MEDIUM;  
        // code to set fan to medium  
    }  
  
    public void low() {  
        speed = LOW;  
        // code to set fan to low  
    }  
  
    public void off() {  
        speed = OFF;  
        // code to turn fan off  
    }  
  
    public int getSpeed() {  
        return speed;  
    }  
}
```

CeilingFan
high()
medium()
low()
off()
getSpeed()

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

Hmm, so to properly implement undo, I'd have to take the previous speed of the ceiling fan into account...

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().



Adding Undo to the CeilingFan commands

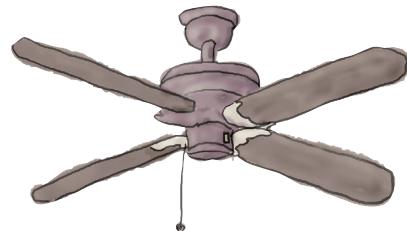
Now let's tackle adding undo to the various CeilingFan commands. To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting. Here's the code for the CeilingFanHighCommand:

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```



We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.



We've got three more ceiling fan commands to write: low, medium, and off. Can you see how these are implemented?

Get ready to test the ceiling fan

Time to load up our remote control with the ceiling fan commands. We're going to load slot 0's on button with the medium setting for the fan and slot 1 with the high setting. Both corresponding off buttons will hold the ceiling fan off command.

Here's our test script:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

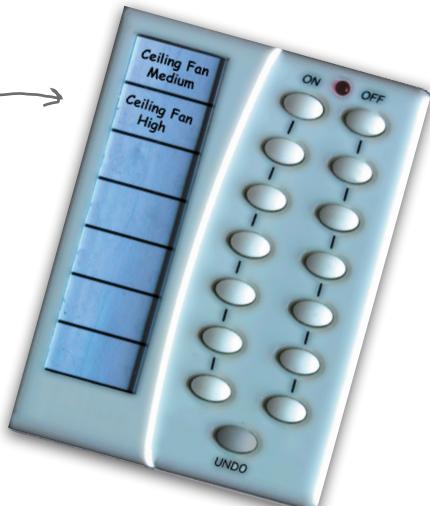
        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
            new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
            new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
            new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

        remoteControl.onButtonWasPushed(0);           ← First, turn the fan on medium.
        remoteControl.offButtonWasPushed(0);          ← Then turn it off.
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();          ← Undo! It should go back to medium...

        remoteControl.onButtonWasPushed(1);           ← Turn it on to high this time.
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();          ← And, one more undo; it should go back
                                                    to medium.
    }
}
```



Here we instantiate three commands: high, medium, and off.

Here we put medium in slot 0, and high in slot 1. We also load up the off command.

First, turn the fan on medium.

Then turn it off.

Undo! It should go back to medium...

Turn it on to high this time.

And, one more undo; it should go back to medium.

Testing the ceiling fan...

Okay, let's fire up the remote, load it with commands, and push some buttons!

```
File Edit Window Help UndoThis!
% java RemoteLoader

Living Room ceiling fan is on medium ← Turn the ceiling fan on
Living Room ceiling fan is off medium, then turn it off.

----- Remote Control -----
[slot 0] CeilingFanMediumCommand CeilingFanOffCommand
[slot 1] CeilingFanHighCommand CeilingFanOffCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] CeilingFanOffCommand ← Here are the commands
                                         in the remote control...

...and undo has the last command
executed, the CeilingFanOffCommand,
with the previous speed of medium.

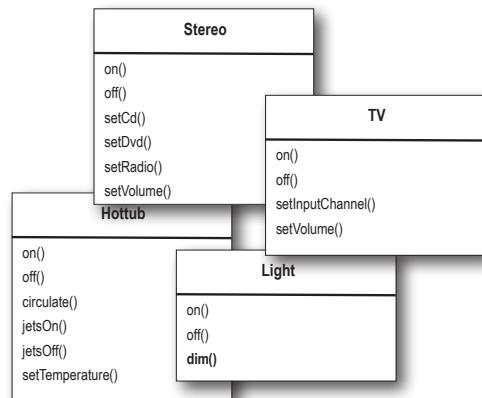
Living Room ceiling fan is on medium ← Undo the last command, and it goes back to medium.
Living Room ceiling fan is on high ← Now, turn it on high.

----- Remote Control -----
[slot 0] CeilingFanMediumCommand CeilingFanOffCommand
[slot 1] CeilingFanHighCommand CeilingFanOffCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] CeilingFanHighCommand ← Now, high is the last
                                         command executed.

Living Room ceiling fan is on medium
%
← One more undo, and the ceiling
fan goes back to medium speed.
```

Every remote needs a Party Mode!

What's the point of having a remote if you can't push one button and have the lights dimmed, the stereo and TV turned on and set to a DVD, and the hot tub fired up?



Mary's idea is to make a new kind of Command that can execute other Commands... and more than one of them! Pretty good idea, huh?

```

public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
  
```

Take an array of Commands and store them in the MacroCommand.

When the macro gets executed by the remote, execute those commands one at a time.

Using a macro command

Let's step through how we use a macro command:

- First we create the set of commands we want to go into the macro:

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Create all the devices: a light, tv, stereo, and hot tub.

Now create all the On commands to control them.



Sharpen your pencil

We will also need commands for the off buttons.
Write the code to create those here:

- Next we create two arrays, one for the On commands and one for the Off commands, and load them with the corresponding commands:

Create an array for On and an array for Off commands...

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...and create two corresponding macros to hold them.

- Then we assign MacroCommand to a button like we always do:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Assign the macro command to a button as we would any command.

test drive the macro command

- ④ Finally, we just need to push some buttons and see if this works.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

Here's the output.

```
File Edit Window Help You Can'tBeatABabka
% java RemoteLoader
----- Remote Control -----
[slot 0] MacroCommand    MacroCommand
[slot 1] NoCommand        NoCommand
[slot 2] NoCommand        NoCommand
[slot 3] NoCommand        NoCommand
[slot 4] NoCommand        NoCommand
[slot 5] NoCommand        NoCommand
[slot 6] NoCommand        NoCommand
[undo] NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees
```

Here are the two macro commands.

All the Commands in the macro are executed when we invoke the on macro...

and when we invoke the off macro. Looks like it works.



The only thing our MacroCommand is missing is its undo functionality. When the undo button is pressed after a macro command, all the commands that were invoked in the macro must undo their previous actions. Here's the code for MacroCommand; go ahead and implement the undo() method:

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        // Implementation goes here
    }
}
```

there are no Dumb Questions

Q: Do I always need a receiver? Why can't the command object implement the details of the execute() method?

A: In general, we strive for “dumb” command objects that just invoke an action on a receiver; however, there are many examples of “smart” command objects that implement most, if not all, of the logic needed to carry out a request. Certainly you can do this; just keep in mind you'll no longer have the same level of decoupling between the invoker and receiver, nor will you be able to parameterize your commands with receivers.

Q: How can I implement a history of undo operations? In other words, I want to be able to press the undo button multiple times?

A: Great question. It's pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

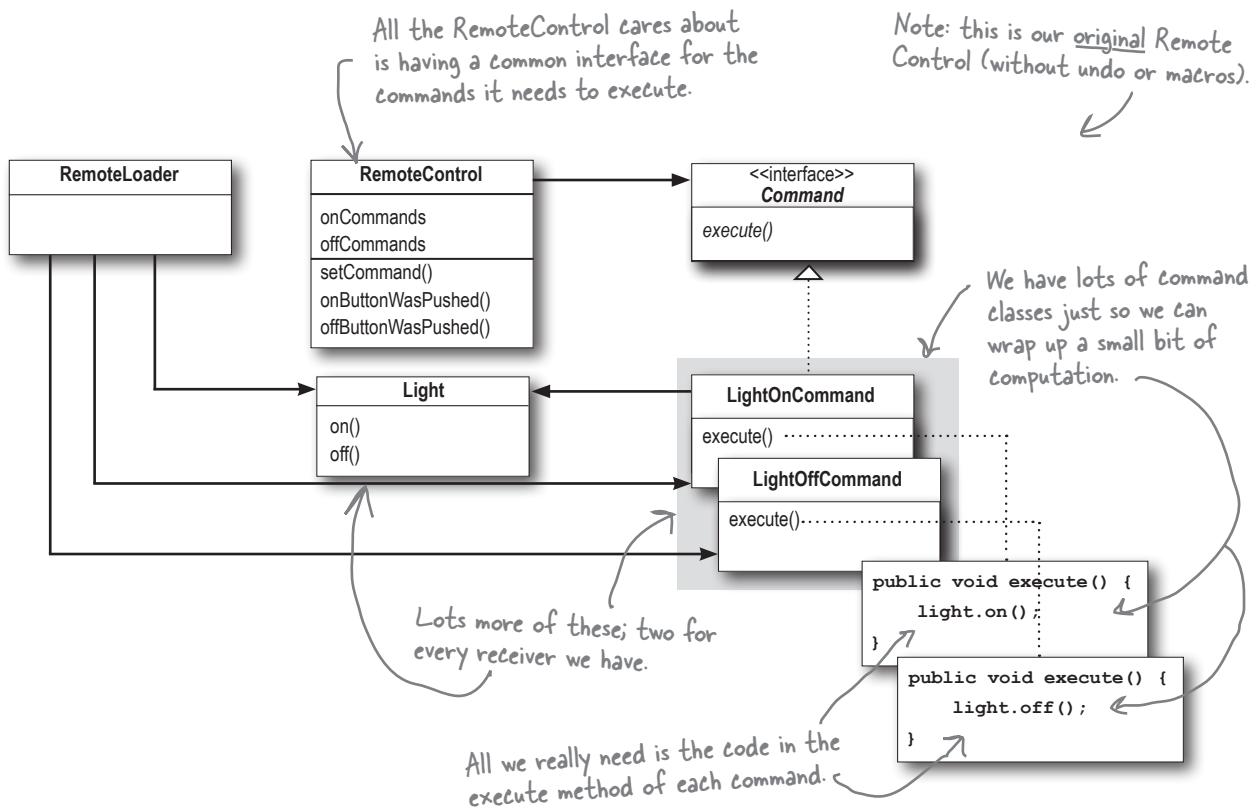
Q: Could I have just implemented party mode as a Command by creating a PartyCommand and putting the calls to execute the other Commands in the PartyCommand's execute() method?

A: You could; however, you'd essentially be “hardcoding” the party mode into the PartyCommand. Why go to the trouble? With the MacroCommand, you can decide dynamically which Commands you want to go into the PartyCommand, so you have more flexibility using MacroCommands. In general, the MacroCommand is a more elegant solution and requires less new code.

can we reduce the number of command classes?

The Command Pattern means lots of command classes

When you use the Command Pattern, you end up with a lot of small classes—the concrete Command implementations—that each encapsulate the request to the corresponding receiver. In our remote control implementation, we have two command classes for each receiver class. For instance, for the Light receiver, we have LightOnCommand and LightOffCommand; for the GarageDoor receiver, we have GarageDoorUpCommand and GarageDoorDownCommand, and so on. That's a lot of extra stuff that's needed to create little bits of packaged-up computation that all have the same interface for the RemoteControl:



Do we really need all these command classes?

A command is simply a piece of packaged-up computation. It's a way for us to have a common interface to the behavior of many different receivers (lights, hot tubs, stereos) each with its own set of actions.

What if you could keep the common interface for all your commands, but take out the bits of computation from inside the concrete Command implementations and use them directly instead? And you could get rid of all those extra classes and simplify your code? Well, with lambda expressions you can. Let's see how...

Simplifying the Remote Control with lambda expressions

While you've seen how straightforward the Command Pattern is, Java gives us a nice tool to ← simplify things even more; namely, the lambda expression. A lambda expression is a short hand for a method—a bit of computation—exactly where you need it. Instead of creating a whole separate class containing that method, instantiating an object from that class, and then calling the method, you can just say, "here's the method I want called" by using a lambda expression. In our case, the method we want called is the execute() method.

Let's replace the LightOnCommand and LightOffCommand objects with lambda expressions to see how this works. Here are the steps to use lambda expressions instead of command objects to add the light on and off commands to the remote control:

Step 1: Create the Receiver

This step is exactly the same as before.

```
Light livingRoomLight = new Light("Living Room");
```

Light
on()
off()

If you aren't yet familiar with lambda expressions (they were added in Java 8) they can take some getting used to. You should be able to follow along over the next few pages, but consult a Java reference to get up to speed on the syntax and semantics if you need to.

Step 2: Set the remote control's commands using lambda expressions

This is where the magic happens. Now, instead of creating LightOnCommand and LightOffCommand objects to pass to remoteControl.setCommand(), we simply pass a lambda expression in place of each object, with the code from their respective execute() methods:

```
remoteControl.setCommand(0, () -> { livingRoomLight.on(); }, () -> { livingRoomLight.off(); })
```

The lambdas get passed as commands to setCommand.

```
public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
```

Here are the two lambda expressions.

Step 3: Push the remote control buttons

This step doesn't change either. Except now, when we call the remote's onButtonWasPushed(0) method, the command that's in slot 0 is a function object (created by the lambda expression). When we call execute() on the command, that method is matched up with the method defined by the lambda expression, which is then executed.

```
remoteControl.onButtonWasPushed(0);
```

What's stored in the onCommands array at slot 0 is the lambda expression we passed to setCommand in Step 2. The execute() method is matched to the method in the lambda expression, and executed.

```
public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
}
```

```
() -> { livingRoomLight.on(); }
```

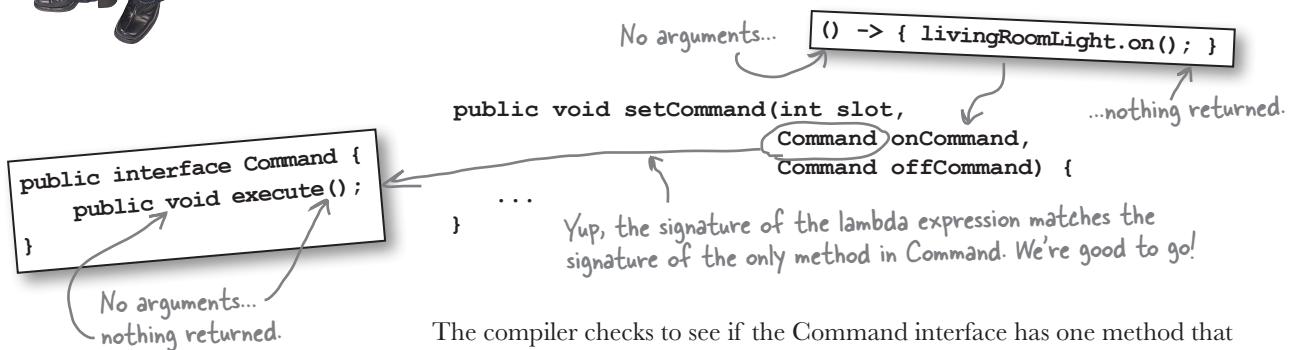


Are you trying to pull another fast one?
The lambda expression we're passing into
the setCommand method doesn't even
have an execute method. So how does the
method in the lambda ever get called?

Well, we did say “magic” didn’t we?

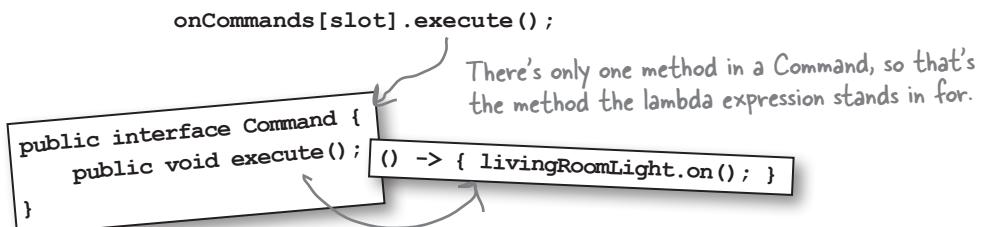
Just kidding... it's actually not all that magical. We're using lambda expressions to stand in for Command objects, and the Command interface has just one method: execute(). The lambda expression we use must have a compatible signature with this method—and it does: execute() takes no arguments (neither does our lambda expression), and returns no value (neither does our lambda expression), so the compiler is happy.

We pass the lambda expression into the Command parameter of the setCommand() method:



The compiler checks to see if the Command interface has one method that matches the lambda expression, and it does: execute().

Then, when we call execute() on that command, the method in the lambda expression is called:



Just remember: as long as the interface of the parameter we're passing the lambda expression to has one (and only one!) method, and that method has a compatible signature with the lambda expression, this will work.

Simplifying even more with method references

We can simplify our code even more using *method references*. When the lambda expression you're passing calls just one method, you can pass a method reference in place of the lambda expression. Like this:

```
remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);
```

This is a reference to the on() method
of the livingRoomLight object.

This is a reference to the off()
method of the livingRoomLight object.

So now, instead of passing a lambda expression that calls the livingRoomLight's on() method, we're passing a *reference to the method itself*.

What if we need to do more than one thing in our lambda expression?

Sometimes, the lambda expressions you'll use to stand in for Command objects have to do more than one thing. Let's take a quick look at how to replace the stereoOnWithCDCCommand and stereoOffCommand objects with lambda expressions, and then we'll look at the complete code for the RemoteLoader so you can see all these ideas come together.

The stereoOffCommand just executes a simple one-line command:

```
stereo.off();
```

So we can use a method reference, `stereo::off`, in place of a lambda expression for this command.

But the stereoOnWithCDCCommand does *three* things:

```
stereo.on();
stereo.setCD();
stereo.setVolume(11);
```

In this case, then, we can't use a method reference. Instead, we can either write the lambda expression in line, or we can create it separately, give it a name, and then pass it to the remoteControl's setCommand() method using that name. Here's how you can create the lambda expression separately, and give it a name:

```
Command stereoOnWithCD = () -> {
    stereo.on(); stereo.setCD(); stereo.setVolume(11);
};

remoteControl.setCommand(3, stereoOnWithCD, stereo::off);
```

This lambda expression does three things
(just like the stereoOnWithCDCCommand's
execute() method did).

We can pass the lambda
expression using its name.

Notice that we use Command as the type of the lambda expression. The lambda expression will match the Command interface's execute() method, and the Command parameter we're passing it to in the setCommand() method.

Test the remote control with lambda expressions

To use lambda expressions to simplify the code for the original Remote Control implementation (without undo), we're going to change the code in the RemoteLoader to replace the concrete Command objects with lambda expressions, and change the RemoteControl constructor to use lambda expressions instead of a NoCommand object. Once we've done that, we can delete all the concrete Command classes (LightOnCommand, LightOffCommand, HottubOnCommand, HottubOffCommand, and so on). And that's it. Everything else stays the same. Make sure you *don't* delete the Command interface; you still need that to match the type of the function objects created by the lambda expressions that get stored in the remote control, and passed to the various methods.

Here's the new code for the RemoteLoader class:

```
public class RemoteLoader {  
    public static void main(String[] args) {  
        RemoteControl remoteControl = new RemoteControl();  
  
        Light livingRoomLight = new Light("Living Room");  
        Light kitchenLight = new Light("Kitchen");  
        CeilingFan ceilingFan = new CeilingFan("Living Room");  
        GarageDoor garageDoor = new GarageDoor("Main house");  
        Stereo stereo = new Stereo("Living Room");  
  
        remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);  
        remoteControl.setCommand(1, kitchenLight::on, kitchenLight::off);  
        remoteControl.setCommand(2, ceilingFan::high, ceilingFan::off);  
  
        Command stereoOnWithCD = () -> {  
            stereo.on(); stereo.setCD(); stereo.setVolume(11);  
        };  
        remoteControl.setCommand(3, stereoOnWithCD, stereo::off);  
        remoteControl.setCommand(4, garageDoor::up, garageDoor::down);  
  
        System.out.println(remoteControl);  
  
        remoteControl.onButtonWasPushed(0);  
        remoteControl.offButtonWasPushed(0);  
        remoteControl.onButtonWasPushed(1);  
        remoteControl.offButtonWasPushed(1);  
        remoteControl.onButtonWasPushed(2);  
        remoteControl.offButtonWasPushed(2);  
        remoteControl.onButtonWasPushed(3);  
        remoteControl.offButtonWasPushed(3);  
    }  
}
```

We've removed all the code to create concrete Command objects (and we deleted all those classes too). Now our code's a lot more concise (and we've gone from 22 classes to 9).

We're using method references everywhere we have simple one-method commands, and a full lambda expression for where we need to do more than one method call.

(You can think of a method reference as a compact lambda expression. They're really the same thing; a method reference is just shorthand for a lambda expression that calls just one method.)

And don't forget, we need to modify the RemoteControl constructor to remove the code to construct NoCommand objects, and replace those with lambda expressions too:

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        for (int i = 0; i < 7; i++) {
            onCommands[i] = () -> { };
            offCommands[i] = () -> { };
        }
    }
    // rest of the code here
}
```

We've removed the code to create a NoCommand object.

Instead of a NoCommand object, we use a lambda expression that does nothing! (Just like the execute() method of the NoCommand object did nothing.)



Check out the results of all those lambda expression commands...

```
File Edit Window Help CommandsGetThingsDone

% java RemoteLoader
----- Remote Control -----
[slot 0] RemoteLoader$$Lambda$1/168423058
[slot 1] RemoteLoader$$Lambda$3/258952499
[slot 2] RemoteLoader$$Lambda$5/1325547227
[slot 3] RemoteLoader$$Lambda$9/1706377736
[slot 4] RemoteControl$$Lambda$1/713338599
[slot 5] RemoteControl$$Lambda$1/713338599
[slot 6] RemoteControl$$Lambda$1/713338599
                ↑
                On slots
                ↘
                Off slots
Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off
%
```

Now when we display the remote control, we see these weird names instead of the Command class names. Not a particularly useful display.

Once again, our Commands in action. Only this time, our commands are defined with lambda expressions instead of Command objects.

there are no
Dumb Questions

Q: Can a lambda expression have parameters or return a value? Or does it always have to be a void, no-argument method?

A: Yes, a lambda expression can have parameters and return a value (take a look back at Chapter 2 to see how we used a one-argument lambda expression in place of an ActionListener object in the Swing observer example). But the rules are the same: the signature of the lambda expression must match the signature of the one method in the type of the object you're using the lambda expression to stand in for. To learn more about how to write lambda expressions with parameters and return values (and how to deal with the types), check out the Java docs.

Q: You keep saying that a lambda expression must match a method in an interface with one, and only one, method. So if an interface has two methods, we can't use a lambda expression?

A: That's right. An interface, like our original Command interface (or ActionListener as another example), that has just one method is known as a *functional interface*. Lambda expressions are designed specifically to replace the methods in these functional interfaces, partly as a way to reduce the code that is required when you have a lot of these small classes with functional interfaces. If your interface has two methods, it's not a functional interface and you won't be able to replace it with a lambda expression. Think about it: a lambda expression is really a replacement for a method, not an entire object. You can't replace two methods with one lambda expression.

Q: Does that mean we can't use lambda expressions for our Remote Control implementation with undo? There, our Command interface has two methods: execute() and undo().

A: That's right. You could probably find a way to use lambdas with undo (by making two different types of commands), but in the end your code would probably be more complex than if you'd just used Command objects like we did when we implemented the RemoteControl with undo earlier in the chapter.

Lambda expressions are meant to be used with functional interfaces (one method only), to simplify your code. If you find yourself trying to work around this to support a case like Command with undo, then using lambda expressions probably isn't the right solution.

Q: Why do the names of on and off slots look so weird when we display the RemoteControl?

A: If you take another look at how we implemented the `toString()` method of RemoteControl, you'll see we're using `getClass()` to get the class of the Command object, and then `getName()` to get the name of the class, and printing that to the console as a string. This was a convenient way to get a name for each slot, but kind of a cheat.

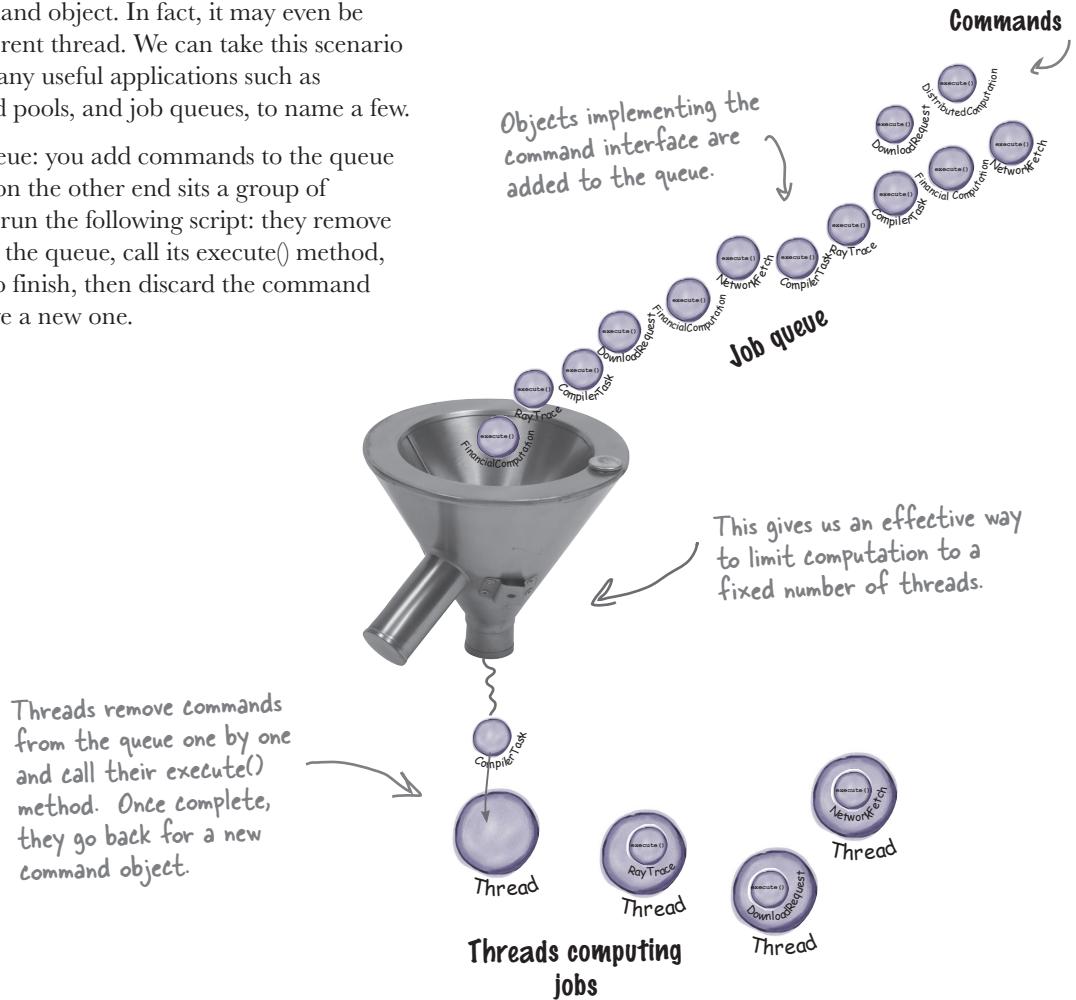
As you can see from the output, lambda expressions don't have nice class names. That's because their names are assigned internally by the Java runtime and Java has no idea what these lambda expressions mean; to Java, they're just function objects that happen to match a method in an interface.

To fix the RemoteControl display, we'd have to modify the `setCommand()` code in RemoteControl, perhaps to allow a name parameter for each slot, and modify the `toString()` method to use this name. Then in RemoteLoader, we'd pass a nice, human-readable name into `setCommand()` along with the commands. This would probably mirror real life more closely (if you're programming your own remote, you'll likely want to set your own custom names).

More uses of the Command Pattern: queuing requests

Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object. Now, the computation itself may be invoked long after some client application creates the command object. In fact, it may even be invoked by a different thread. We can take this scenario and apply it to many useful applications such as schedulers, thread pools, and job queues, to name a few.

Imagine a job queue: you add commands to the queue on one end, and on the other end sits a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.



Note that the job queue classes are totally decoupled from the objects that are doing the computation. One minute a thread may be computing a financial computation, and the next it may be retrieving something from the network. The job queue objects don't care; they just retrieve commands and call execute(). Likewise, as long as you put objects into the queue that implement the Command Pattern, your execute() method will be invoked when a thread is available.



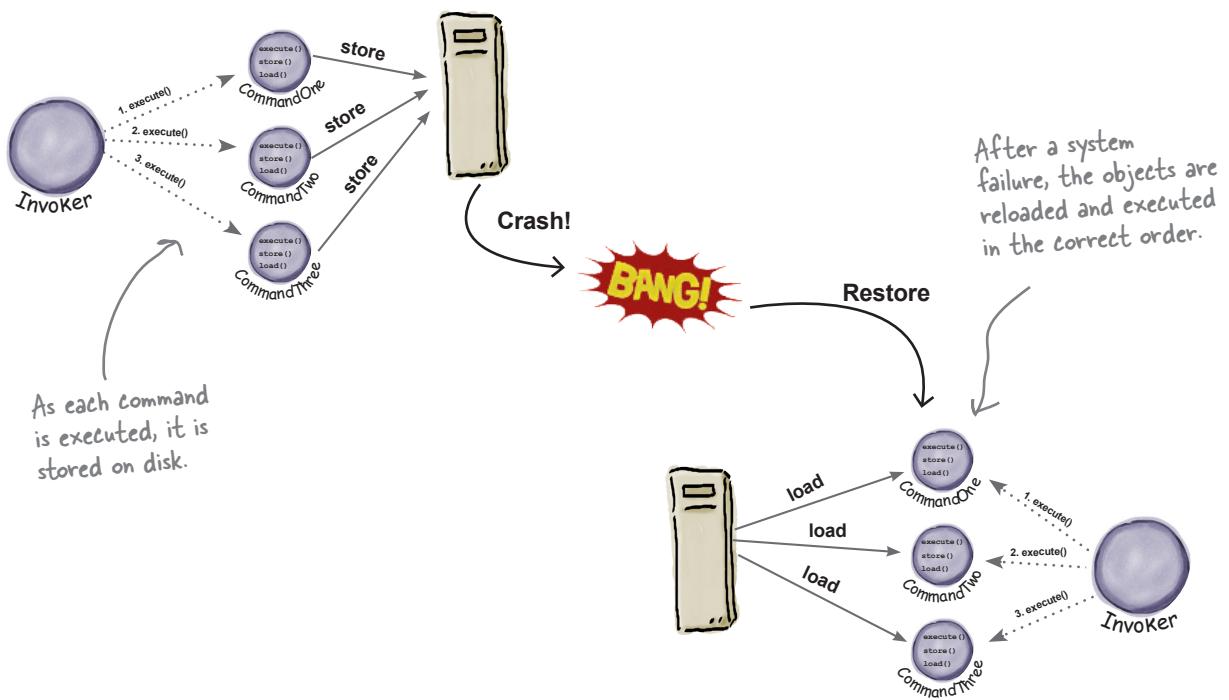
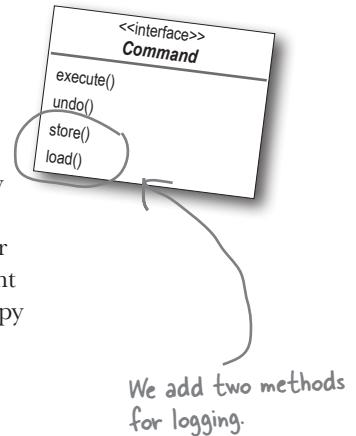
How might a web server make use of such a queue? What other applications can you think of?

More uses of the Command Pattern: logging requests

The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions. The Command Pattern can support these semantics with the addition of two methods: `store()` and `load()`. In Java we could use object serialization to implement these methods, but the normal caveats for using serialization for persistence apply.

How does this work? As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their `execute()` methods in batch and in order.

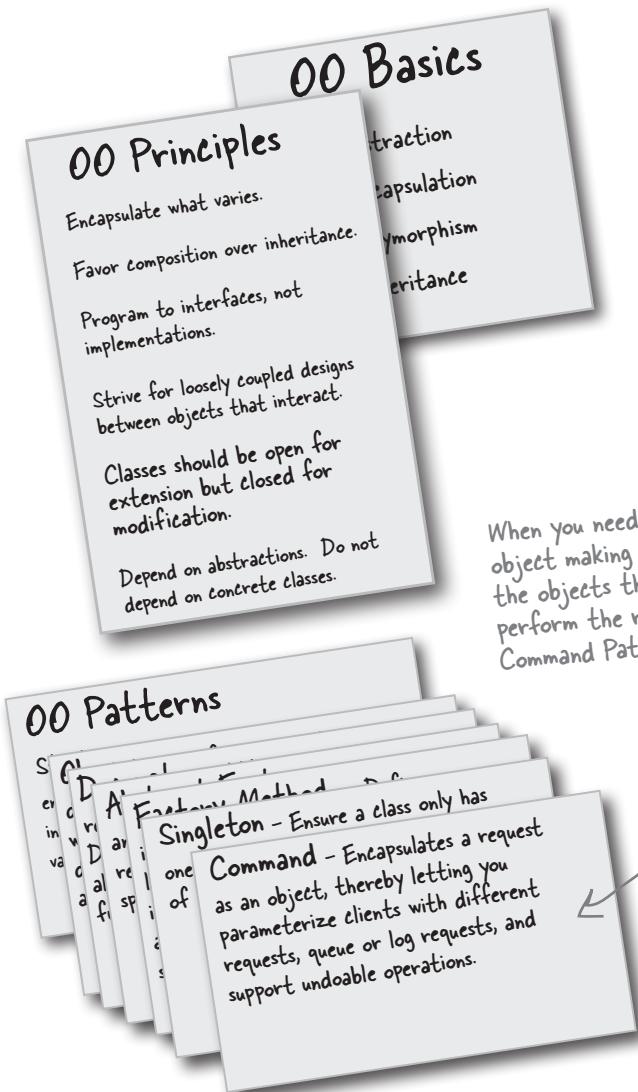
Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last check point, and if there is a system failure, apply those operations to our checkpoint. Take, for example, a spreadsheet application: we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs. In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.





Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a pattern that allows us to encapsulate methods into Command objects: store them, pass them around, and invoke them when you need them.



BULLET POINTS

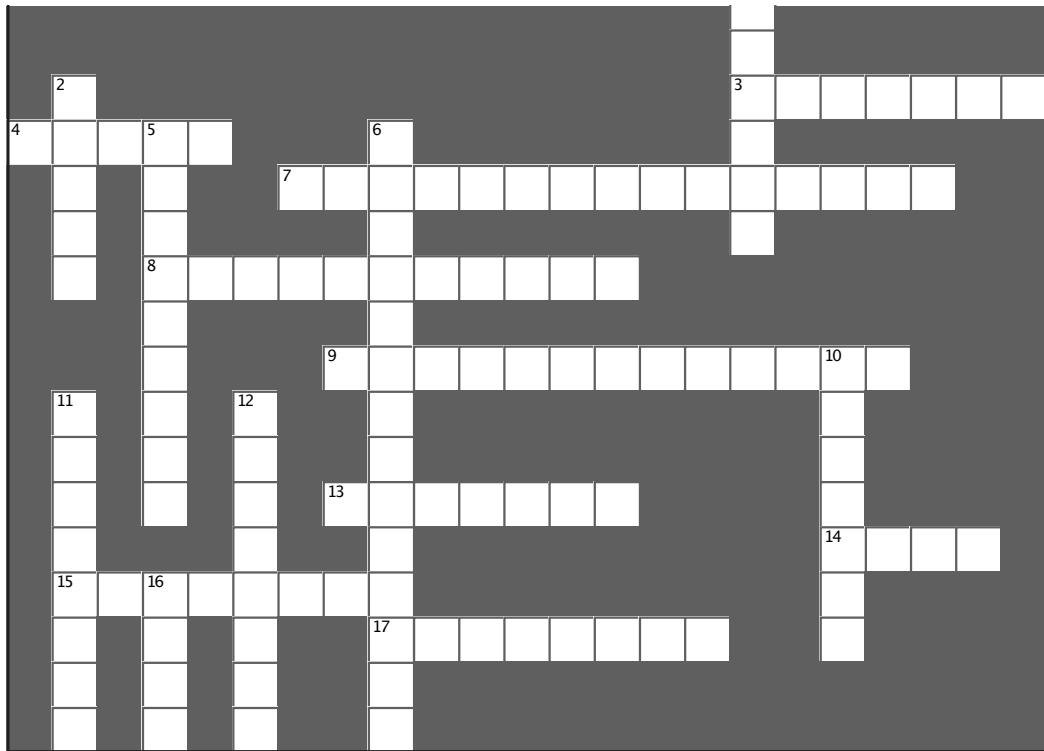
- The Command Pattern decouples an object making a request from the one that knows how to perform it.
- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.
- Macro Commands are a simple extension of Command that allow multiple commands to be invoked. Likewise, Macro Commands can easily support undo().
- In practice, it is not uncommon for "smart" Command objects to implement the request themselves rather than delegating to a receiver.
- Commands may also be used to implement logging and transactional systems.



Design Patterns Crossword

Time to take a breather and let it all sink in.

It's another crossword; all of the solution words are from this chapter.



ACROSS

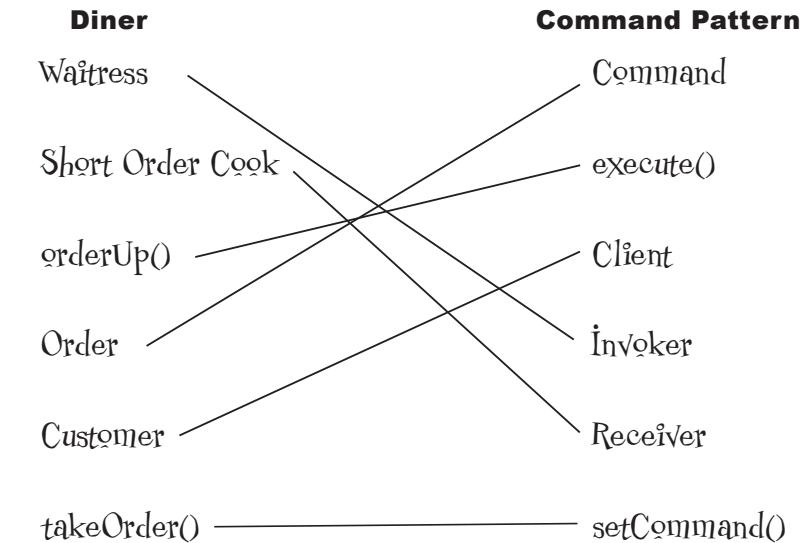
3. The Waitress was one.
4. A command _____ a set of actions and a receiver.
7. Dr. Seuss diner food.
8. Our favorite city.
9. Act as the receivers in the remote control.
13. Object that knows the actions and the receiver.
14. Another thing Command can do.
15. Object that knows how to get things done.
17. A command encapsulates this.

DOWN

1. Role of customer in the Command Pattern.
2. Our first command object controlled this.
5. Invoker and receiver are _____.
6. Company that got us word-of-mouth business.
10. All commands provide this.
11. The Cook and this person were definitely decoupled.
12. Carries out a request.
16. Waitress didn't do this.

WHO DOES WHAT? SOLUTION

Match the diner objects and methods with the corresponding names from the Command Pattern



Sharpen your pencil Solution

Here's the code for the GarageDoorOpenCommand class.

```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;

    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }
    public void execute() {
        garageDoor.up();
    }
}
```

Here's the output:

```
File Edit Window Help GreenEggs&Ham
%java RemoteControlTest
Light is on
Garage Door is Open
%
```



Exercise Solution

Here is the undo() method for the MacroCommand.

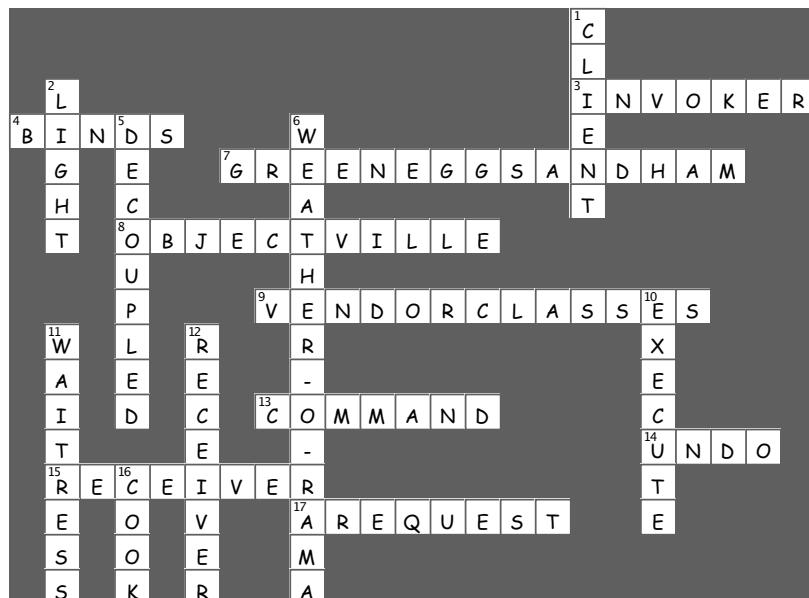
```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
    public void undo() {
        for (int i = commands.length - 1; i >= 0; i--) {
            commands[i].undo();
        }
    }
}
```



Sharpen your pencil Solution

Here is the code to create commands for the off button.

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```



7 the Adapter and Facade Patterns

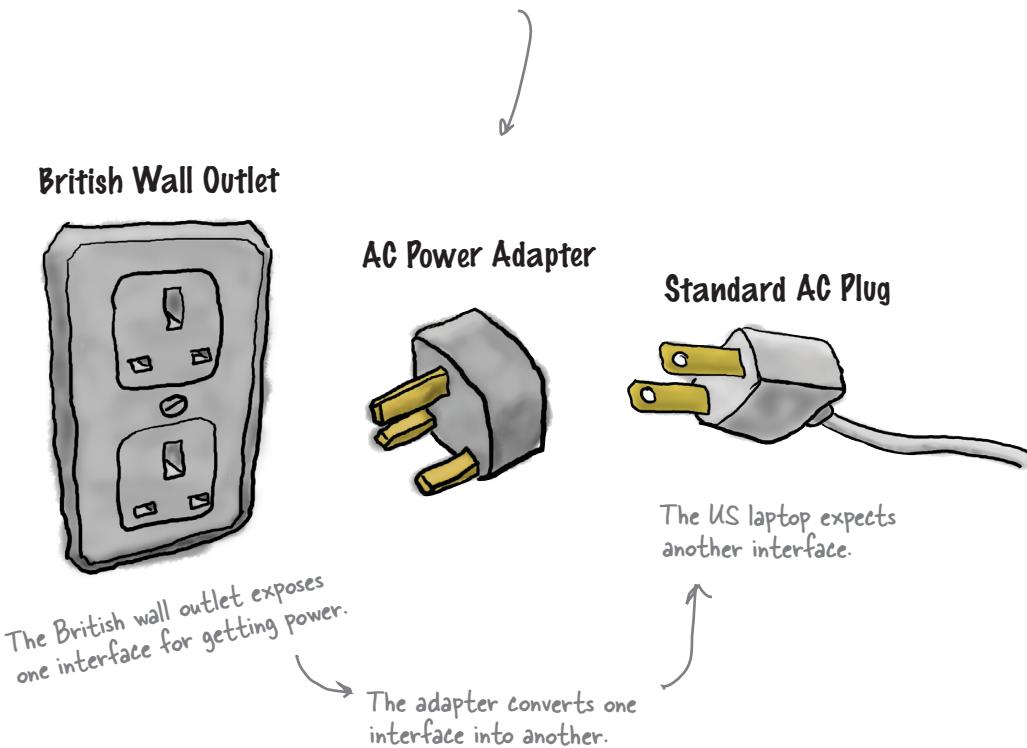
Being Adaptive



In this chapter we're going to attempt such impossible feats as putting a square peg in a round hole. Sound impossible? Not when we have Design Patterns. Remember the Decorator Pattern? We **wrapped objects** to give them new responsibilities. Now we're going to wrap some objects with a different purpose: to make their interfaces look like something they're not. Why would we do that? So we can adapt a design expecting one interface to a class that implements a different interface. That's not all; while we're at it, we're going to look at another pattern that wraps objects to simplify their interface.

Adapters all around us

You'll have no trouble understanding what an OO adapter is because the real world is full of them. How's this for an example: Have you ever needed to use a US-made laptop in Great Britain? Then you've probably needed an AC power adapter...



You know what the adapter does: it sits in between the plug of your laptop and the British AC outlet; its job is to adapt the British outlet so that you can plug your laptop into it and receive power. Or look at it this way: the adapter changes the interface of the outlet into one that your laptop expects.

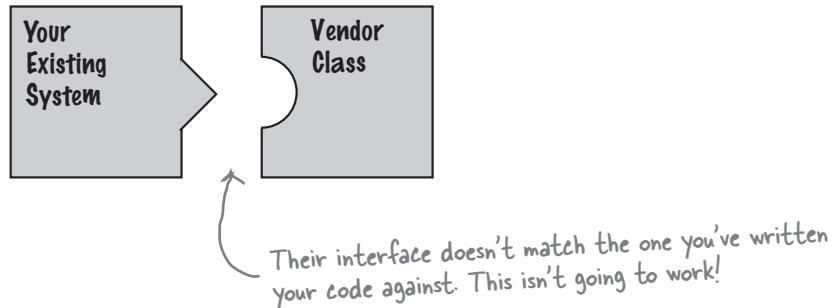
Some AC adapters are simple—they only change the shape of the outlet so that it matches your plug, and they pass the AC current straight through—but other adapters are more complex internally and may need to step the power up or down to match your devices' needs.

Okay, that's the real world; what about object-oriented adapters? Well, our 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.

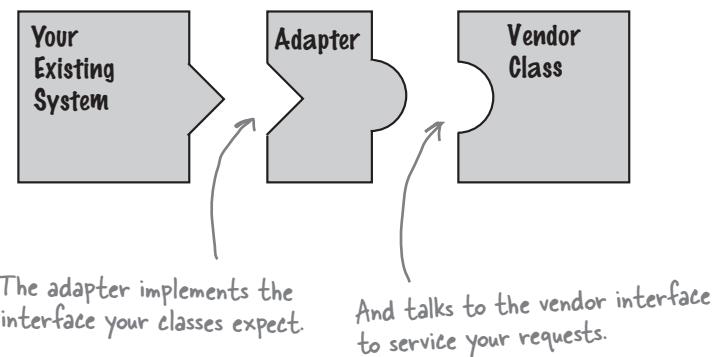
How many other real-world adapters can you think of?

Object-oriented adapters

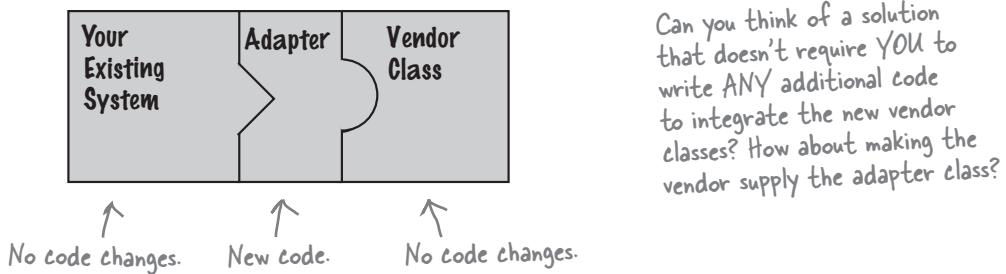
Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



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

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test drive the adapter

Now we just need some code to test drive our adapter:

```

public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();
        WildTurkey turkey = new WildTurkey();
        TurkeyAdapter turkeyAdapter = new TurkeyAdapter(turkey);
        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();
        System.out.println("\nThe Duck says...");
        testDuck(duck);
        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}

```

Let's create a Duck...
...and a Turkey.
And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run

```

File Edit Window Help Don'tForgetToDuck
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance

```

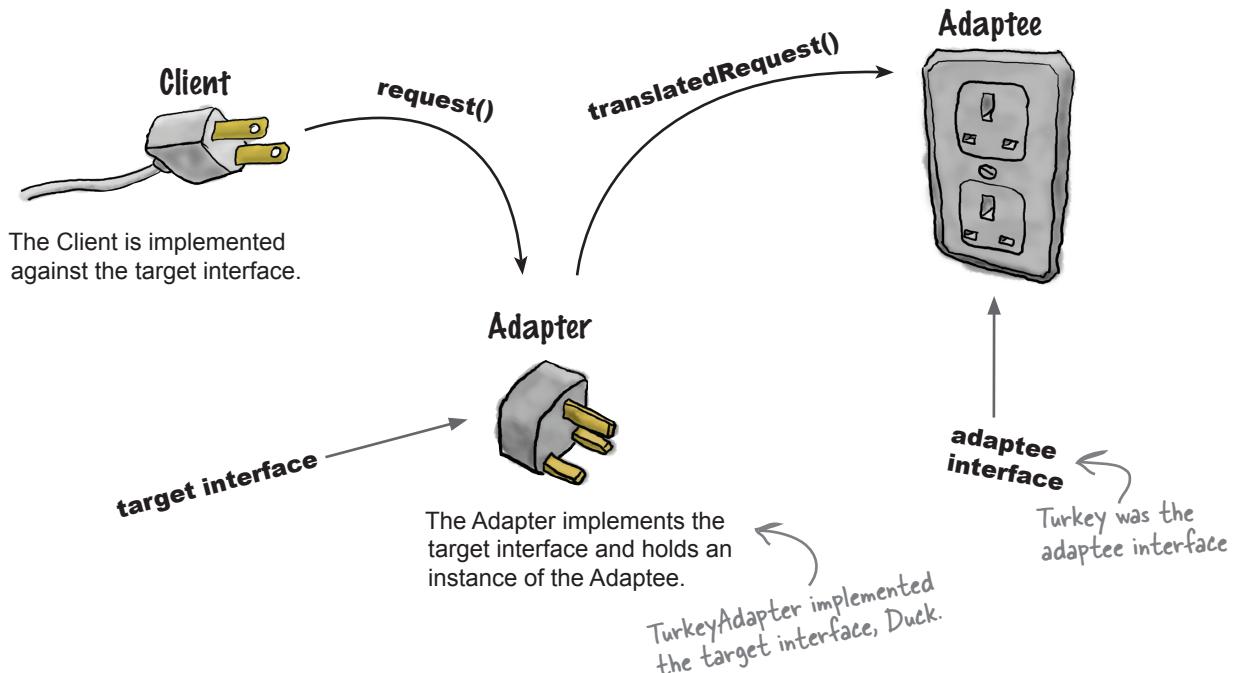
The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained

Now that we have an idea of what an Adapter is, let's step back and look at all the pieces again.



Here's how the Client uses the Adapter

- ➊ The client makes a request to the adapter by calling a method on it using the target interface.
- ➋ The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
- ➌ 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, and 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.

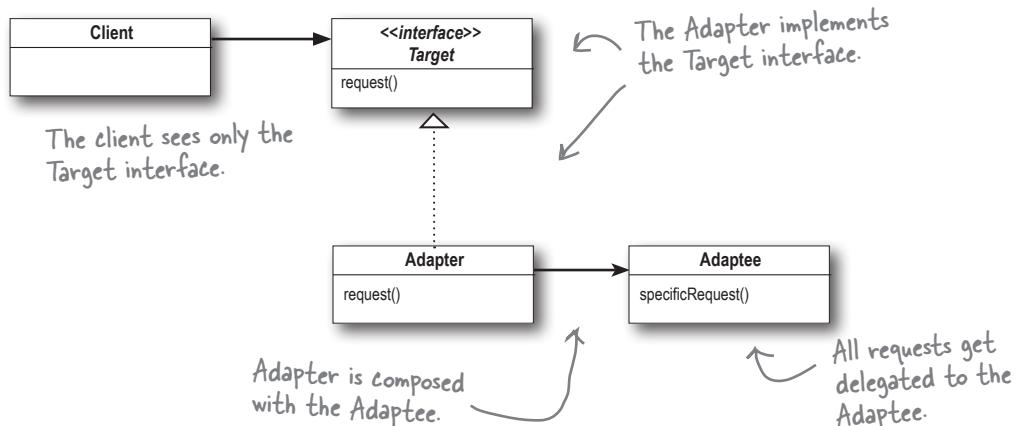
Adapter Pattern defined

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:



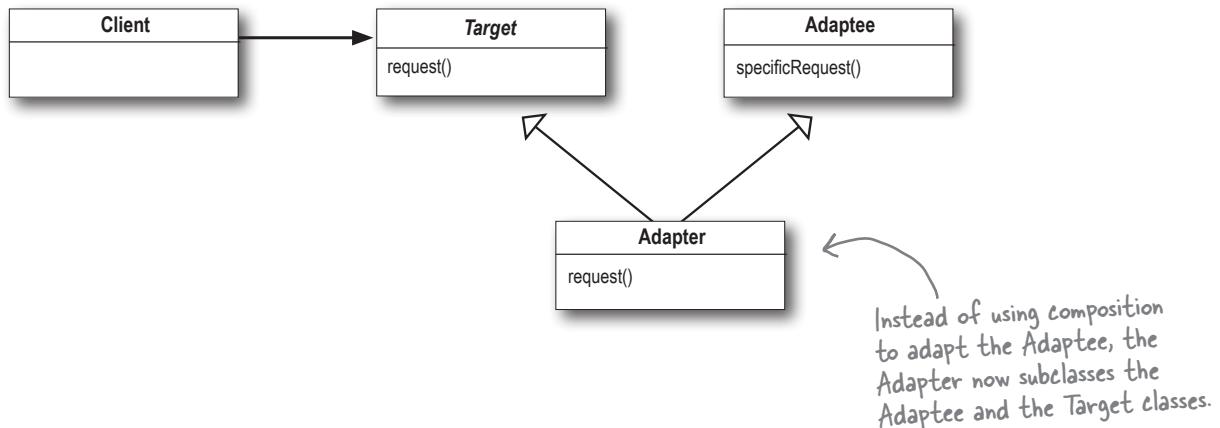
The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

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.



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.



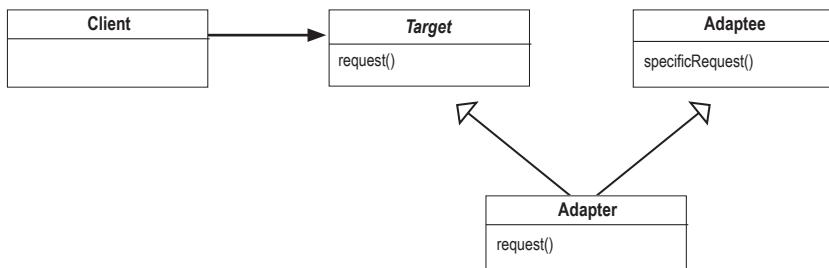
Object adapters and class adapters use two different means of adapting the adaptee (composition versus inheritance). How do these implementation differences affect the flexibility of the adapter?



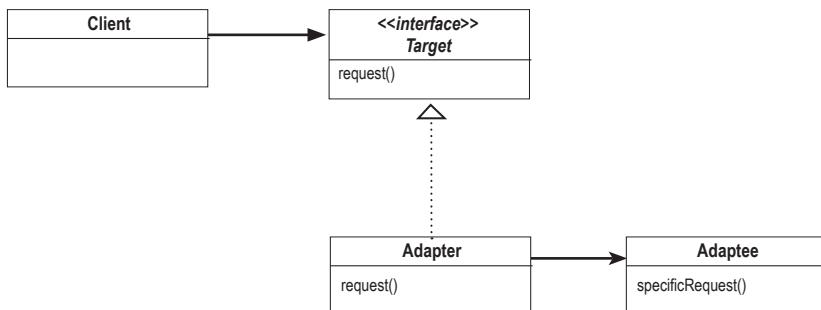
Duck Magnets

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.

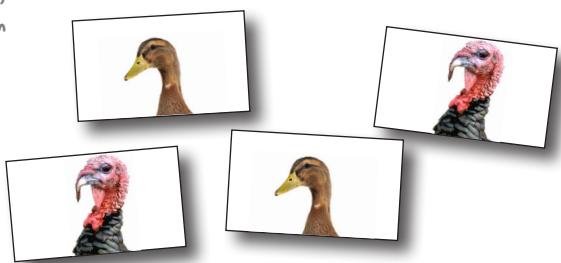
Class Adapter



Object Adapter



Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.



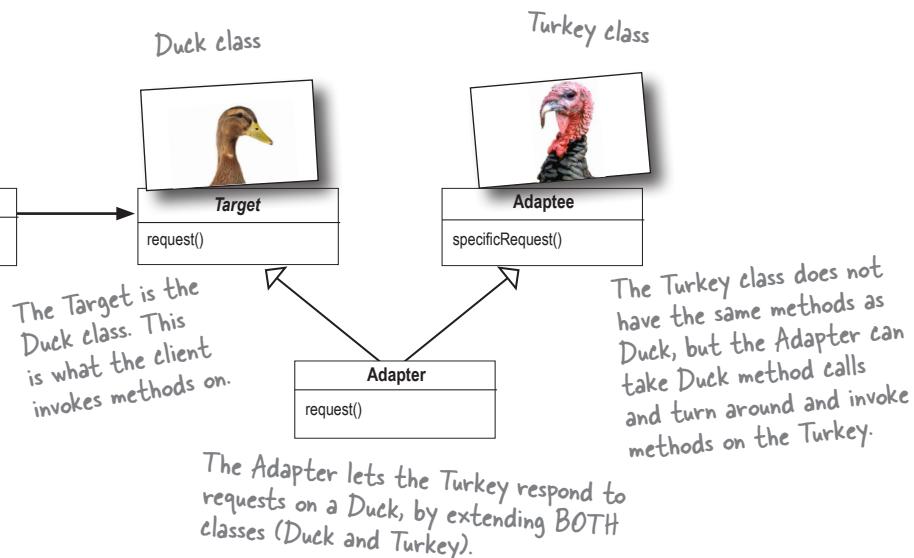


Duck Magnets Answer

Note: the class adapter uses multiple inheritance, so you can't do it in Java...

Class Adapter

Client thinks he's talking to a Duck.



Object Adapter

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.



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, but 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?

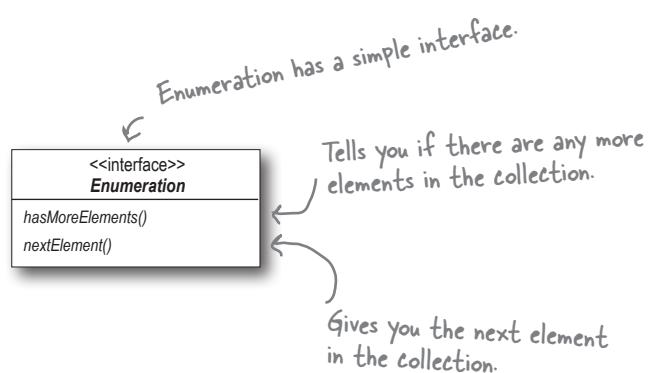
Sounds messy...

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)...

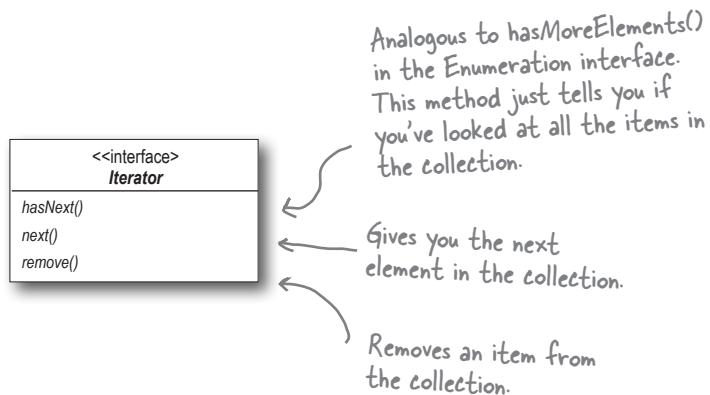
Old-world Enumerators

If you've been around Java for a while you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, `elements()`, which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



New-world Iterators

The newer Collection classes use an Iterator interface that, like Enumeration, allows you to iterate through a set of items in a collection, but also adds the ability to remove items.

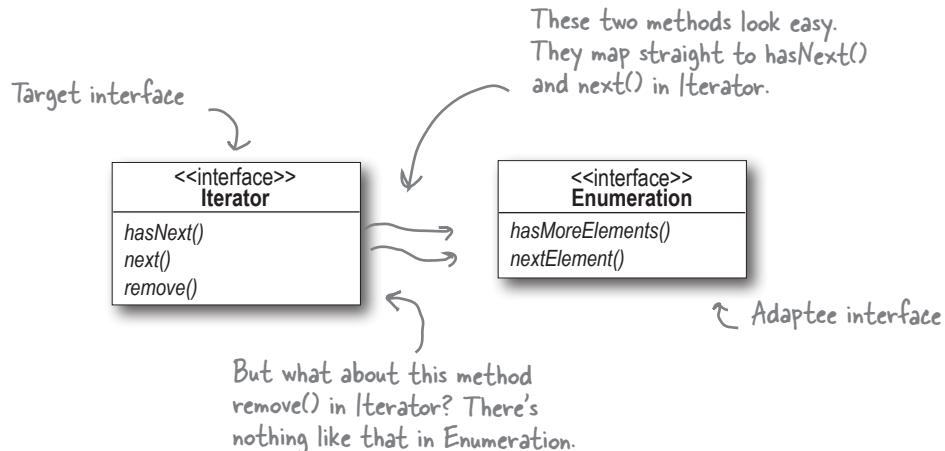


And today...

We are often faced with legacy code that exposes the Enumeration interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

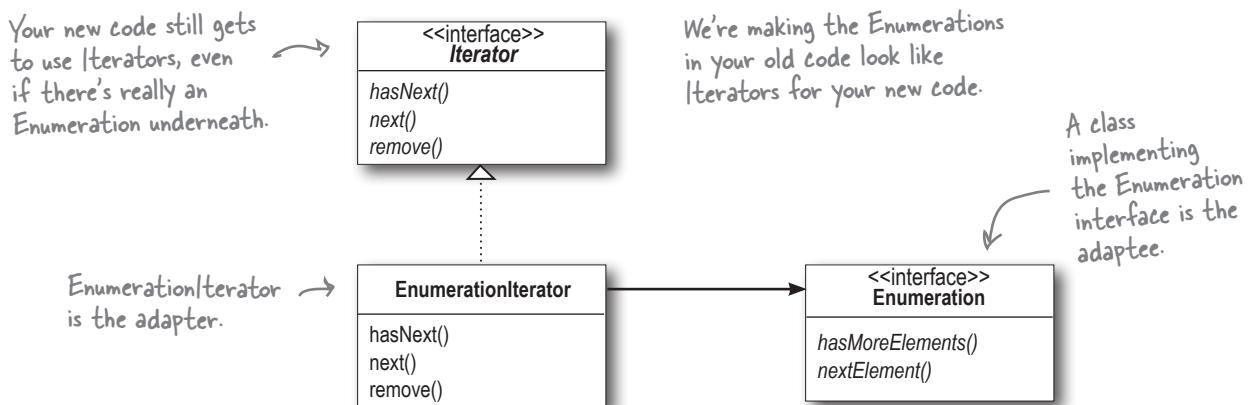
Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and 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:



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<Object> {
    Enumeration<?> enumeration;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface... it has to look like an Iterator.

The Enumeration we're adapting. We're using composition so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

... and the Iterator's next() method is delegated to the Enumeration's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.



While Java has gone in the direction of the Iterator, there is nevertheless a lot of legacy client code that depends on the Enumeration interface, so an Adapter that converts an Iterator to an Enumeration is also quite useful.

Write an Adapter that adapts an Iterator to an Enumeration. You can test your code by adapting an ArrayList. The ArrayList class supports the Iterator interface but doesn't support Enumerations (well, not yet anyway).



Some AC adapters do more than just change the interface—they add other features like surge protection, indicator lights, and other bells and whistles.

If you were going to implement these kinds of features, what pattern would you use?

Fireside Chats



Tonight's talk: **The Decorator Pattern and the Adapter Pattern discuss their differences.**

Decorator:

I'm important. My job is all about *responsibility*—you know that when a Decorator is involved there's going to be some new responsibilities or behaviors added to your design.

That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.

Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.

Adapter:

You 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.

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."

Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job.

Decorator:

Well, us decorators do that as well, only we allow *new behavior* to be added to classes without altering existing code. I still say that adapters are just fancy decorators—I mean, just like us, you wrap an object.

Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap; we aren't a *simple pass through*.

Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *miles apart* in our *intent*.

Adapter:

But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *any* code; they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.

No, no, no, not at all. We *always* convert the interface of what we wrap; you *never* do. I'd say a decorator is like an adapter; it is just that you don't change the interface!

Hey, who are you calling a simple pass through? Come on down and we'll see how long *you* last converting a few interfaces!

Oh yeah, I'm with you there.

And now for something different...

There's another pattern in this chapter.

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.

We're going to look at a pattern now that alters an interface, but for a different reason: to simplify the interface. It's aptly named the Facade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well-lit facade.



Match each pattern with its intent:

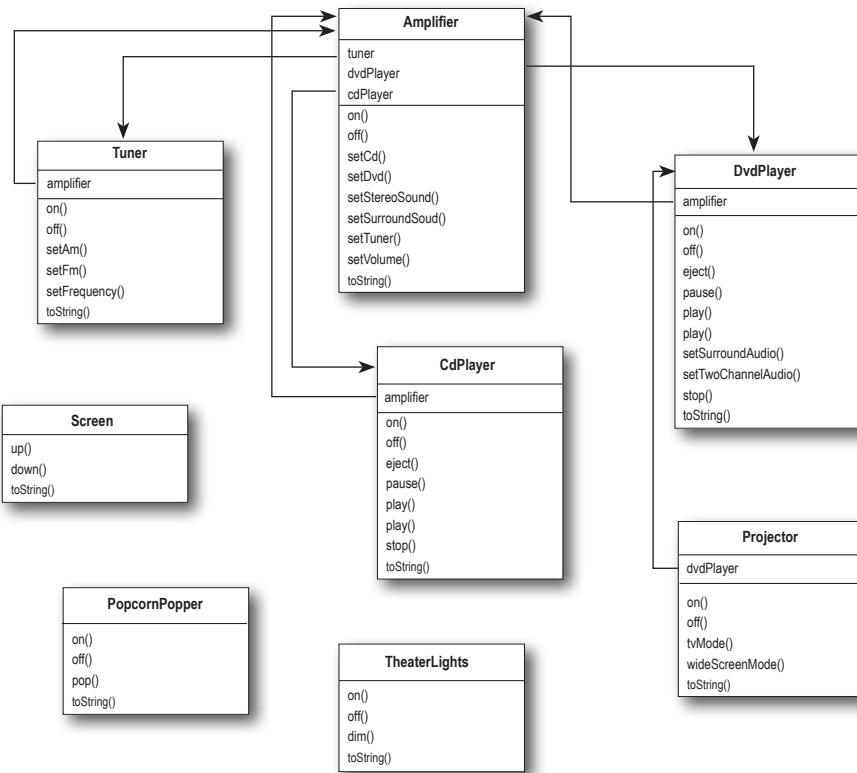
Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

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:



That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.

You've spent weeks running wire, mounting the projector, making all the connections, and fine tuning. Now it's time to put it all in motion and enjoy a movie...

Watching a movie (the hard way)

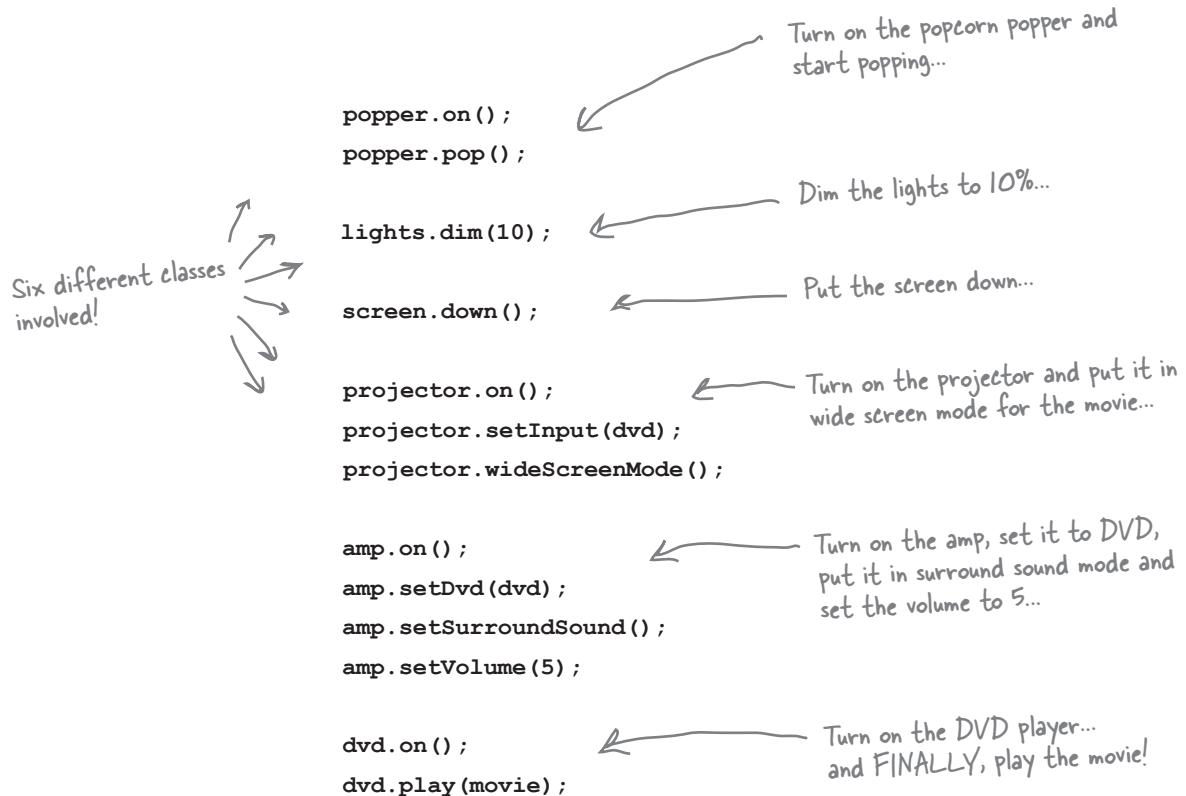
Pick out a DVD, relax, and get ready for movie magic. Oh, there's just one thing—to watch the movie, you need to perform a few tasks:

- 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:



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.

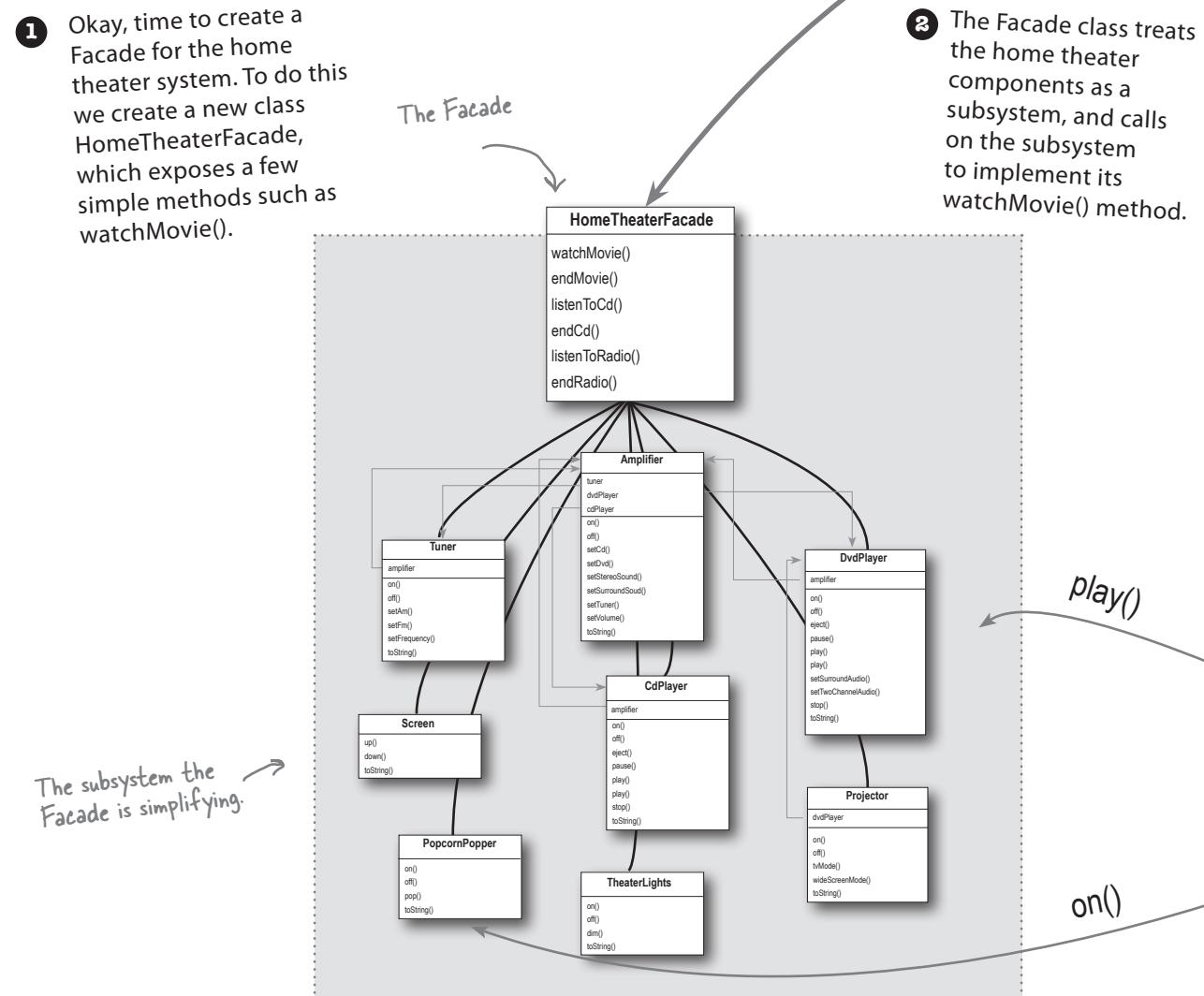
So what to do? The complexity of using your home theater is becoming apparent!

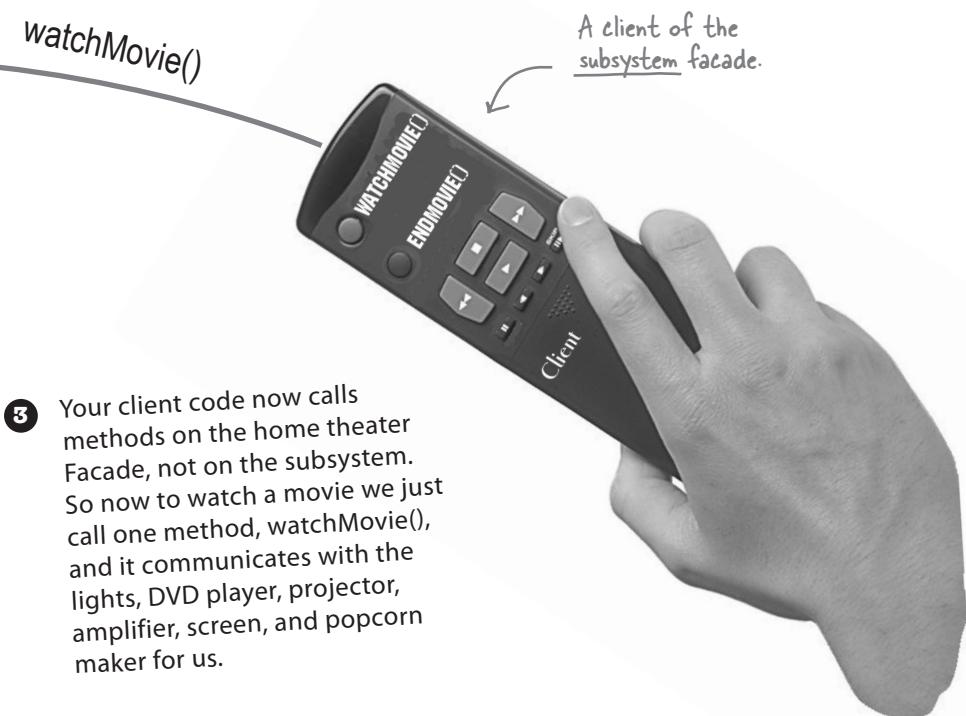
Let's see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

Lights, Camera, Facade!

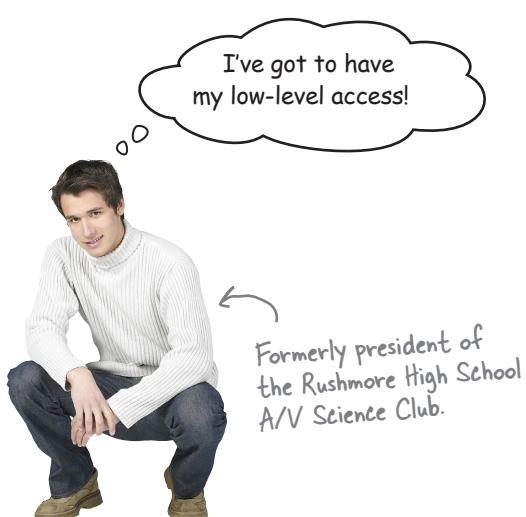
A Facade is just what you need: with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.

Let's take a look at how the Facade operates:





- ➊ Your client code now calls methods on the home theater facade, not on the subsystem. So now to watch a movie we just call one method, `watchMovie()`, and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.



- ➋ The Facade still leaves the subsystem accessible to be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

there are no Dumb Questions

Q: If the facade encapsulates the subsystem classes, how does a client that needs lower-level functionality gain access to them?

A: Facades don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces. This is a nice property of the Facade Pattern: it provides a simplified interface while still exposing the full functionality of the system to those who may need it.

Q: Does the facade add any functionality or does it just pass through each request to the subsystem?

A: A facade is free to add its own "smarts" in addition to making use of the subsystem. For instance, while our home theater facade doesn't implement any new behavior, it is smart enough to know that the popcorn popper has to be turned on before it can pop (as well as the details of how to turn on and stage a movie showing).

Q: Does each subsystem have only one facade?

A: Not necessarily. The pattern certainly allows for any number of facades to be created for a given subsystem.

Q: What is the benefit of the facade other than the fact that I now have a simpler interface?

A: The Facade Pattern also allows you to decouple your client implementation from any one subsystem. Let's say that you get a big raise and decide to upgrade your home theater to all new components that have different interfaces. Well, if you coded your client to the facade rather than the subsystem, your client code doesn't need to change, just the facade (and hopefully the manufacturer is supplying that!).

Q: So the way to tell the difference between the Adapter Pattern and the Facade Pattern is that the adapter wraps one class and the facade may represent many classes?

A: No! Remember, the Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface.

The difference between the two is not in terms of how many classes they "wrap," it is in their intent. The intent of the Adapter Pattern is to alter an interface so that it matches one a client is expecting. The intent of the Facade Pattern is to provide a simplified interface to a subsystem.

A facade not only simplifies an interface, it decouples a client from a subsystem of components.

Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.

Constructing your home theater facade

Let's step through the construction of the HomeTheaterFacade. The first step is to use composition so that the facade has access to all the components of the subsystem:

```

public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
        Tuner tuner,
        DvdPlayer dvd,
        CdPlayer cd,
        Projector projector,
        Screen screen,
        TheaterLights lights,
        PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // other methods here
}

```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementing the simplified interface

Now it's time to bring the components of the subsystem together into a unified interface.
Let's implement the `watchMovie()` and `endMovie()` methods:

```
public void watchMovie(String movie) {  
    System.out.println("Get ready to watch a movie...");  
    popper.on();  
    popper.pop();  
    lights.dim(10);  
    screen.down();  
    projector.on();  
    projector.wideScreenMode();  
    amp.on();  
    amp.setDvd(dvd);  
    amp.setSurroundSound();  
    amp.setVolume(5);  
    dvd.on();  
    dvd.play(movie);  
}  
  
public void endMovie() {  
    System.out.println("Shutting movie theater down...");  
    popper.off();  
    lights.on();  
    screen.up();  
    projector.off();  
    amp.off();  
    dvd.stop();  
    dvd.eject();  
    dvd.off();  
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.



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
    }
}
```

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.

```
HomeTheaterFacade homeTheater =
    new HomeTheaterFacade(amp, tuner, dvd, cd,
                         projector, screen, lights, popper);
```

First you instantiate the Facade with all the components in the subsystem.

```
homeTheater.watchMovie("Raiders of the Lost Ark");
homeTheater.endMovie();
}
```

Use the simplified interface to first start the movie up, and then shut it down.

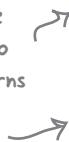
}

Here's the output.

Calling the Facade's `watchMovie()` does all this work for us...



...and here, we're done watching the movie, so calling `endMovie()` turns everything off.



```
File Edit Window Help SnakesWhyDntHaveToBeSnakes?
%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
%
```

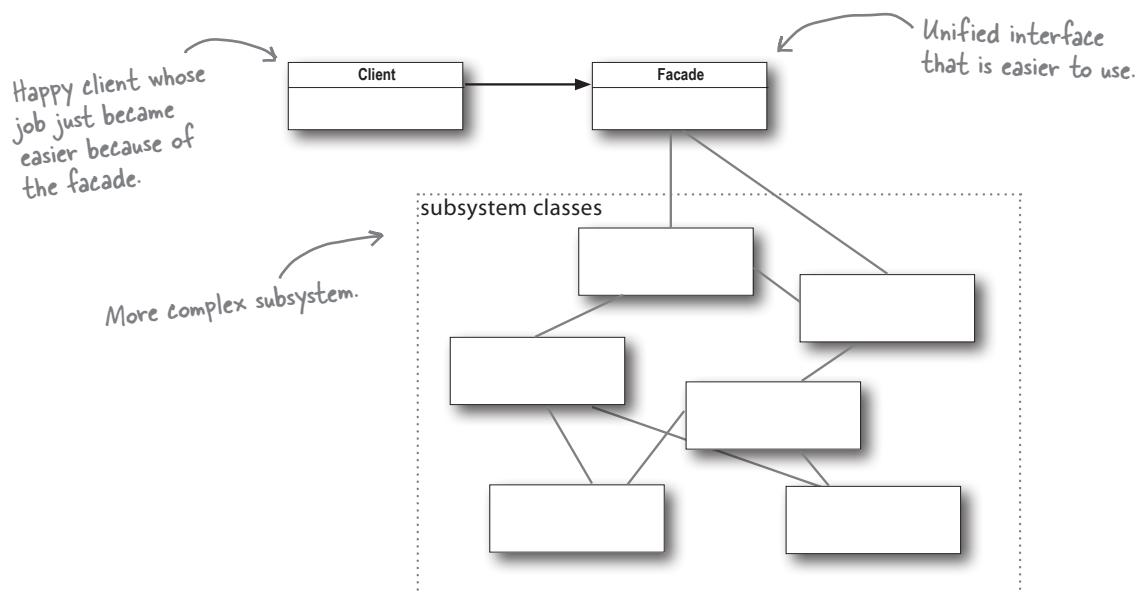
Facade Pattern defined

To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind-bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object-oriented principle.

Before we introduce that new principle, let's take a look at the official definition of the pattern:

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

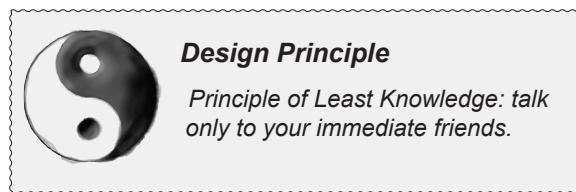
There isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade is to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram:



That's it; you've got another pattern under your belt! Now, it's time for that new OO principle. Watch out, this one can challenge some assumptions!

The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close “friends.” The principle is usually stated as:



But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.



How many classes 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:

```

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
    }
}

```

The code is annotated with several arrows pointing from handwritten text to specific parts of the code:

- An arrow points to the declaration of the `engine` field with the text: "Here's a component of this class. We can call its methods."
- An arrow points to the constructor `public Car()` with the text: "Here we're creating a new object; its methods are legal."
- An arrow points to the parameter `key` in the `start` method with the text: "You can call a method on an object passed as a parameter."
- An arrow points to the line `key.turns()` with the text: "You can call a method on a component of the object."
- An arrow points to the local method `updateDashboardDisplay` with the text: "You can call a local method within the object."
- An arrow points to the line `doors.lock()` with the text: "You can call a method on an object you create or instantiate."

*there are no
Dumb Questions*

Q: There is another principle called the Law of Demeter; how are they related?

A: The two are one and the same and you'll encounter these terms being used interchangeably. We prefer to use the Principle of Least Knowledge for a couple of reasons: (1) the name is more intuitive and (2) the use of the word "Law" implies we always have to apply this principle. In fact, no principle is a law, all principles should

be used when and where they are helpful. All design involves tradeoffs (abstractions versus speed, space versus time, and so on) and while principles provide guidance, 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**

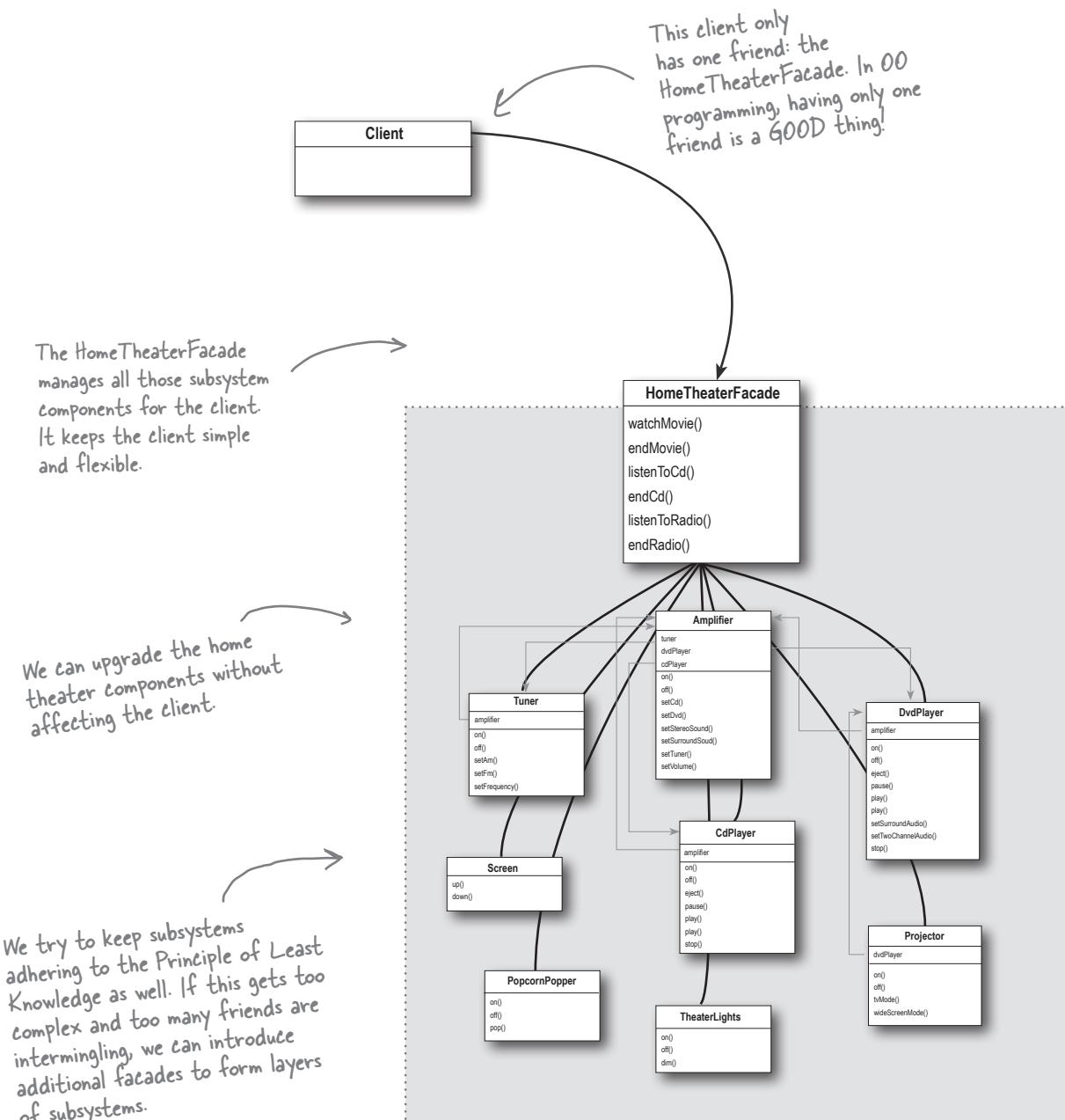


Can you think of a common use of Java that violates the Principle of Least Knowledge?

Should you care?

Answer: How about System.out.println()?

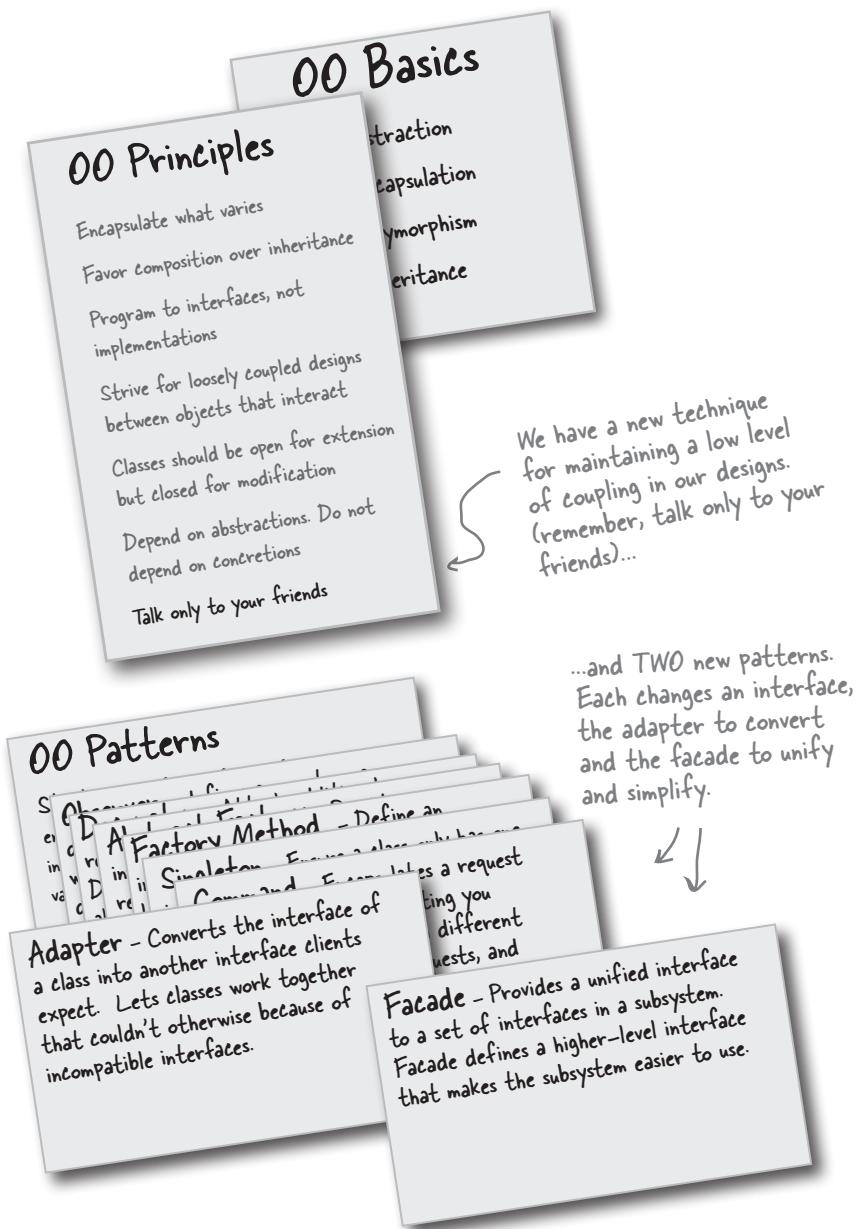
The Facade and the Principle of Least Knowledge





Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.



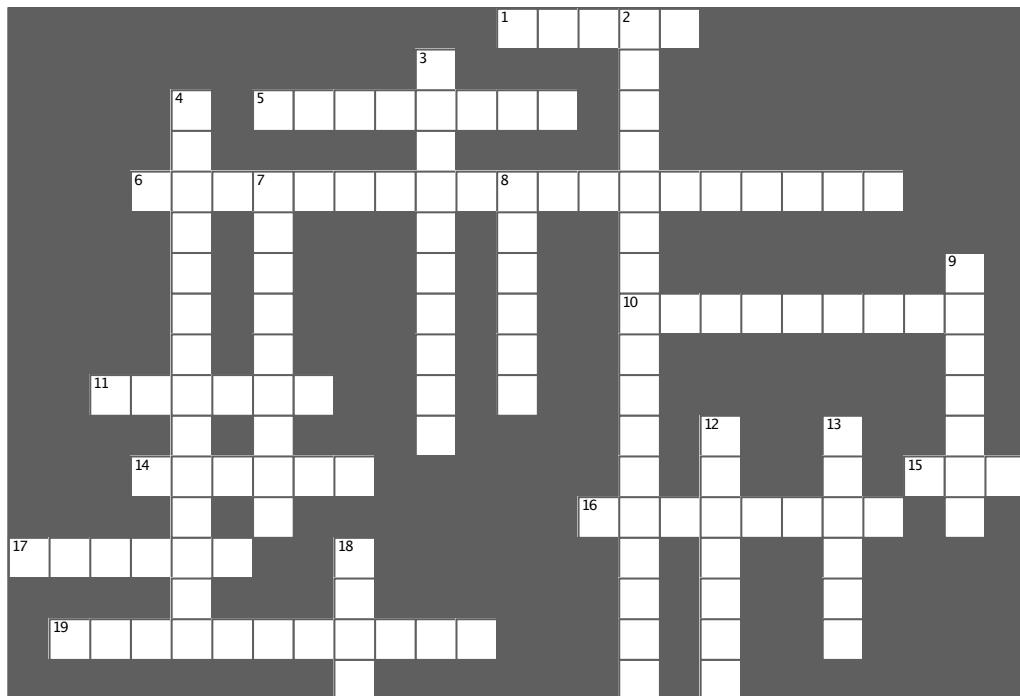
BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.



Design Patterns Crossword

Yes, it's another crossword. All of the solution words are from this chapter.



ACROSS

1. True or false? Adapters can wrap only one object.
5. An Adapter _____ an interface.
6. Movie we watched (five words).
10. If in Britain, you might need one of these (two words).
11. Adapter with two roles (two words).
14. Facade still _____ low-level access.
15. Ducks do it better than Turkeys.
16. Disadvantage of the Principle of Least Knowledge: too many _____.
17. A _____ simplifies an interface.
19. New American dream (two words).

DOWN

2. Decorator called Adapter this (three words).
3. One advantage of Facade.
4. Principle that wasn't as easy as it sounded (two words).
7. A _____ adds new behavior.
8. Masquerading as a Duck.
9. Example that violates the Principle of Least Knowledge: System.out._____.
12. No movie is complete without this.
13. Adapter client uses the _____ interface.
18. An Adapter and a Decorator can be said to _____ an object.



Sharpen your pencil Solution

```
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();
        }
    }
}
```

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Here's our solution:

Now we are adapting Turkeys to Ducks, so we implement the Turkey interface.

We stash a reference to the Duck we are adapting.

We also recreate a random object; take a look at the fly() method to see how it is used.

A gobble just becomes a quack.

Since Ducks fly a lot longer than Turkeys, we decided to only fly the Duck on average one of five times.



Sharpen your pencil Solution

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;
    // other methods and constructor
    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
public House {
    WeatherStation station;
    // other methods and constructor
    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }
    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Violates the Principle of Least Knowledge!
You are calling the method of an object returned from another call.



Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?



Exercise Solution

You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumeration.

Notice we keep the type parameter generic so this will work for any type of object.

```
public class IteratorEnumeration implements Enumeration<Object> {
    Iterator<?> iterator;
    public IteratorEnumeration(Iterator<?> iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

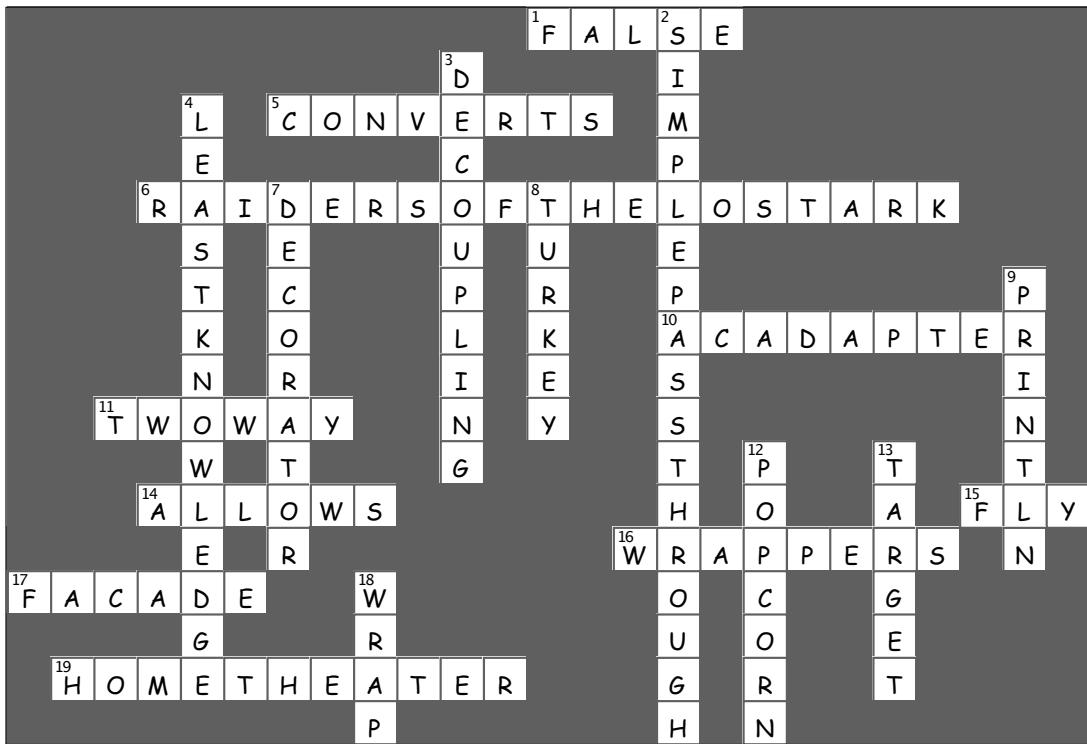
* WHO DOES WHAT? SOLUTION *

Match each pattern with its intent:

Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler



Design Patterns Crossword Solution



8 the Template Method Pattern

Encapsulating Algorithms



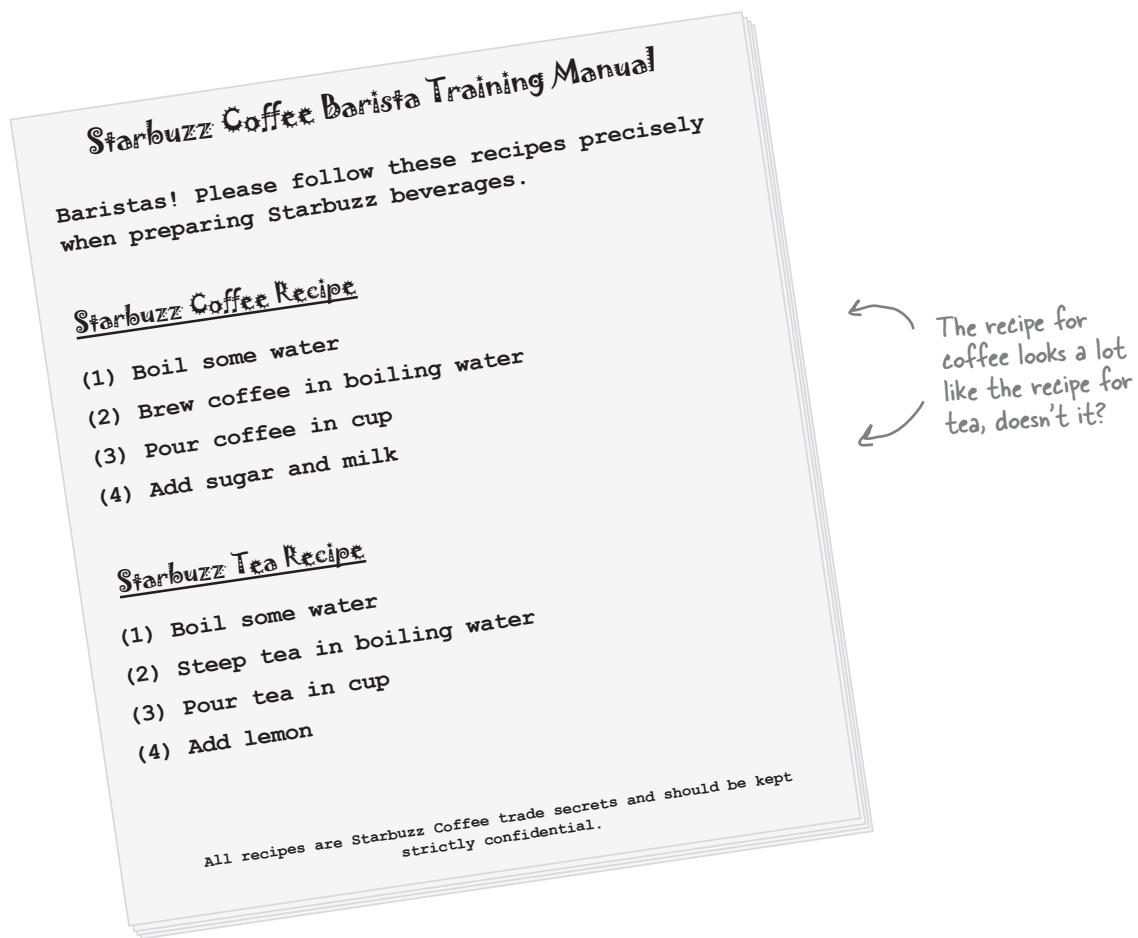
We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas...what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.

coffee and tea recipes are similar

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:



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.
↓
public class Coffee {
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
    public void boilWater() {
        System.out.println("Boiling water");
    }
    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

And now the Tea...

```

public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

```

This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.



When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?



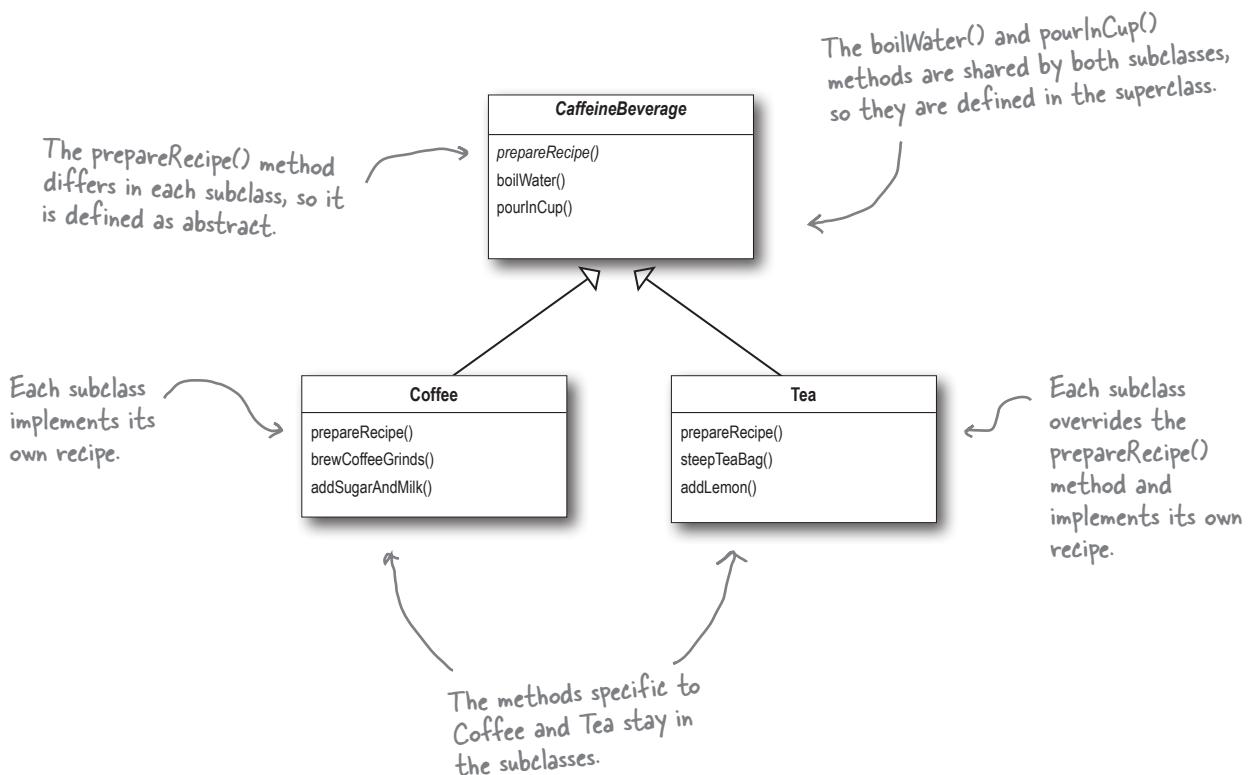


Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:

Sir, may I abstract your Coffee, Tea?

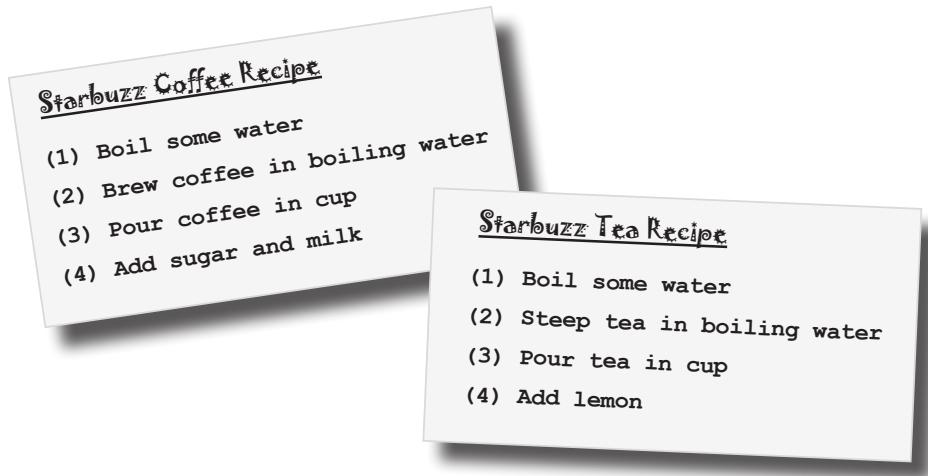
It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:



Did we do a good job on the redesign? Hmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

So what else do Coffee and Tea have in common? Let's start with the recipes.



Notice that both recipes follow the same algorithm:

- 1** Boil some water.
- 2** Use the hot water to extract the coffee or tea.
- 3** Pour the resulting beverage into a cup.
- 4** Add the appropriate condiments to the beverage.

These aren't abstracted but are the same; they just apply to different beverages.

These two are already abstracted into the base class.

So, can we find a way to abstract `prepareRecipe()` too? Yes, let's find out...

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

- 1 The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods, while Tea uses steepTeaBag() and addLemon() methods.



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:

```

public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();
    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

```

CaffeineBeverage is abstract,
just like in the class design.

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:

```

public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }

    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}

```

As in our design, Tea and Coffee
now extend CaffeineBeverage.

Tea needs to define brew() and
addCondiments()—the two abstract
methods from CaffeineBeverage.

Same for Coffee, except Coffee
deals with coffee, and sugar and milk
instead of tea bags and lemon.



Sharpen your pencil

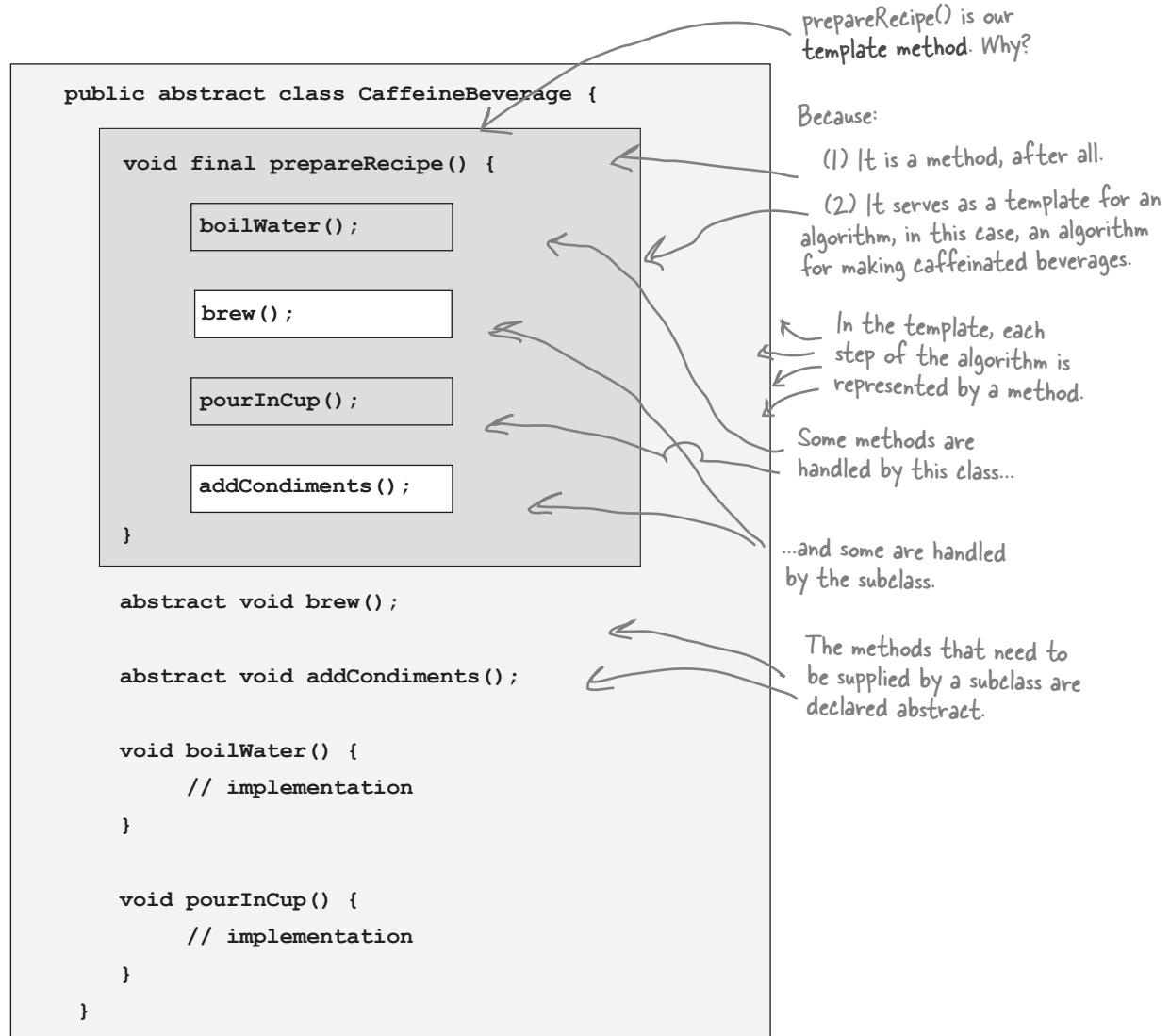
Draw the new class diagram now that we've moved the implementation of `prepareRecipe()` into the `CaffeineBeverage` class:

What have we done?



Meet the Template Method

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method":



The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Let's make some tea...

Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

Behind the Scenes



- 1 Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

```
boilWater();
brew();
pourInCup();
addCondiments();
```

- 2 Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

- 3 First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.

The `prepareRecipe()` method controls the algorithm. No one can change this, and it counts on subclasses to provide some or all of the implementation.

- 4 Next we need to brew the tea, which only the subclass knows how to do:

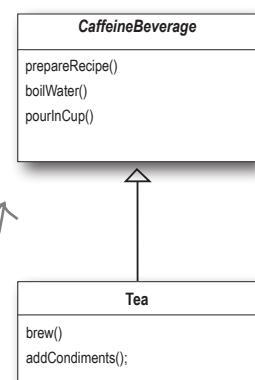
```
brew();
```

Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

```
pourInCup();
```

- 6 Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```



What did the Template Method get us?



Underpowered Tea & Coffee implementation



New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.

The CaffeineBeverage class runs the show; it has the algorithm, and protects it.

Code is duplicated across Coffee and Tea.

The CaffeineBeverage class maximizes reuse among the subclasses.

Code changes to the algorithm require opening the subclasses and making multiple changes.

The algorithm lives in one place and code changes only need to be made there.

Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.

The Template Method 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.

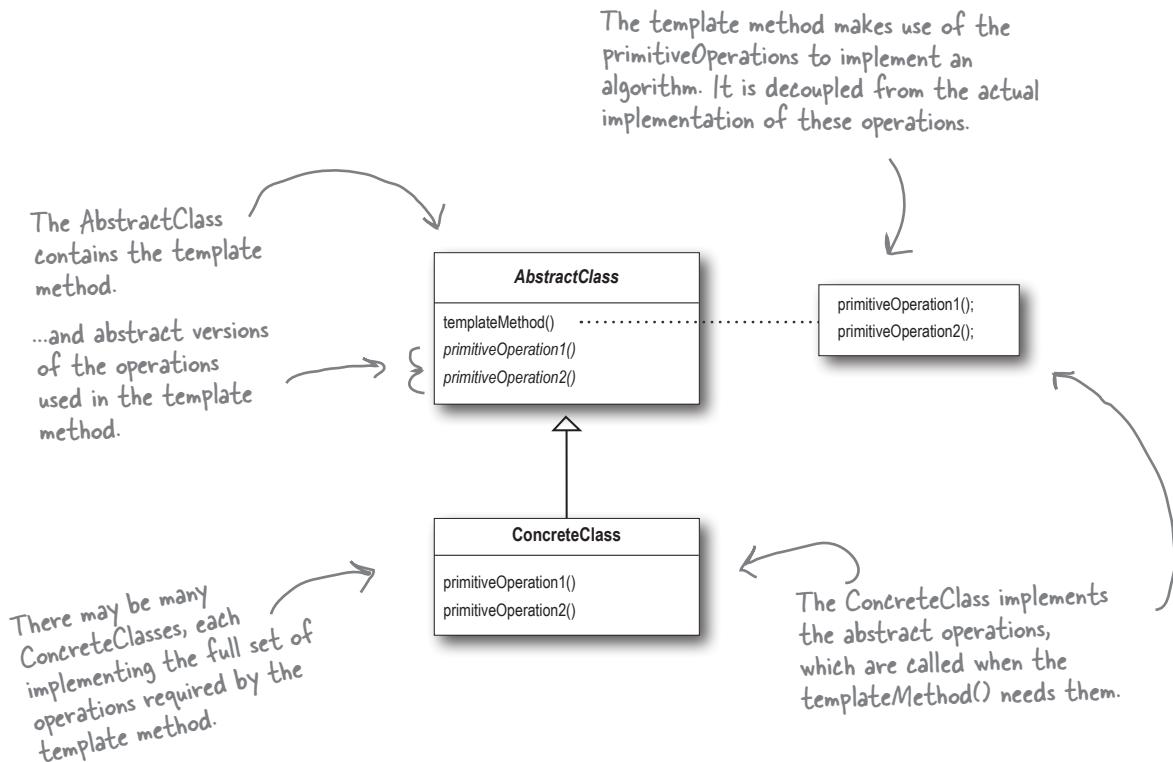
Template Method Pattern defined

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:





Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

```
abstract class AbstractClass {  
  
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
    }  
  
}
```

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

The template method defines the sequence of steps, each represented by a method.

```
abstract void primitiveOperation1();  
  
abstract void primitiveOperation2();  
  
void concreteOperation() {  
    // implementation here  
}  
}
```

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 methods that can go in the abstract class:

We've changed the `templateMethod()` to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

A concrete method, but it does nothing!

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.

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Hooked on Template Method...

A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

There are several uses of hooks; let’s take a look at one now. We’ll talk about a few other uses later:

```
public abstract class CaffeineBeverageWithHook {  
  
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        if (customerWantsCondiments()) {  
            addCondiments();  
        }  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    boolean customerWantsCondiments() {  
        return true;  
    }  
}
```

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.



We've added a little conditional statement that bases its success on a concrete method, `customerWantsCondiments()`. If the customer WANTS condiments, only then do we call `addCondiments()`.

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a hook because the subclass can override this method, but doesn't have to.

Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false, depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

Let's run the Test Drive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee.

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();           ← Create a tea.
        CoffeeWithHook coffeeHook = new CoffeeWithHook();  ← A coffee.

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();                         ← And call prepareRecipe()
                                                       on both!

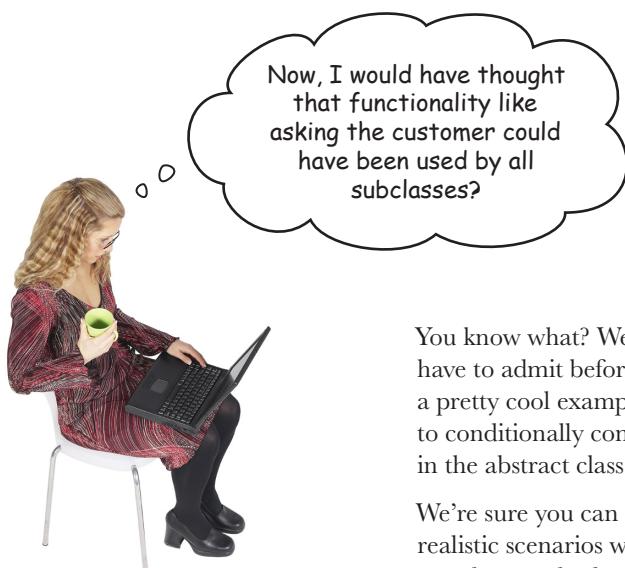
        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

And let's give it a run...

```
File Edit Window Help send-more-honesttea
%java BeverageTestDrive
Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y   ←
Adding Lemon

A steaming cup of tea, and yes,
of course we want that lemon!
) ←

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n   ←
%
And a nice hot cup of coffee,
but we'll pass on the waistline
expanding condiments. ←
```



You know what? We agree with you. But you have to admit before you thought of that, it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no Dumb Questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: What are hooks really supposed to be used for?

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass's implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen, or just happened. For instance, a hook method like `justReOrderedList()` allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen, a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: Does a subclass have to implement all the abstract methods in the `AbstractClass`?

A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

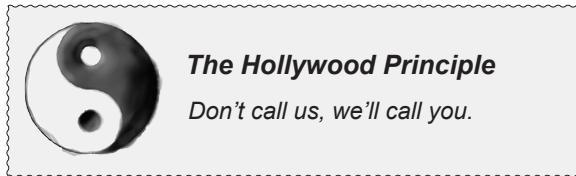
Q: It seems like I should keep my abstract methods small in number; otherwise, it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes 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 methods, easing the burden on the subclasses of your abstract class.

The Hollywood Principle

We've got another design principle for you; it's called the Hollywood Principle:

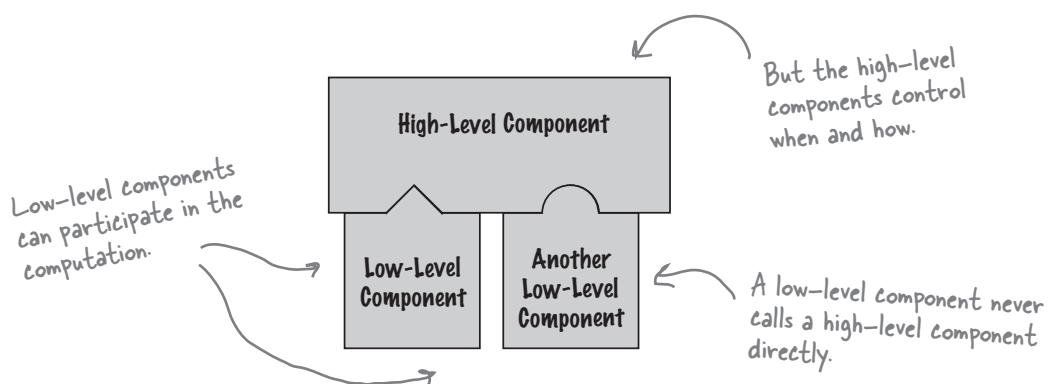


Easy to remember, right? But what has it got to do with OO design?

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.

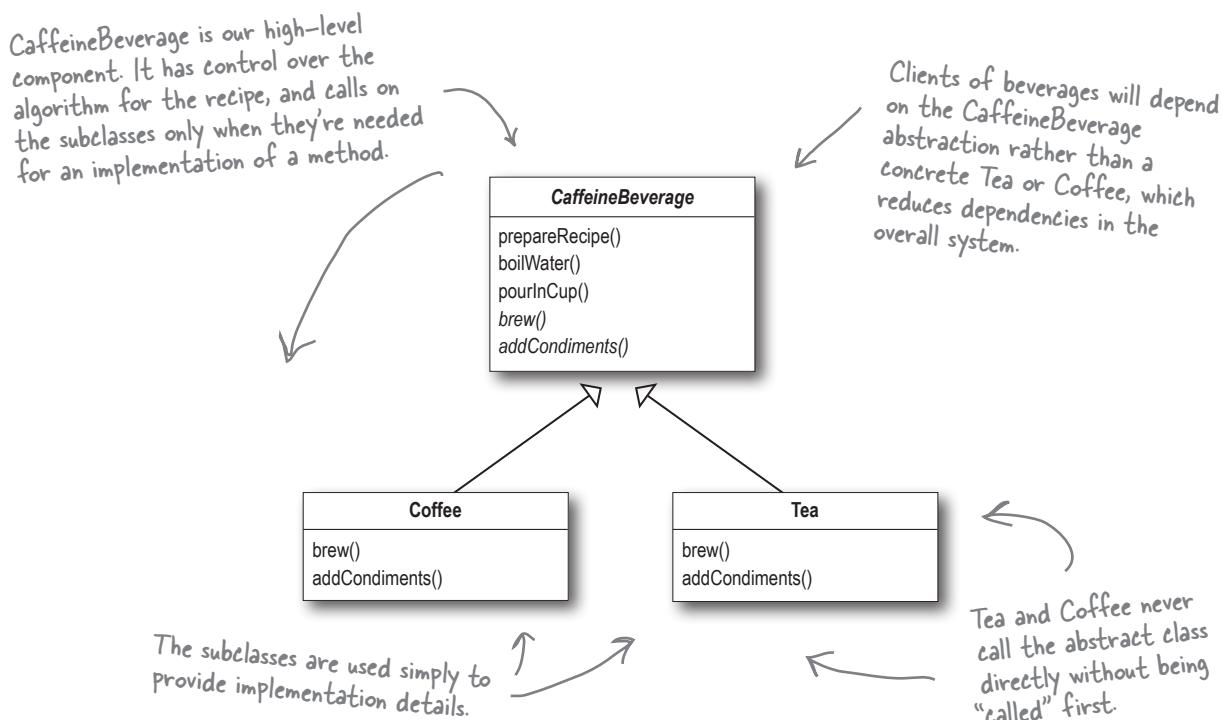
With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components a “don’t call us, we’ll call you” treatment.

You've heard me say it before, and I'll say it again: don't call me, I'll call you!



The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How? Let's take another look at our CaffeineBeverage design:



What other 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.



Match each pattern with its description:

Pattern

Template Method

Strategy

Factory Method

Description

Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.

Subclasses decide how to implement steps in an algorithm.

Subclasses decide which concrete classes to instantiate.

Template Methods in the Wild

The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild. You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.

This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...

In training, we study the classic patterns. However, when we are out in the real world, we must learn to recognize the patterns out of context. We must also learn to recognize variations of patterns, because in the real world a square hole is not always truly square.



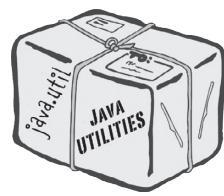
Sorting with Template Method

What's something we often need to do with arrays?
Sort them!

Recognizing that, the designers of the Java Arrays class have provided us with a handy template method for sorting. Let's take a look at how this method operates:

We actually have two methods here and they act together to provide the sort functionality.

We've pared down this code a little to make it easier to explain. If you'd like to see it all, grab the Java source code and check it out...



The first method, `sort()`, is just a helper method that creates a copy of the array and passes it along as the destination array to the `mergeSort()` method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The `mergeSort()` method contains the sort algorithm, and relies on an implementation of the `compareTo()` method to complete the algorithm. If you're interested in the nitty gritty of how the sorting happens, you'll want to check out the Java source code.

```
private static void mergeSort(Object src[], Object dest[],
    int low, int high, int off)
{
    // a lot of other code here
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
            ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    // and a lot of other code here
}
```

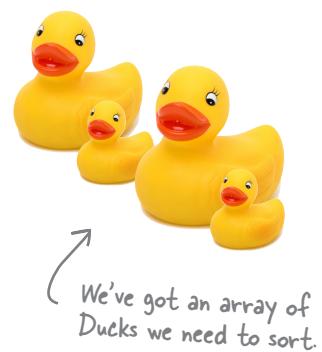
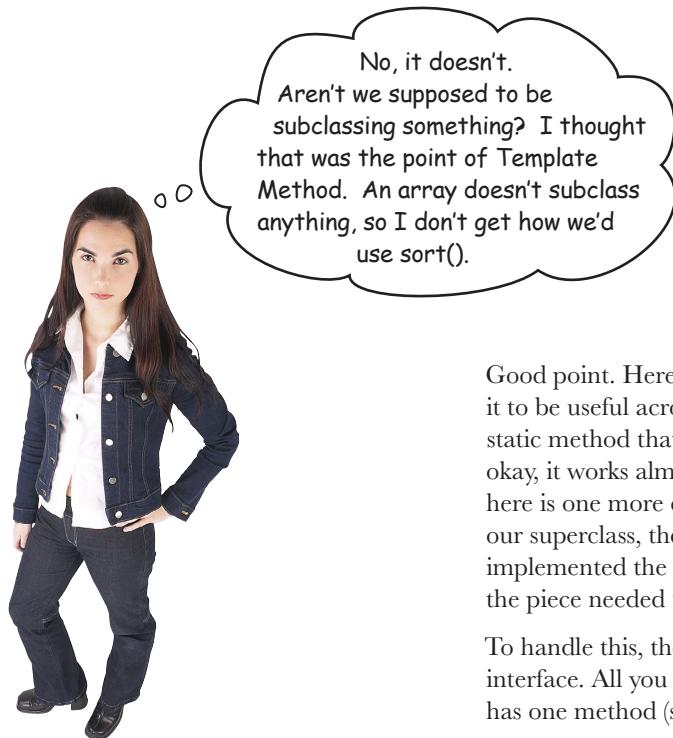
Think of this as the template method.

This is a concrete method, already defined in the Arrays class.

`compareTo()` is the method we need to implement to "fill out" the template method.

We've got some ducks to sort...

Let's say you have an array of ducks that you'd like to sort. How do you do it? Well, the sort template method in Arrays gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the `compareTo()` method... Make sense?

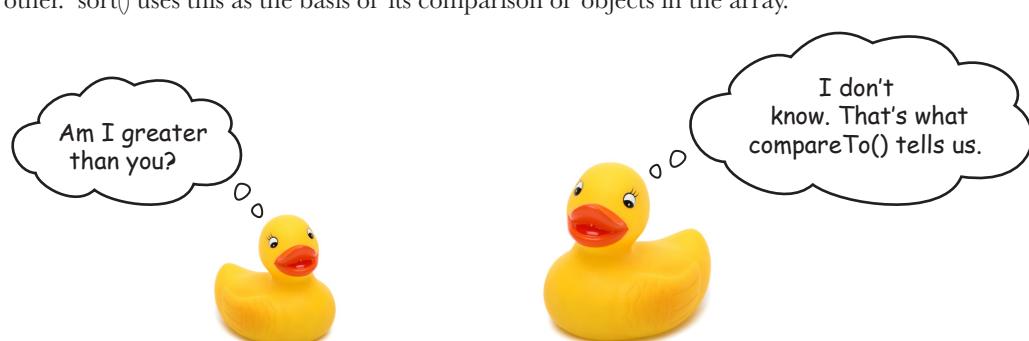


Good point. Here's the deal: the designers of `sort()` wanted it to be useful across all arrays, so they had to make `sort()` a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because `sort()` really isn't defined in our superclass, the `sort()` method needs to know that you've implemented the `compareTo()` method, or else you don't have the piece needed to complete the sort algorithm.

To handle this, the designers made use of the `Comparable` interface. All you have to do is implement this interface, which has one method (surprise): `compareTo()`.

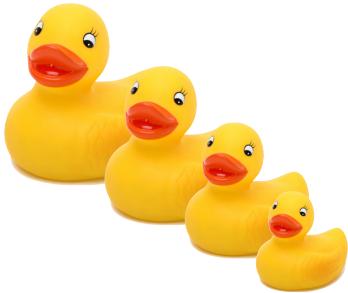
What is `compareTo()`?

The `compareTo()` method compares two objects and returns whether one is less than, greater than, or equal to the other. `sort()` uses this as the basis of its comparison of objects in the array.



Comparing Ducks and Ducks

Okay, so you know that if you want to sort Ducks, you're going to have to implement this compareTo() method; by doing that you'll give the Arrays class what it needs to complete the algorithm and sort your ducks.



Here's the duck implementation:

```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Object object) {
        Duck otherDuck = (Duck) object;
        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Remember, we need to implement the Comparable interface since we aren't really subclassing.

Our Ducks have a name and a weight!

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Let's sort some Ducks

Here's the test drive for sorting Ducks...

```

public class DuckSortTestDrive {

    public static void main(String[] args) {
        Duck[] ducks = {
            new Duck("Daffy", 8),
            new Duck("Dewey", 2),
            new Duck("Howard", 7),
            new Duck("Louie", 2),
            new Duck("Donald", 10),
            new Duck("Huey", 2)
        };
        System.out.println("Before sorting:");
        display(ducks); ← Let's print them to see
                           their names and weights.

        Arrays.sort(ducks); ← It's sort time!

        System.out.println("\nAfter sorting:");
        display(ducks); ← Let's print them (again) to see
                           their names and weights.
    }

    public static void display(Duck[] ducks) {
        for (Duck d : ducks) {
            System.out.println(d);
        }
    }
}

```

Notice that we call `Arrays`' static method `sort`, and pass it our Ducks.

Let the sorting commence!

```

File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive
Before sorting:
Daffy weighs 8
Dewey weighs 2      The unsorted Ducks
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2      The sorted Ducks
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%

```

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...



Behind
the Scenes

- 1 First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ...};
```

- 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

- 4 If the Ducks are not in sorted order, they're swapped with the concrete `swap()` method in `Arrays`:

```
swap();
```

- 5 The `sort()` method continues comparing and swapping Ducks until the array is in the correct order!

```
for (int i=low; i<high; i++) {
    ... compareTo() ...
    ... swap() ...
}
```

The `sort()` method controls the algorithm; no class can change this. `sort()` counts on a `Comparable` class to provide the implementation of `compareTo()`.

Duck
<code>compareTo()</code> <code>toString()</code>

No inheritance,
unlike a typical
template method.

Arrays
<code>sort()</code> <code>swap()</code>

there are no Dumb Questions

Q: Is this really the Template Method Pattern, or are you trying too hard?

A: The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps—and the Arrays sort is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints.

The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted.

So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.

Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?

A: You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way—we're using the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the entire algorithm. The algorithm that Arrays implements for sort is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.

Q: Are there other examples of template methods in the Java API?

A: Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the template method read(byte b[], int off, int len).



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?

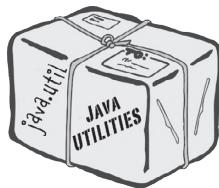


Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

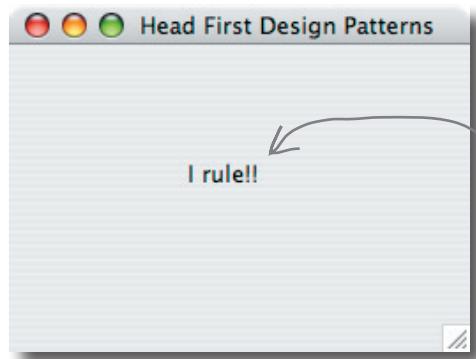
Swingin' with Frames

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:



```
public class MyFrame extends JFrame {  
  
    public MyFrame(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        this.setSize(300,300);  
        this.setVisible(true);  
    }  
  
    public void paint(Graphics graphics) {  
        super.paint(graphics);  
        String msg = "I rule!!";  
        graphics.drawString(msg, 100, 100);  
    }  
  
    public static void main(String[] args) {  
        MyFrame myFrame = new MyFrame("Head First Design Patterns");  
    }  
}
```



We're extending `JFrame`, which contains a method `update()` that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the `paint()` hook method.

Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls `paint()`. By default, `paint()` does nothing... it's a hook. We're overriding `paint()`, and telling the JFrame to draw a message in the window.

Here's the message that gets painted in the frame because we've hooked into the `paint()` method.

Applets

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:

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }
}
```

The `init` hook allows the applet to do whatever it wants to initialize the applet the first time.

```
public void start() {
    message = "Now I'm starting up...";
    repaint();
}
```

`repaint()` is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

```
public void stop() {
    message = "Oh, now I'm being stopped...";
    repaint();
}
```

The `start` hook allows the applet to do something when the applet is just about to be displayed on the web page.

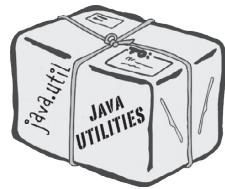
```
public void destroy() {
    // applet is going away...
}
```

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.

```
public void paint(Graphics g) {
    g.drawString(message, 5, 15);
}
```

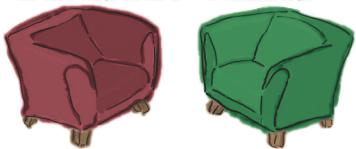
And the `destroy` hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

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:



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.

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.

How's that? My superclass is abstract.

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.

Got it. Don't call us, we'll call you...

Strategy:

You might be a little more efficient (just a little) and require fewer objects. *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, I guess... but, what about dependency?
You're way more dependent than me.

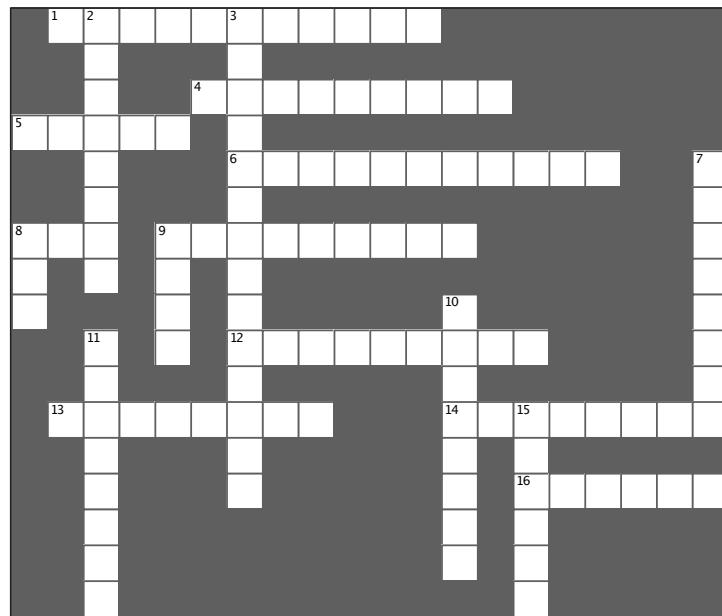
But you have to depend on methods implemented in your subclasses, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

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.



Design Patterns Crossword

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 _____.
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 give 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 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.

OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

OO Patterns

- Observer - Define a dependency between objects.
- Decorator - Adds additional responsibilities to an object without changing its interface.
- Factory Method - Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Singleton - Ensures a class has only one instance, and provides a global point of access to it.
- Command - Turn a request into a query command.
- Adapter - Encapsulates a request.
- Facade - Encapsulates a request.
- Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Strategy - Encapsulates a request.

And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

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.

Only talk to your friends.

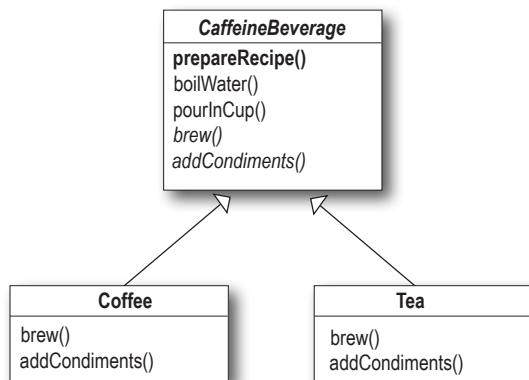
BULLET POINTS

- A “template method” defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method’s abstract class may define concrete methods, abstract methods, and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- You’ll see lots of uses of the Template Method Pattern in real-world code, but don’t expect it all (like any pattern) to be designed “by the book.”
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is a specialization of Template Method.



Sharpen your pencil Solution

Draw the new class diagram now that we've moved `prepareRecipe()` into the `CaffeineBeverage` class:



* WHO DOES WHAT? SOLUTION *

Match each pattern with its description:

Pattern	Description
Template Method	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.



Design Patterns Crossword Solution

It's that time again...

