





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 <code>PizzaStore</code> . 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 <i>Pizza</i> . 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.

NOTE

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.

NOTE

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.

NOTE

If you override an implemented method, then your base class wasn't really an

abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.

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

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate 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.

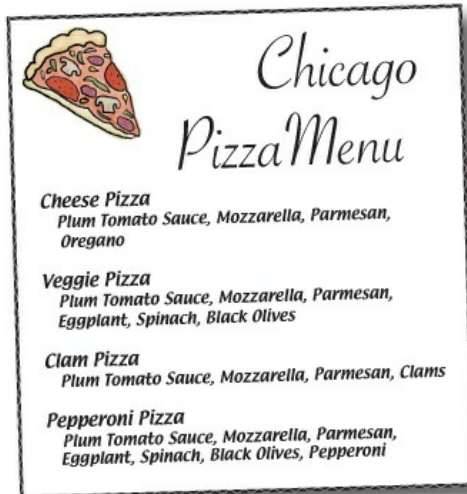
Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *procedures*, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase 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:



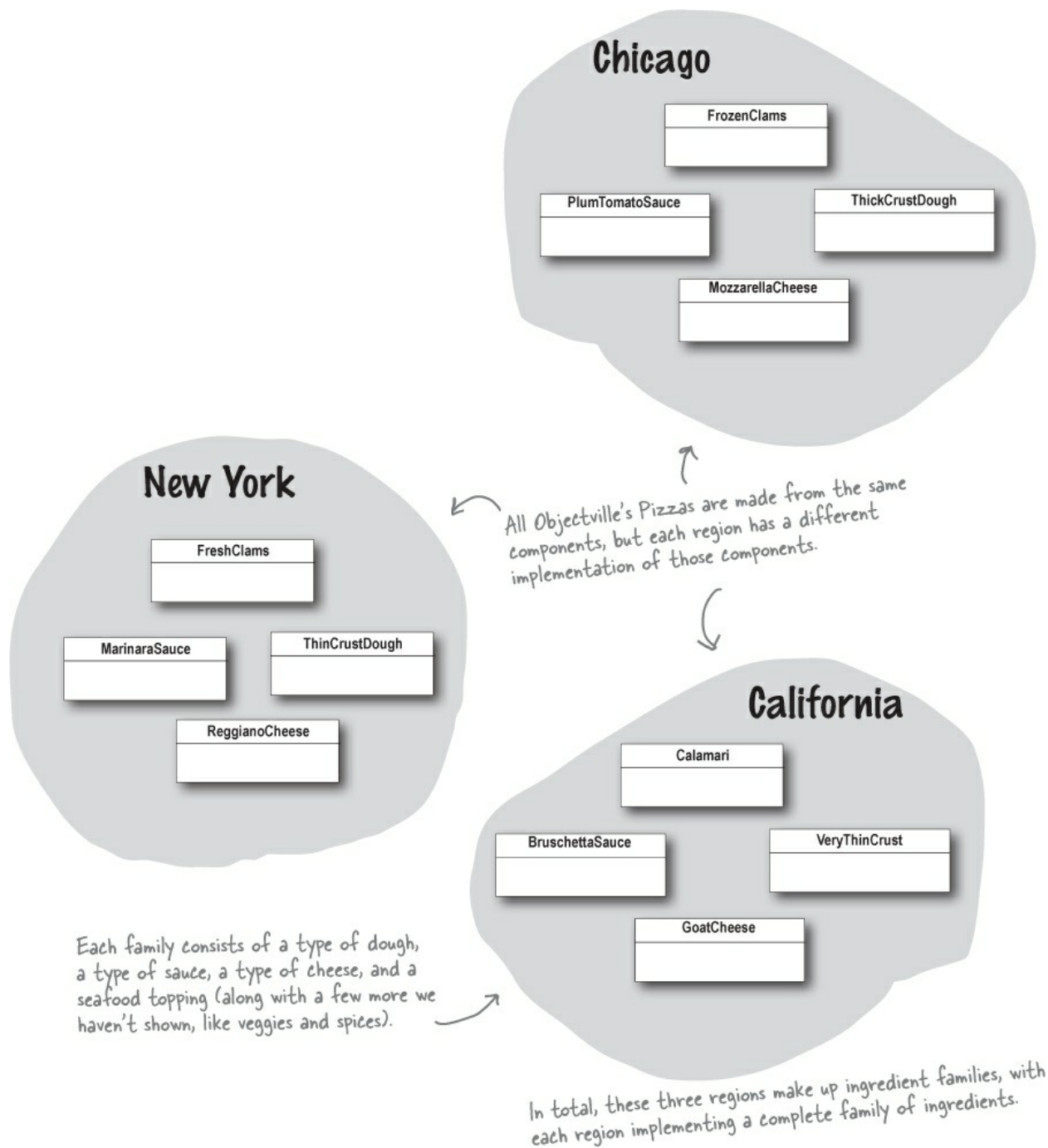
We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.



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.



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();  
  
}
```

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

Lots of new classes here, one per ingredient.

NOTE

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:

- ① Build a factory for each region. To do this, you'll create a subclass of `PizzaIngredientFactory` that implements each create method.
- ② 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.
- ③ 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();  
    }  
}
```

For each ingredient in the ingredient family, we create the New York version.

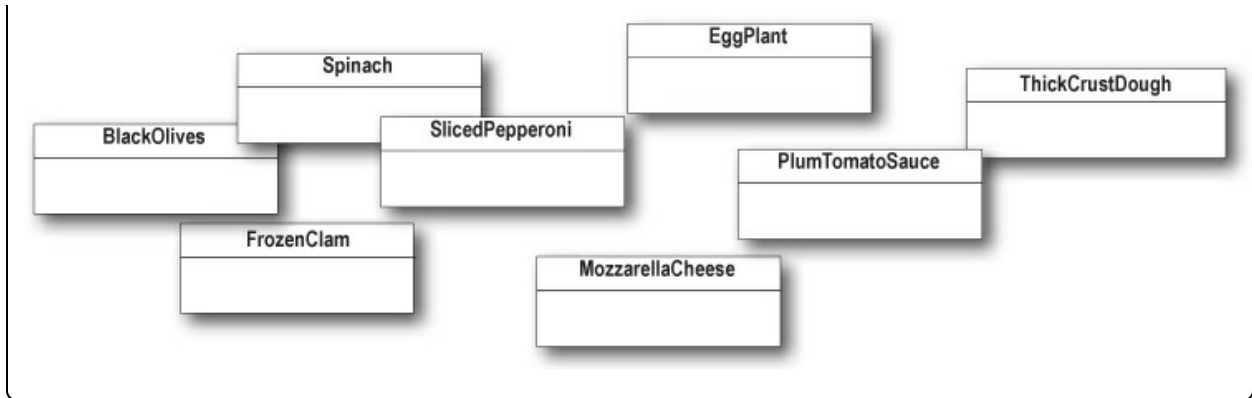
For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

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

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

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
    }
}

```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

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.

BRAIN POWER

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

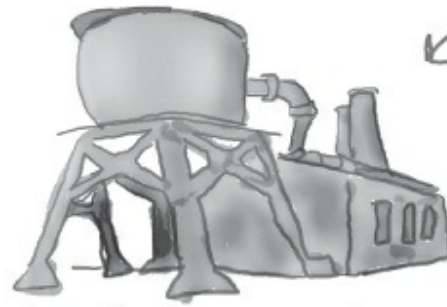
from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts — such as different regions, different operating systems, or different look and feels.

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.

From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.

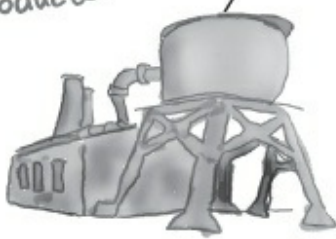
We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.



Defines the interface.

Objectville Abstract Ingredient Factory

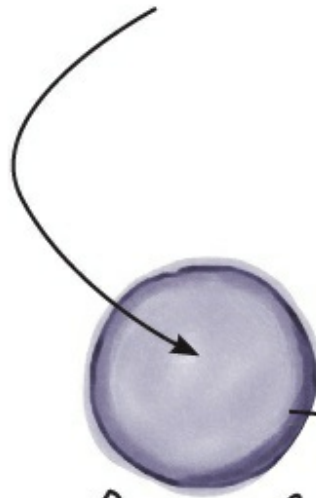
Provides implementations for products.



New York



Chicago



PizzaStore

Pizza made with ingredients produced by concrete factory.



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...



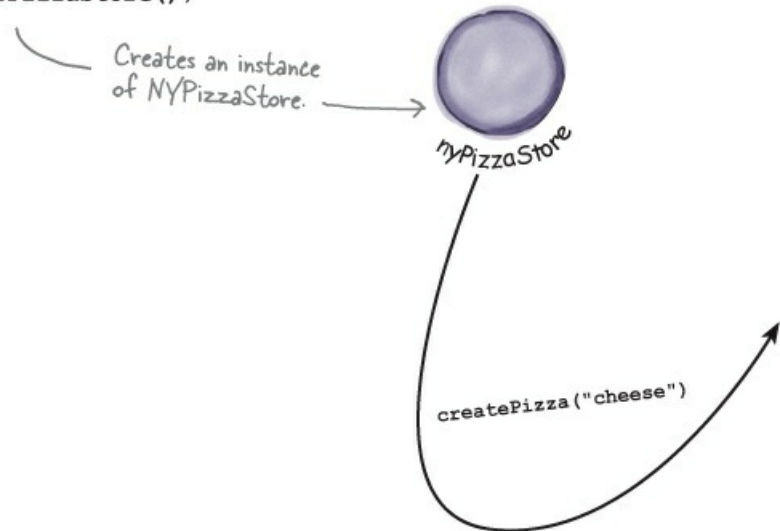
Behind the Scenes



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();
```



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



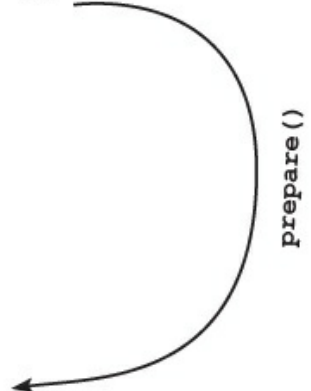
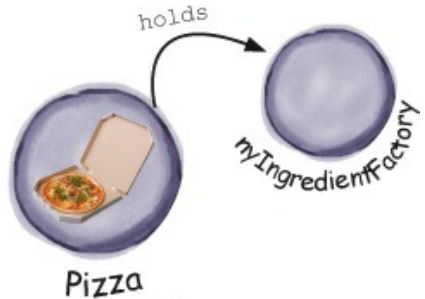
Behind the Scenes

④ When the `createPizza()` method is called, that's when our ingredient factory gets involved:

The ingredient factory is chosen and instantiated in the PizzaStore and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.



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();  
}
```

Thin crust
Marinara
Reggiano

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.

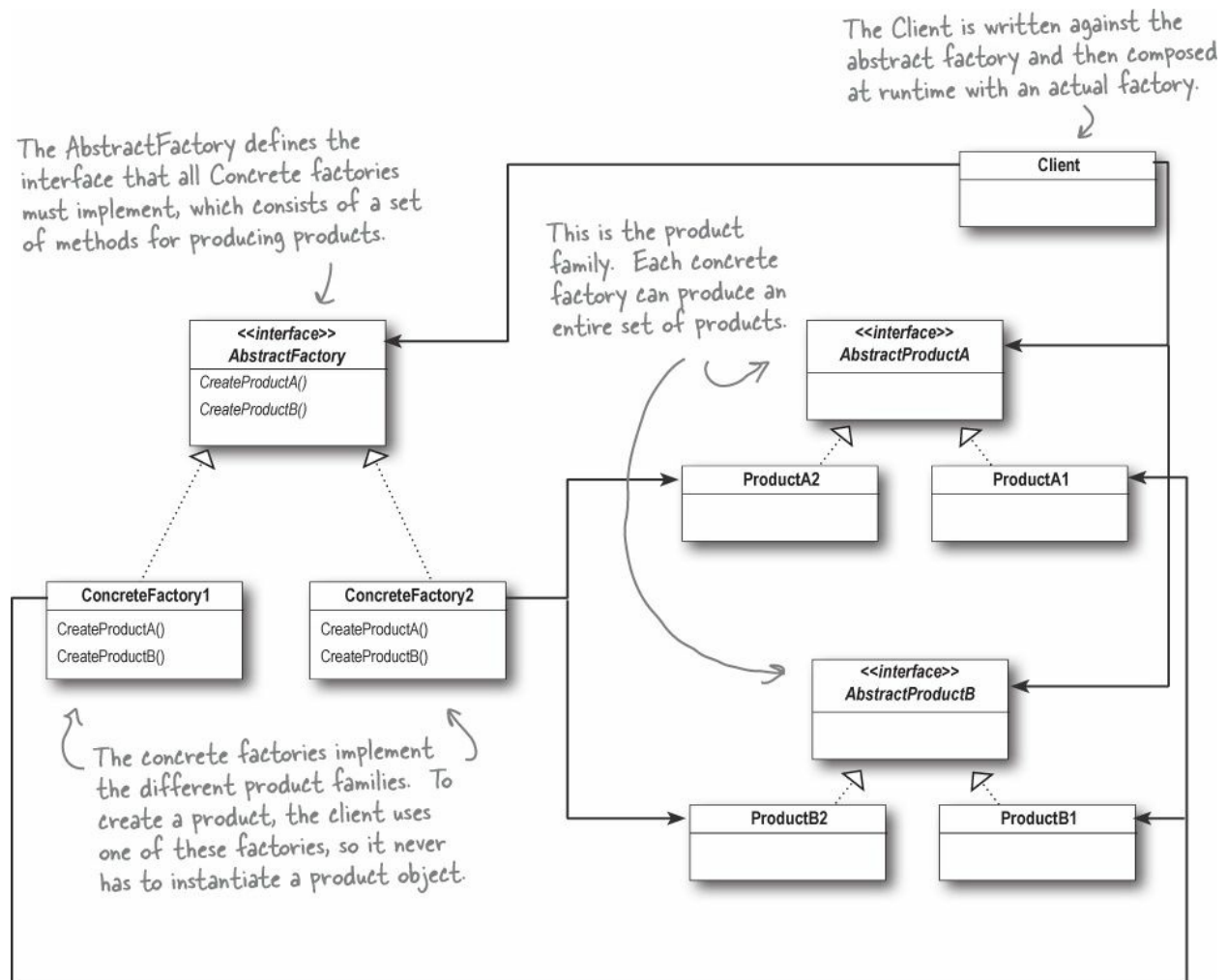
Abstract Factory Pattern defined

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:

NOTE

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

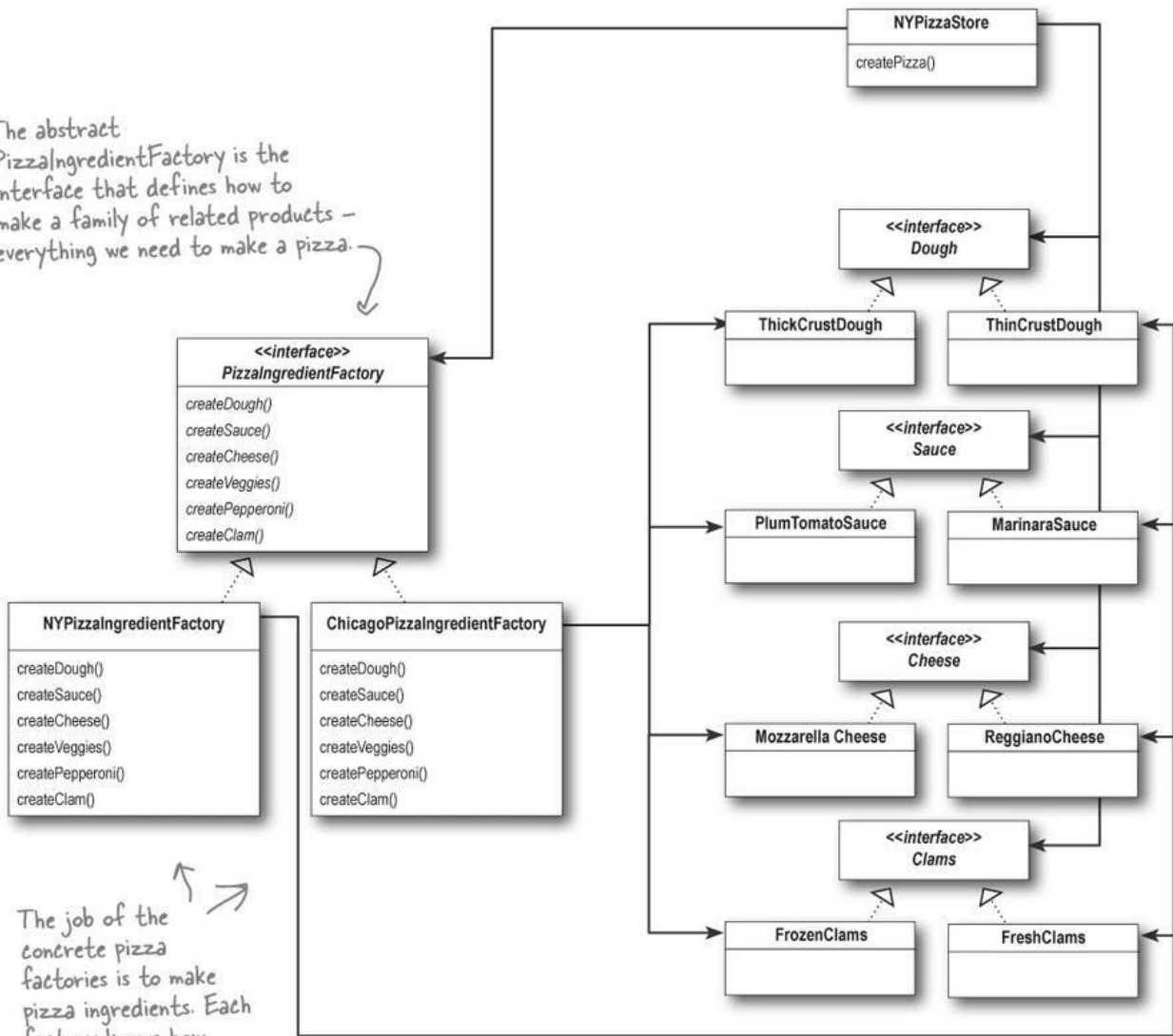
We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's 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 clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.

The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products - everything we need to make a pizza.



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.



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...

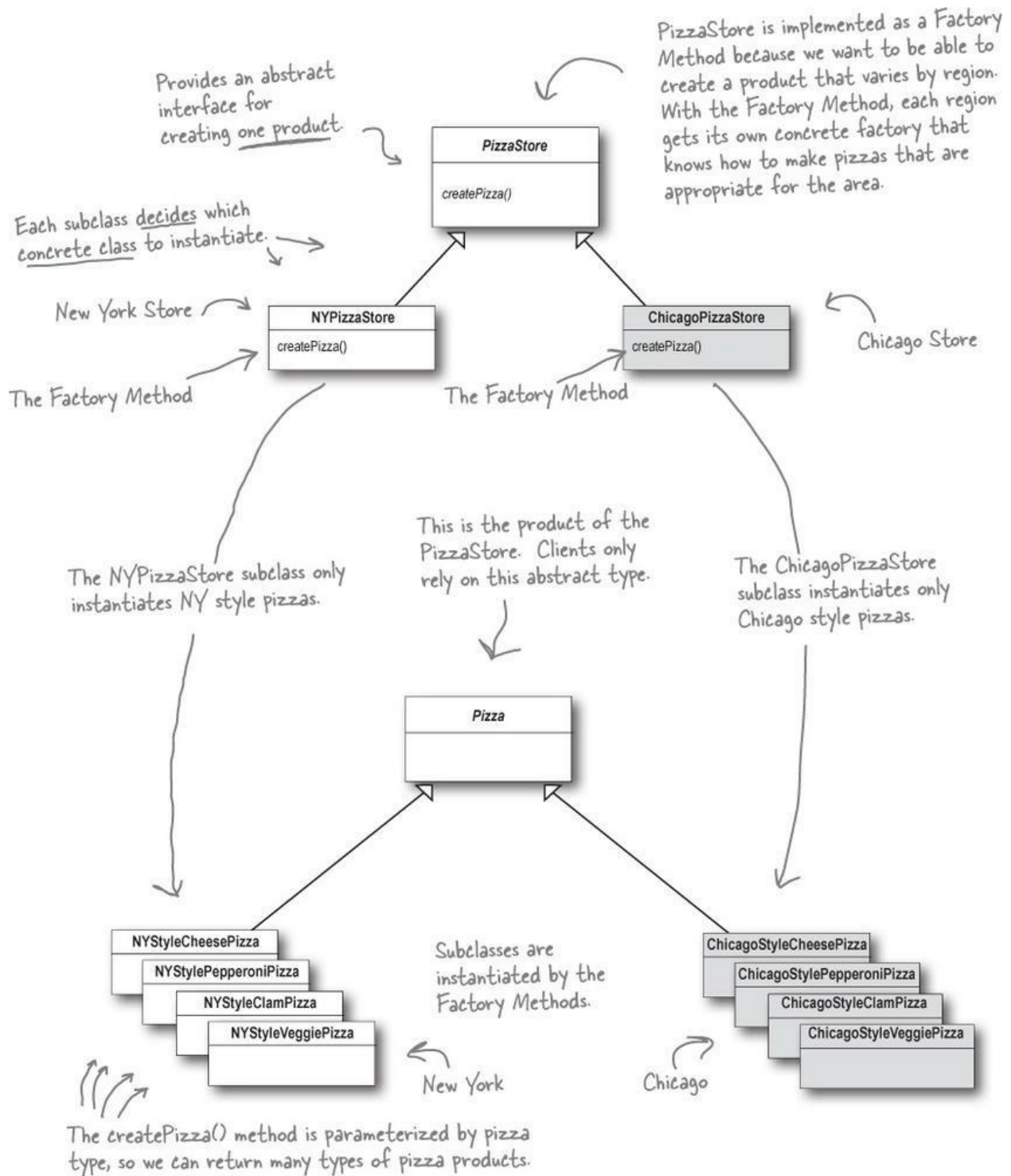
Factory Method: ...while in my case I usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

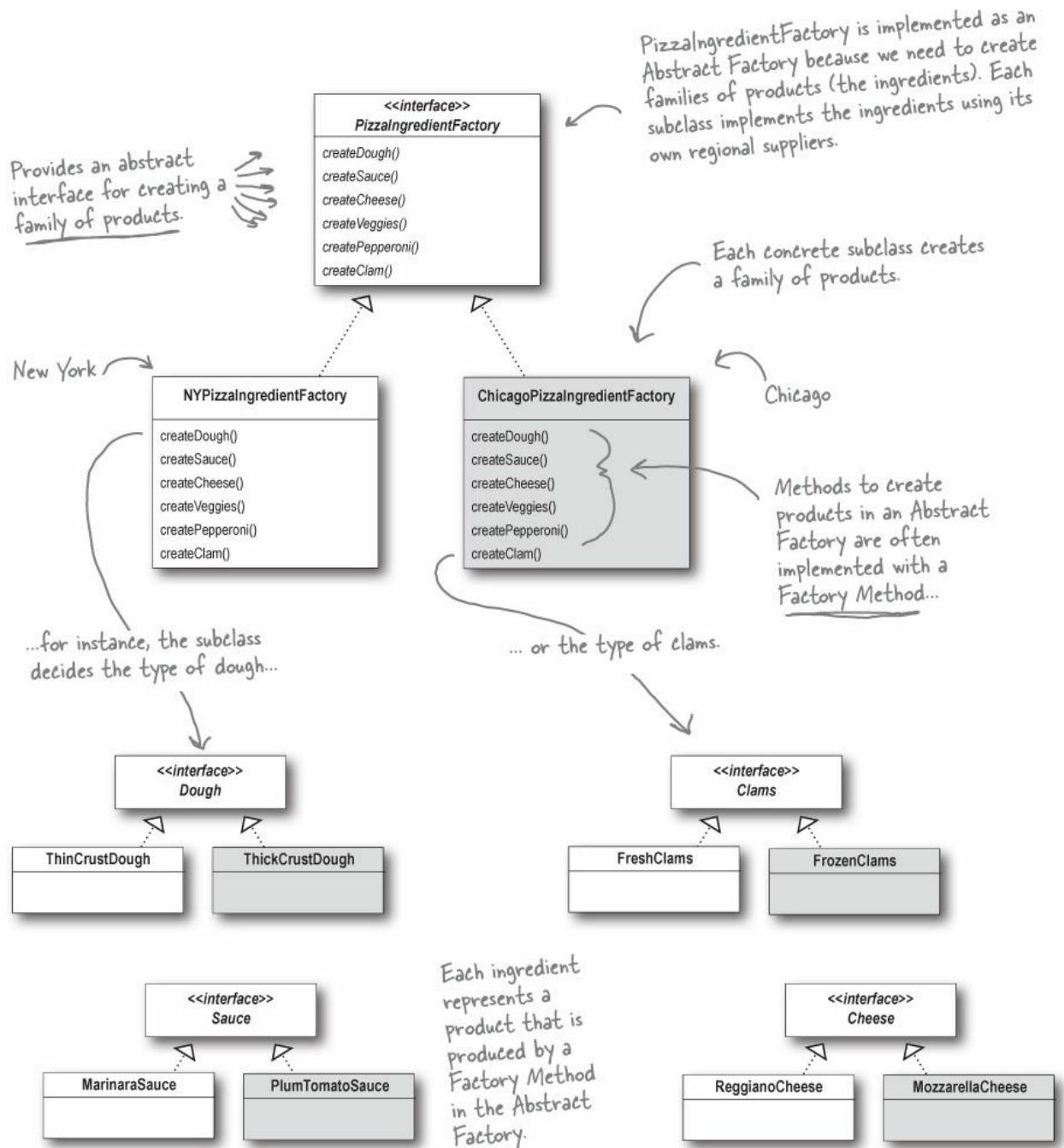
HeadFirst: It sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. You both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, 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





NOTE

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.

OO Basics

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.

abstraction

encapsulation

polymorphism

inheritance

We have a new principle that guides us to keep things abstract whenever possible.

OO Patterns

Abstract Factory - Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method - Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to the subclasses.

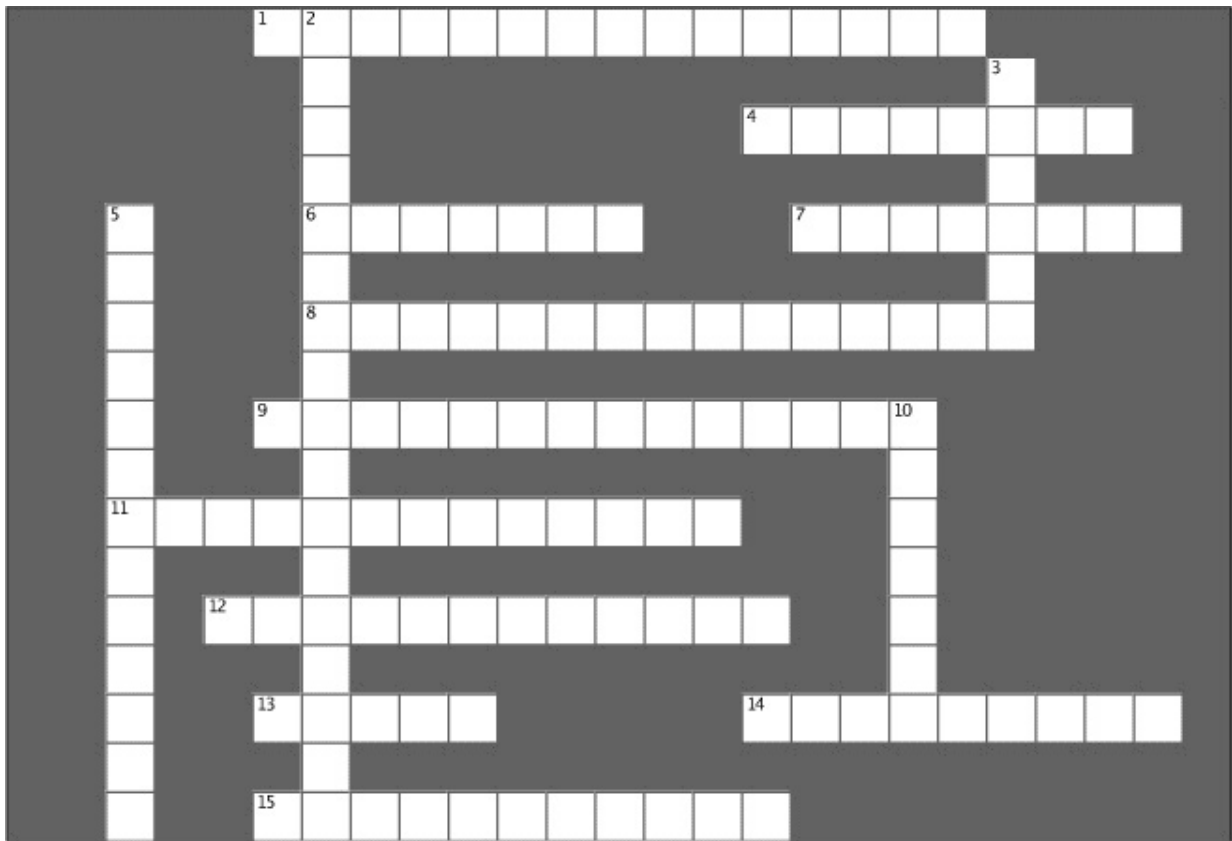
Both of these new patterns encapsulate object creation and lead to more decoupled, flexible designs.

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	Down
<p>1. In Factory Method, each franchise is a _____.</p> <p>4. In Factory Method, who decides which class to instantiate?</p> <p>6. Role of PizzaStore in Factory Method Pattern.</p> <p>7. All New York style pizzas use this kind of cheese.</p> <p>8. In Abstract Factory, each ingredient factory is a _____.</p> <p>9. When you use new, you are programming to an _____.</p> <p>11. createPizza() is a _____ (two words).</p> <p>12. Joel likes this kind of pizza.</p> <p>13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction.</p> <p>14. When a class instantiates an object from a</p>	<p>2. We used _____ in Simple Factory and Abstract Factory, and inheritance in Factory Method.</p> <p>3. Abstract Factory creates a _____ of products.</p> <p>5. Not a REAL factory pattern, but handy nonetheless.</p> <p>10. Ethan likes this kind of pizza.</p>

concrete class, it's _____ on that object.

15. All factory patterns allow us to _____ object creation.

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();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

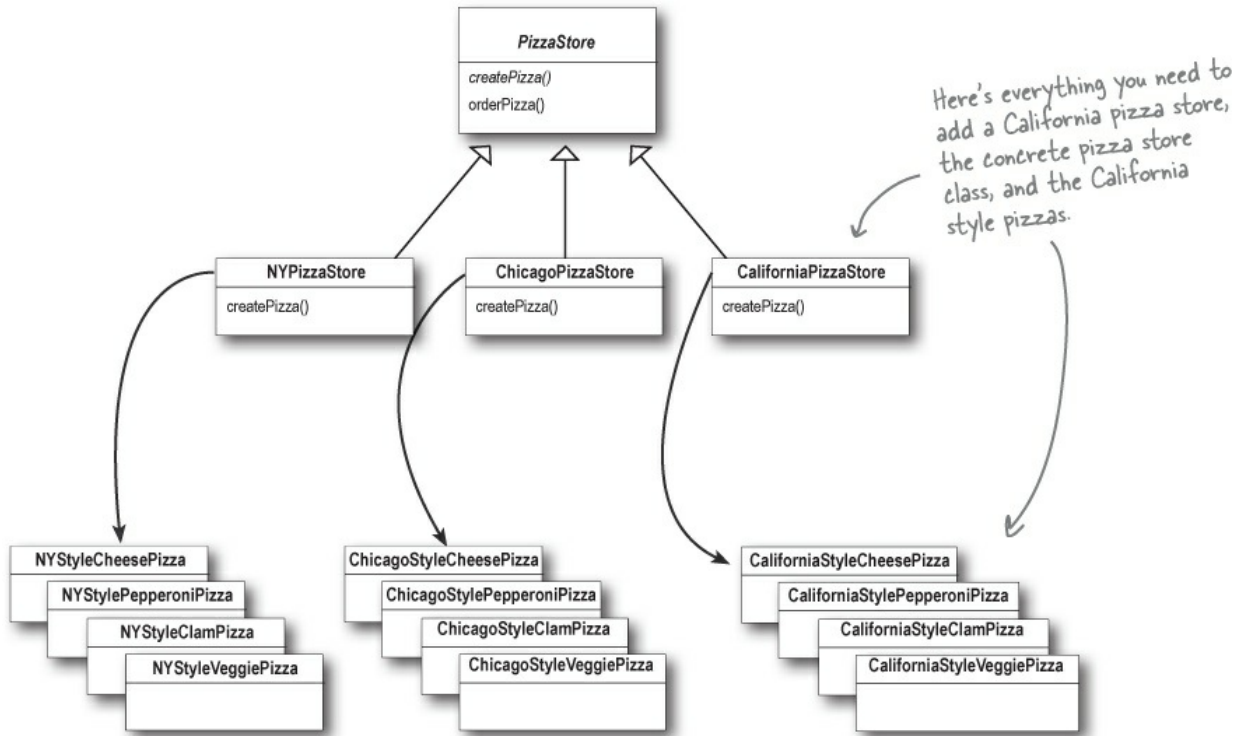
For the Chicago pizza store, we just have to make sure we create Chicago style pizzas...

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new CaliforniaStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

and for the California pizza store, we create California style pizzas.

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!

NOTE

Here are our suggestions...

Mashed Potatoes with Roasted Garlic

BBQ Sauce

Artichoke Hearts

M&M's

Peanuts

A very dependent PizzaStore

SHARPEN YOUR PENCIL SOLUTION

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();
    }

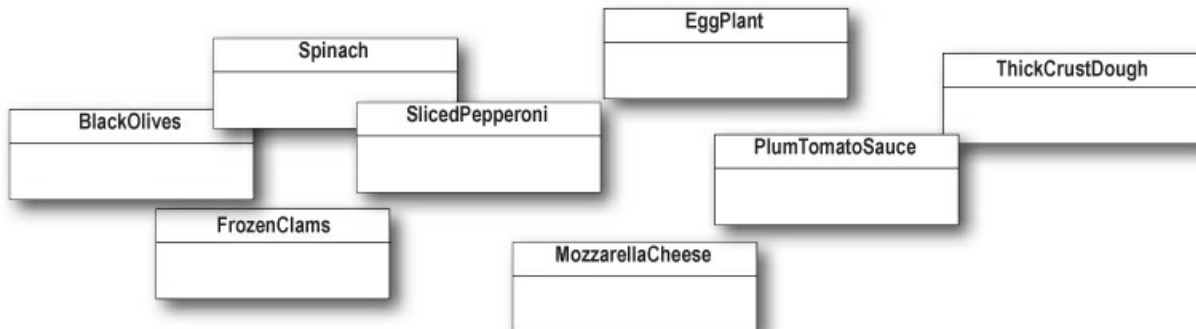
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                               new Spinach(),
                               new Eggplant() };
        return veggies;
    }

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

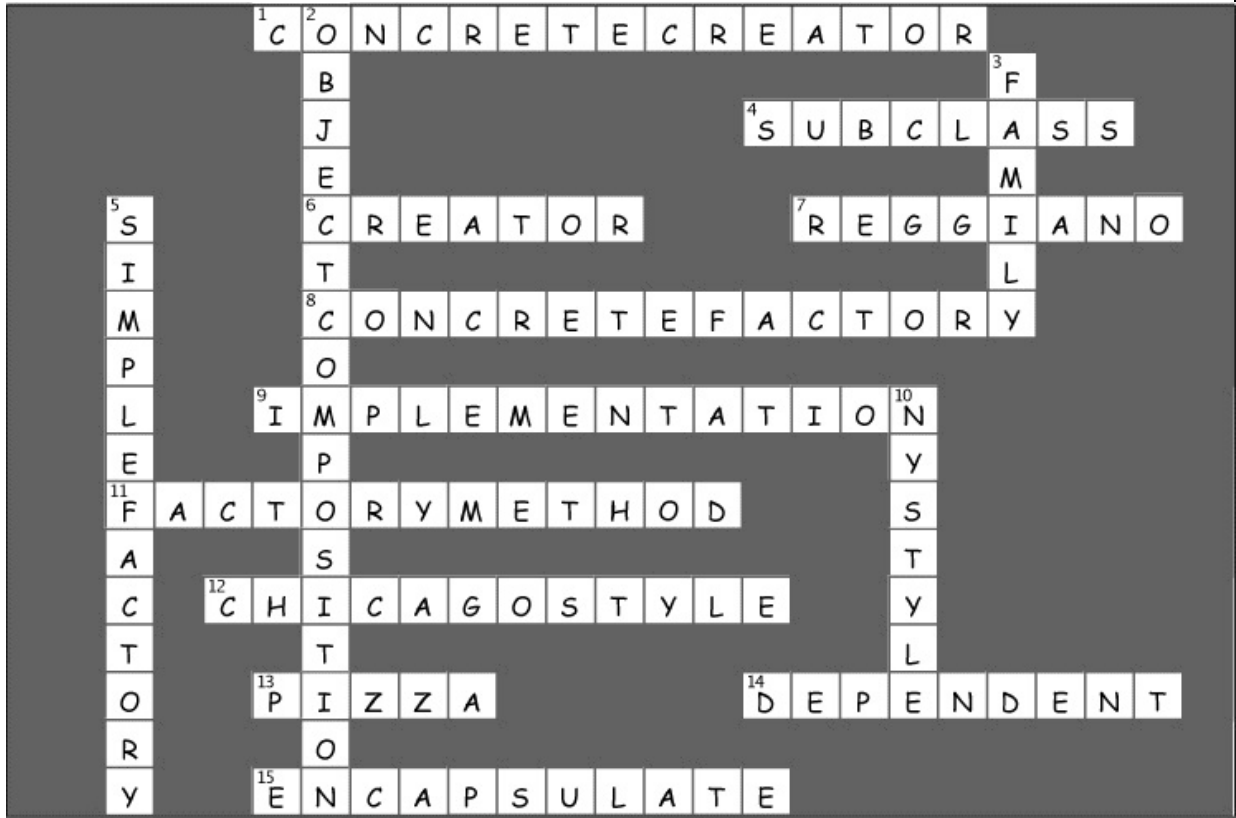
    public Clams createClam() {
        return new FrozenClams();
    }
}

```



DESIGN PATTERNS CROSSWORD SOLUTION

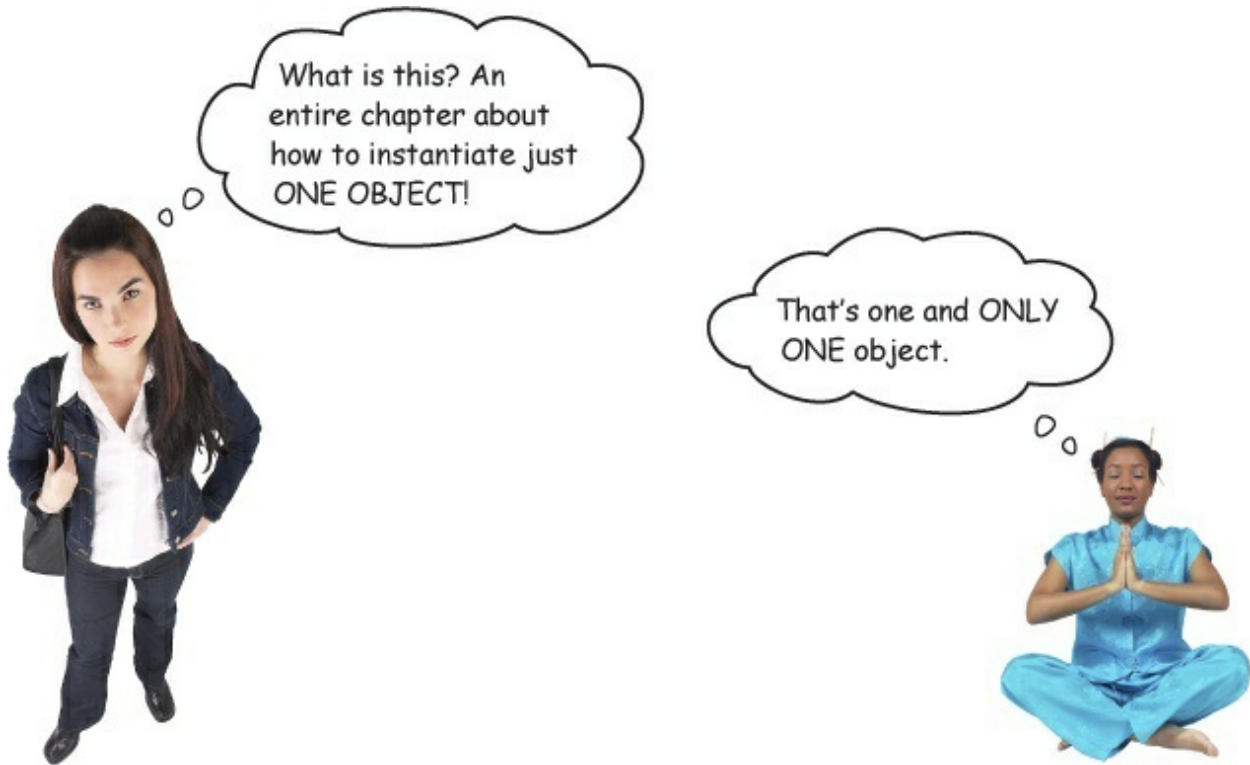
It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter. Here's the solution.



Chapter 5. The Singleton Pattern: One of a Kind Objects



Our next step 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?	<code>new MyObject();</code>
And, what if another object wanted to create a MyObject? Could it call new on MyObject again?	Yes, of course.
So as long as we have a class, can we always instantiate it one or more times?	Yes. Well, only if it's a public class.
And if not?	Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.
Hmm, interesting. Did you know you could do this?	No, I'd never thought of it, but I guess it makes sense because it is a legal definition.
	

<p>What does it mean?</p>	<p>I suppose it is a class that can't be instantiated because it has a private constructor.</p>
<p>Well, is there ANY object that could use the private constructor?</p>	<p>Hmm, I think the code in MyClass is the only code that could call it. But that doesn't make much sense.</p>
<p>Why not?</p>	<p>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()".</p>
<p>Okay. It was just a thought.</p> <p>What does this mean?</p> 	<p>MyClass is a class with a static method. We can call the static method like this:</p> <pre>MyClass.getInstance();</pre>
<p>Why did you use MyClass, instead of some object name?</p>	<p>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.</p>
<p>Very interesting. What if we put things together.</p> <p>Now can I instantiate a MyClass?</p> 	<p>Wow, you sure can.</p>
<p>So, now can you think of a second way to instantiate an object?</p>	<pre>MyClass.getInstance();</pre>
<p>Can you finish the code so that only ONE instance of MyClass is ever created?</p>	<p>Yes, I think so... (You'll find the code on the next page.)</p>

Dissecting the classic Singleton Pattern implementation

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

Let's rename MyClass to Singleton.

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

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

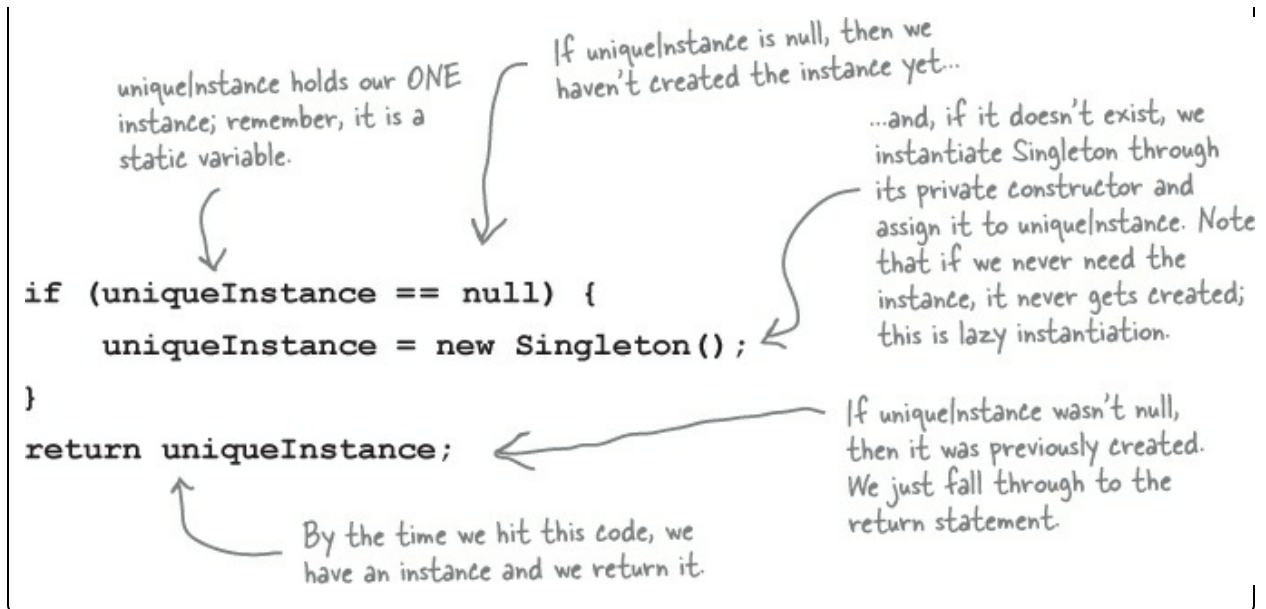
The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

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

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



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.

BRAIN POWER

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

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

SHARPEN YOUR PENCIL

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

singleton?

```
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:

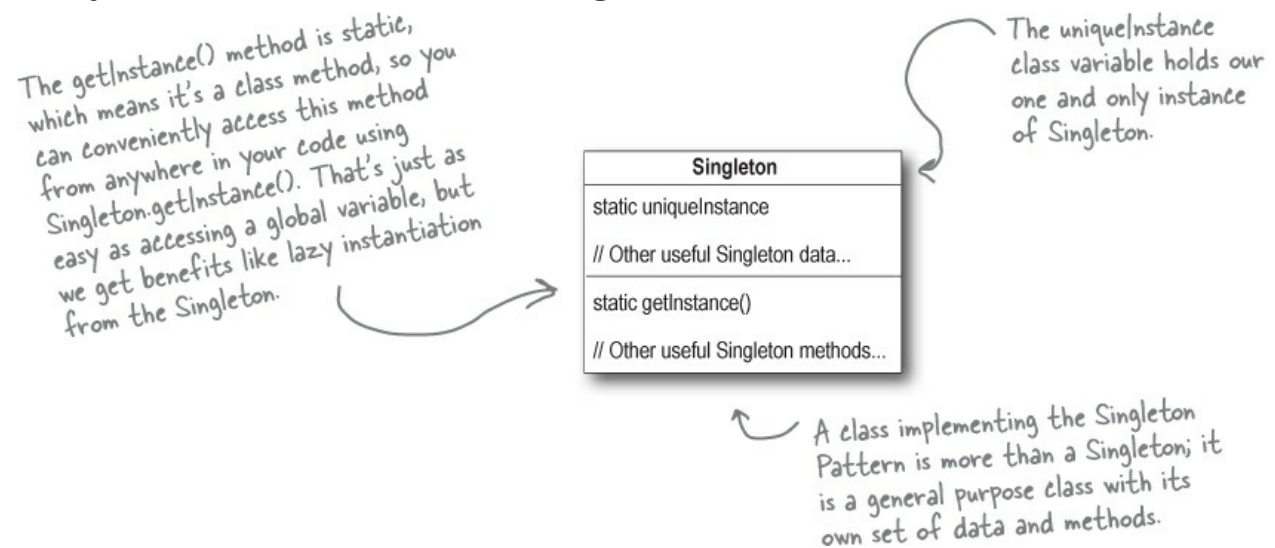
NOTE

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:



Houston, Hershey, PA 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();
```

Make sure you check your answer in **BE the JVM Solution** before continuing!

<pre>public static ChocolateBoiler getInstance() {</pre>	<pre> if (uniqueInstance == null) {</pre>	<pre> uniqueInstance = new ChocolateBoiler();</pre>	<p>Thread One</p>		<p>Thread Two</p>	<p>Value of uniqueInstance</p>
<pre> }</pre>	<pre> return uniqueInstance;</pre>	<pre>}</pre>				

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.

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;
    }
}
```

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

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

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

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

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

Let's check out the code:

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

    private Singleton() {}

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

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

*The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

If performance is an issue in your use of the `getInstance()` method then this method of implementing the Singleton can drastically reduce the overhead.

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: Q: For such a simple pattern consisting of only one class, Singletons sure seem to have some problems.

A: 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: 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: 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: 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: 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.

RELAX

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

Prior to Java 1.2, a bug in the garbage collector allowed Singletons to be prematurely collected if there was no global reference to them. In other words, you could create a Singleton and if the only reference to the Singleton was in the Singleton itself, it would be collected and destroyed by the garbage collector. This leads to confusing bugs because after the Singleton is "collected," the next call to getInstance() 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: 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: 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: Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

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

A: 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 Principles

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

OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

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

OO Patterns

- Singleton - Ensure a class only has one instance and provide a global point of access to it.
- Factory Method
- Define an

NOTE
As you've seen, despite its apparent simplicity, there are a lot of details involved in the

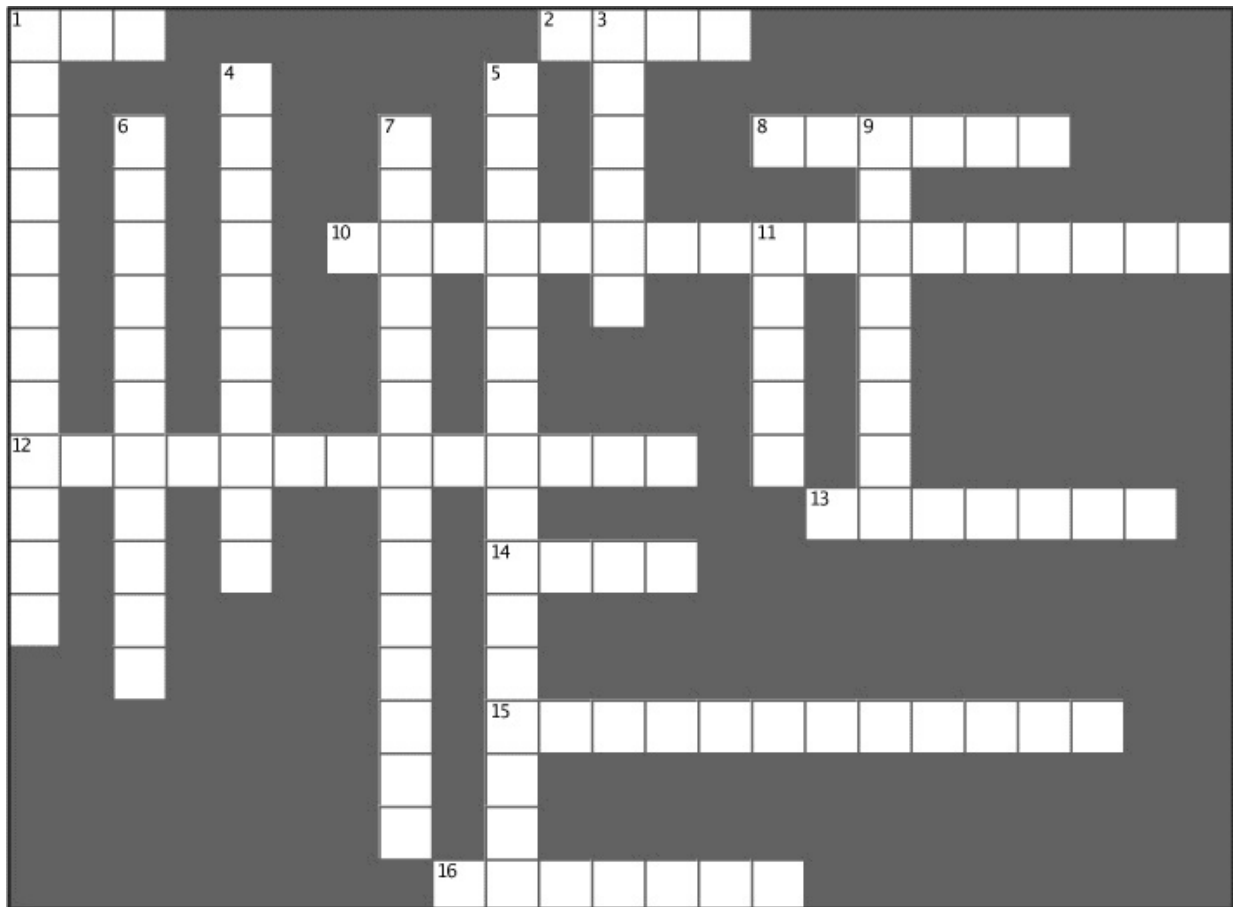
Singleton's implementation. After reading this chapter, though, you are ready to go out and use Singleton in the wild.

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.

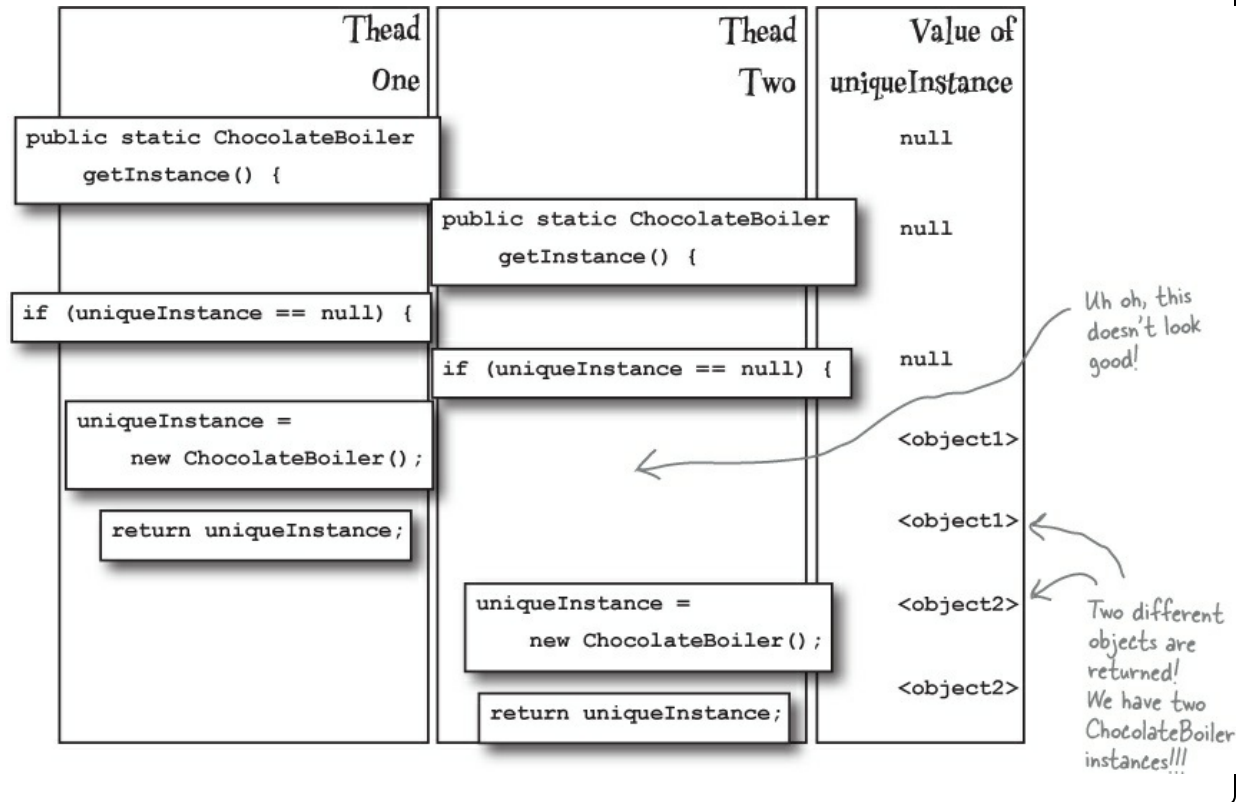
DESIGN PATTERNS CROSSWORD

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	Down
<p>1. It was “one of a kind.”</p> <p>2. Added to chocolate in the boiler.</p> <p>8. An incorrect implementation caused this to overflow.</p> <p>10. Singleton provides a single instance and _____ (three words).</p> <p>12. Flawed multi-threading approach if not using Java 5 or later.</p> <p>13. Chocolate capital of the USA.</p> <p>14. One advantage over global variables: _____ creation.</p> <p>15. Company that produces boilers.</p> <p>16. To totally defeat the new constructor, we have to declare the constructor _____.</p>	<p>1. Multiple _____ can cause problems.</p> <p>3. A Singleton is a class that manages an instance of _____.</p> <p>4. If you don’t need to worry about lazy instantiation, you can create your instance _____.</p> <p>5. Prior to Java 1.2, this can eat your Singletons (two words).</p> <p>6. The Singleton was embarrassed it had no public _____.</p> <p>7. The classic implementation doesn’t handle this.</p> <p>9. Singleton ensures only one of these exists.</p> <p>11. The Singleton Pattern has one.</p>

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 _____

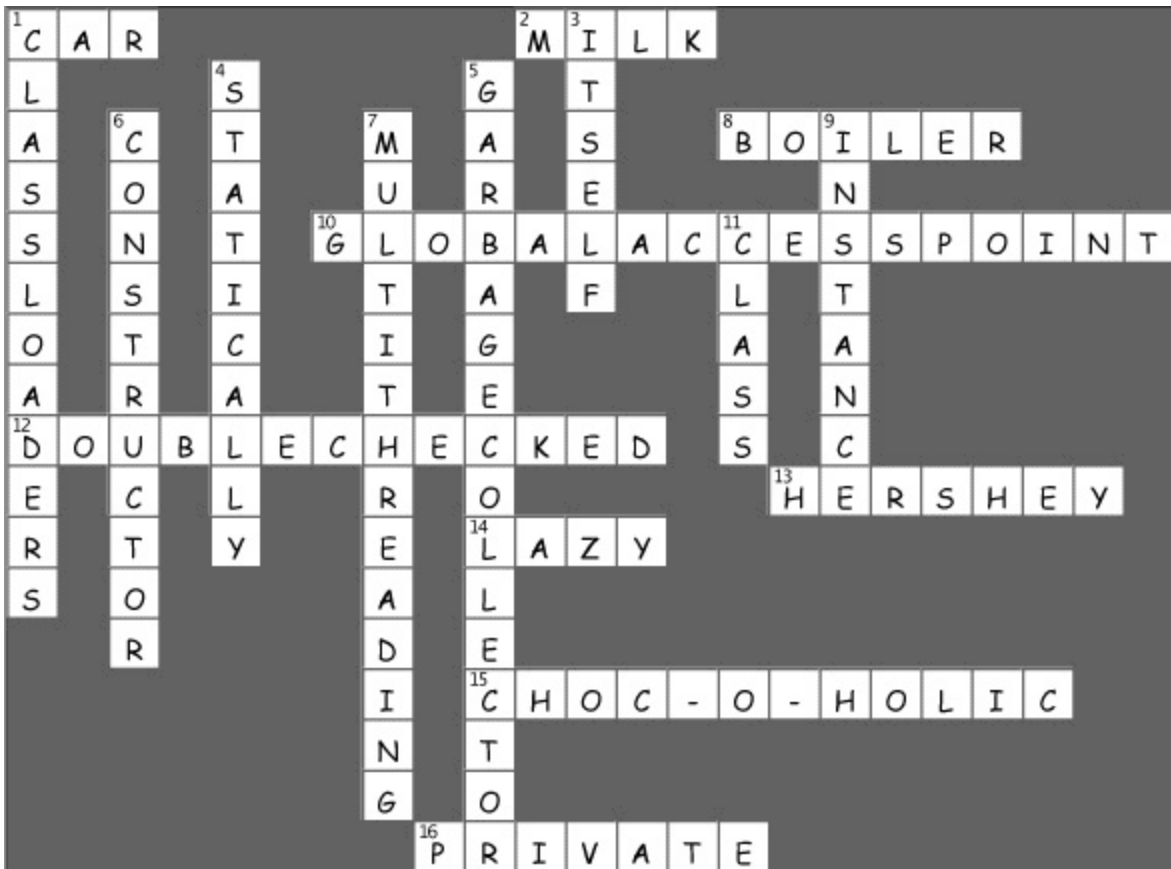
the instance would cause no concerns. This solution would work as well as the synchronized _____

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. _____

DESIGN PATTERNS CROSSWORD SOLUTION



Chapter 6. The Command Pattern: Encapsulating Invocation



In this chapter, we take encapsulation to a whole new level: we're going to encapsulate method invocation. That's right; by encapsulating 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.

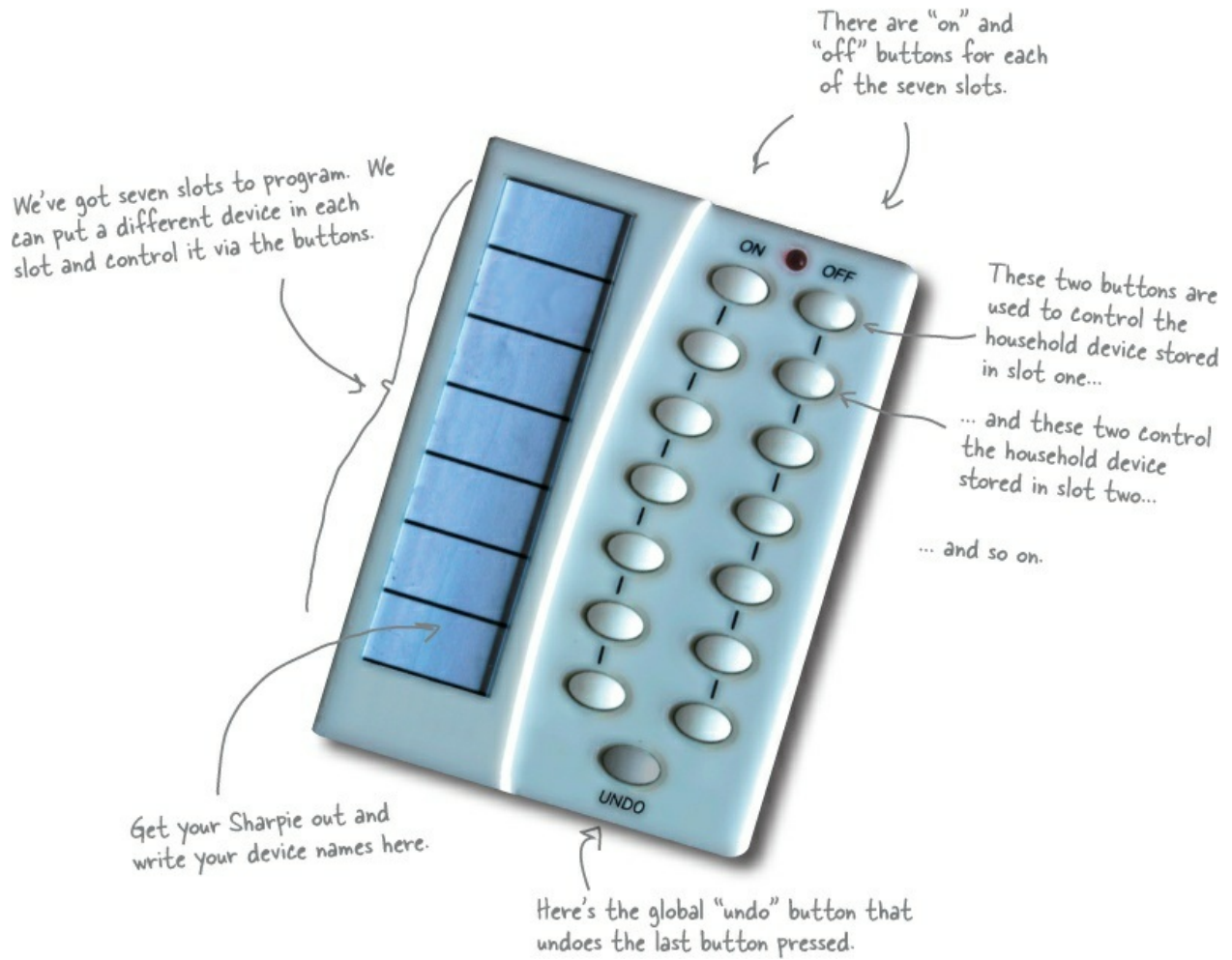
Sincerely,

Bill Thompson

Bill "X-10" Thompson, CEO

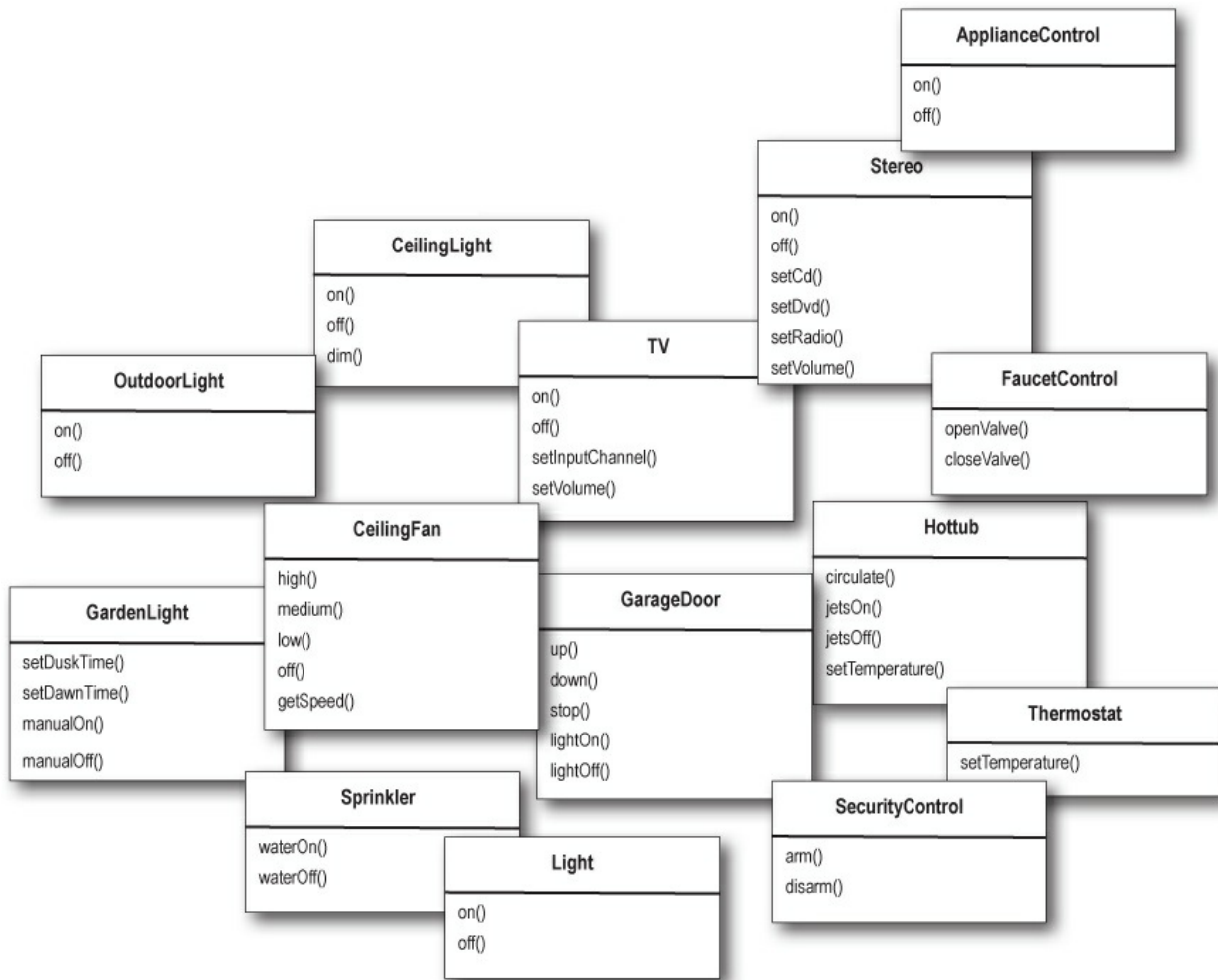


Free hardware! Let's check out the Remote Control...



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.

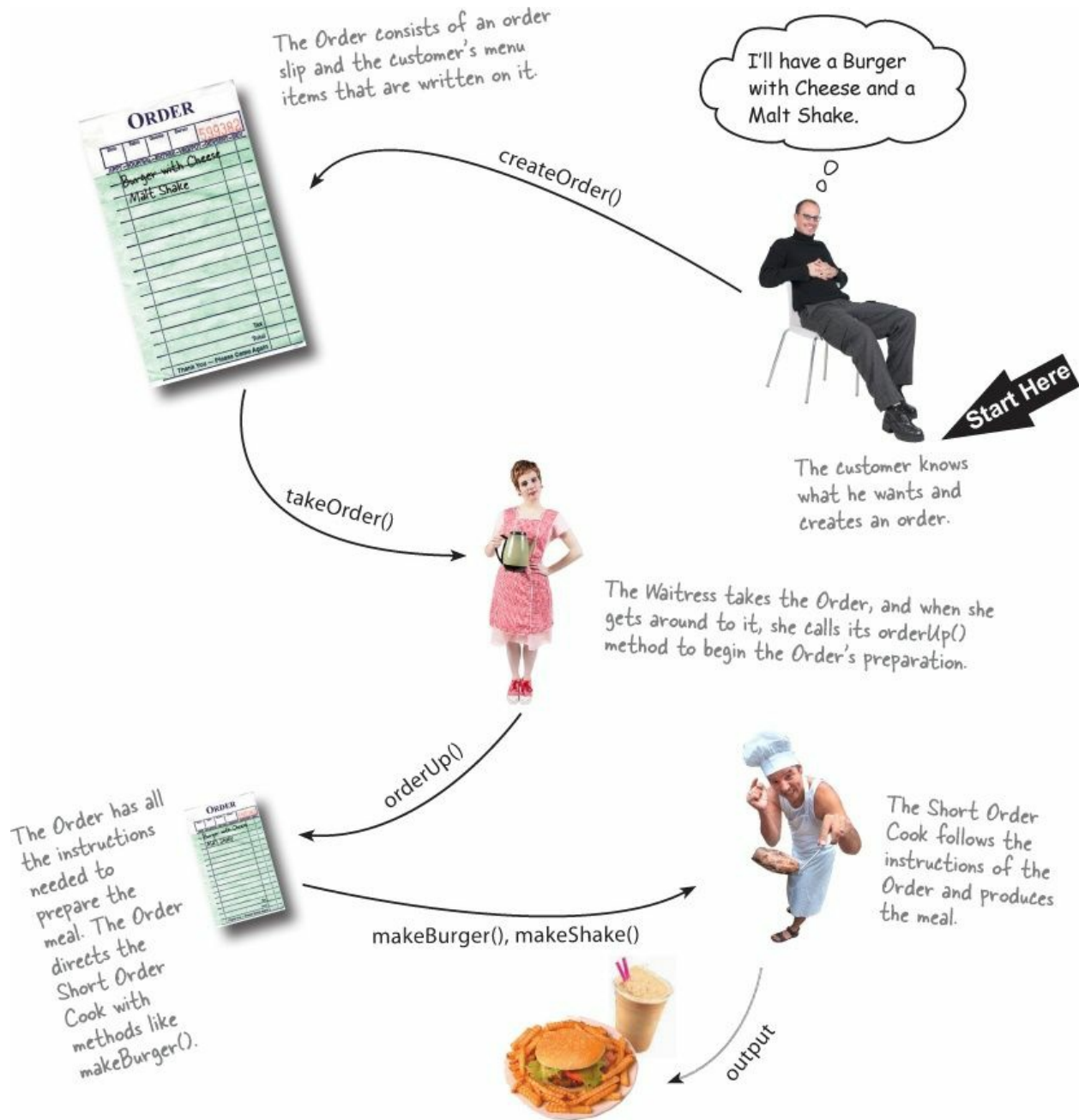
Checking in at the Objectville Diner...

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!"



NOTE

Okay, in real life a waitress would probably care what is on the Order Slip and who cooks it, but this is Objectville... work with us here!

The 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

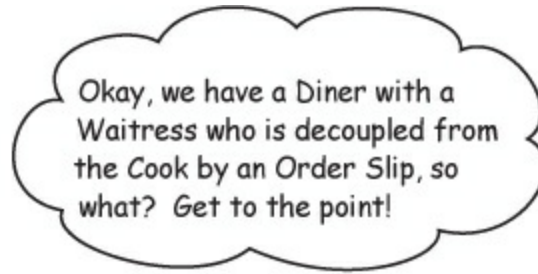
already discussed, in Objectville, the Waitress really isn't worried about what's on the order or who is going to prepare it; she just knows Order Slips have an `orderUp()` method she can call to get the job done.

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.



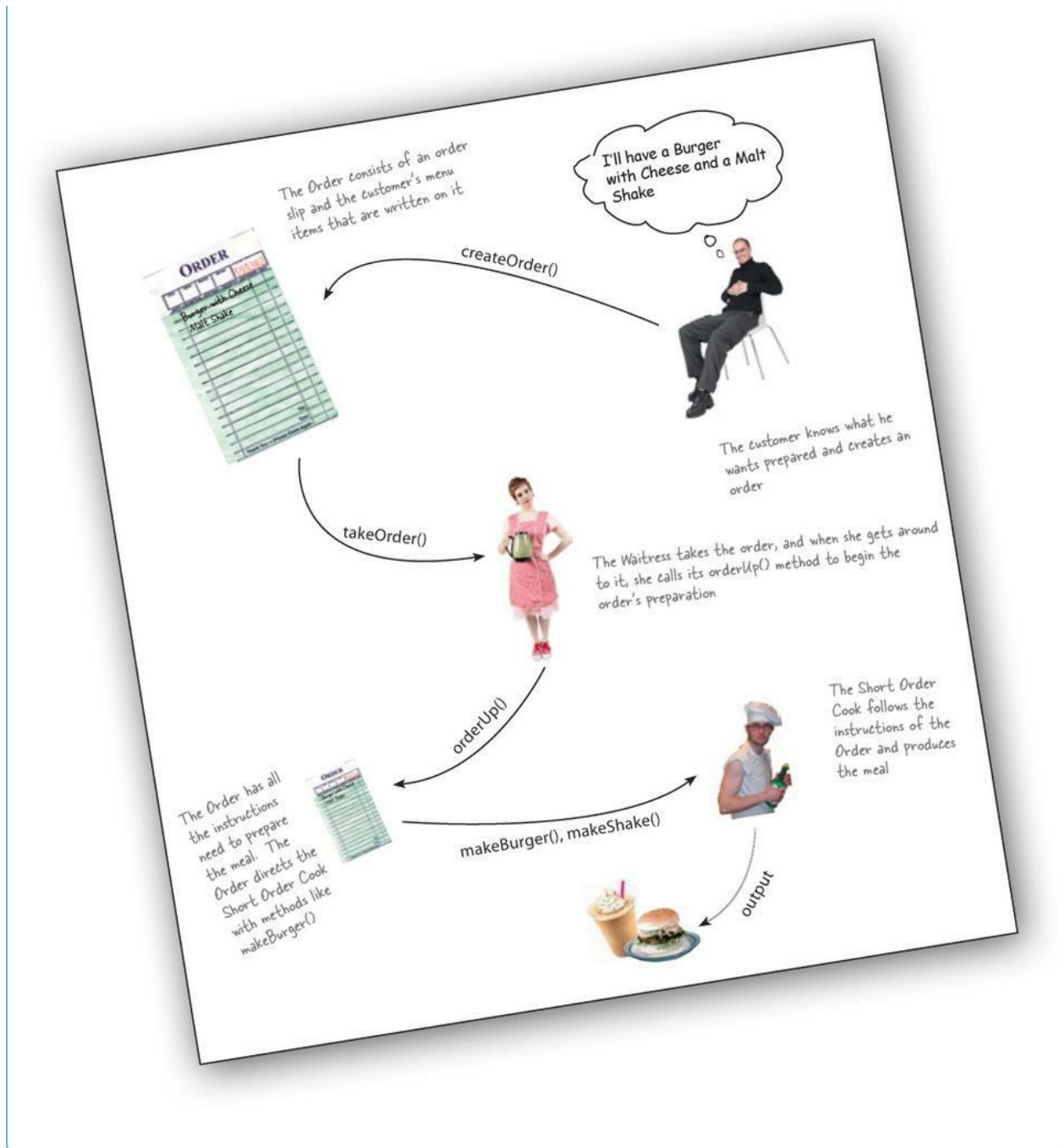
Patience, we're getting there...

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...

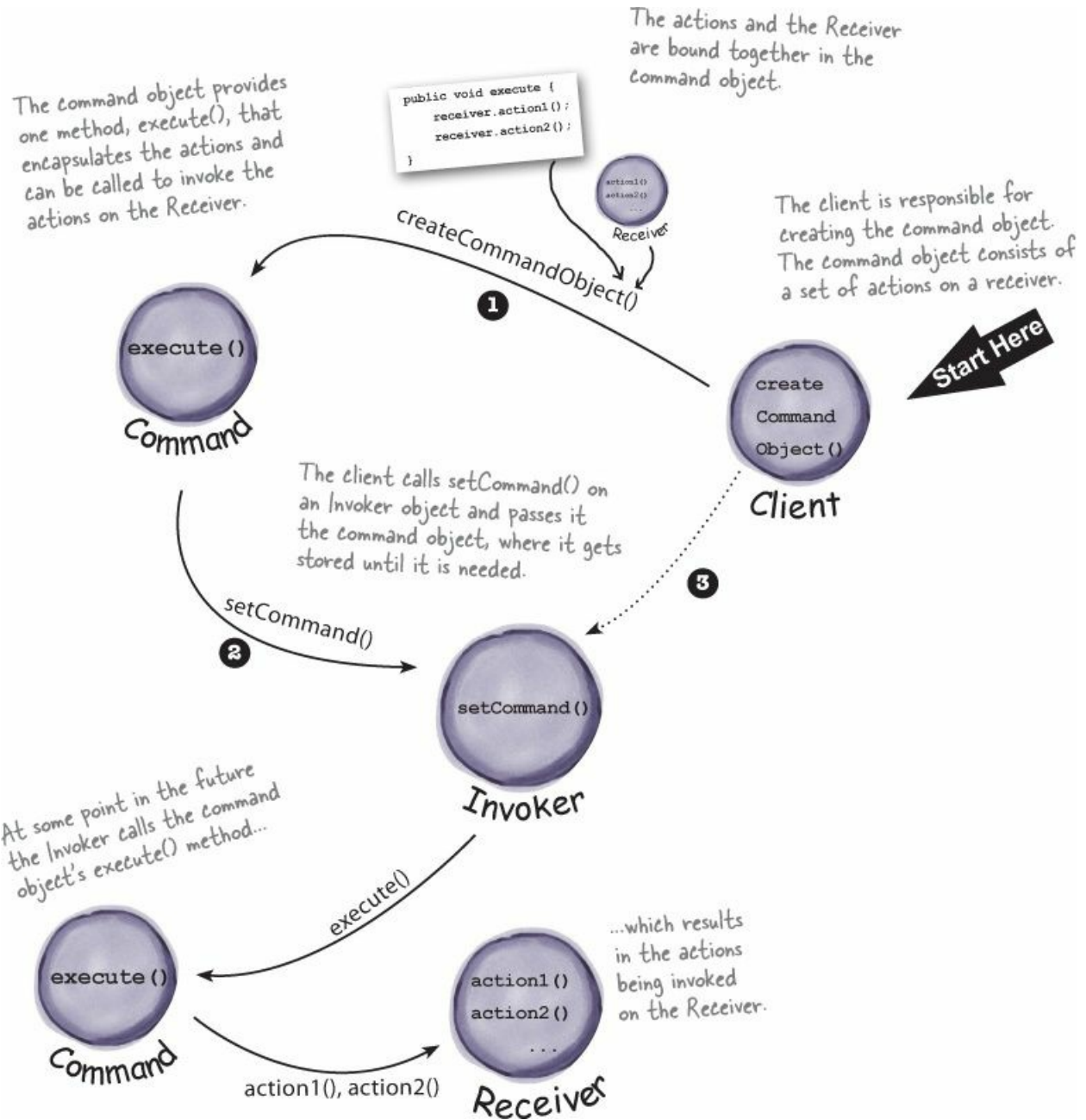
BRAIN POWER

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.

WHO DOES WHAT?

Match the diner objects and methods with the corresponding names from the Command Pattern.

Diner	Command Pattern
Waitress	Command
Short Order Cook	execute()
orderUp()	Client
Order	Invoker
Customer	Receiver
takeOrder()	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()`.

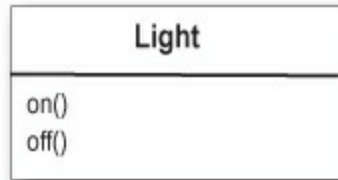
Here's the Command interface:

```
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:



```
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.

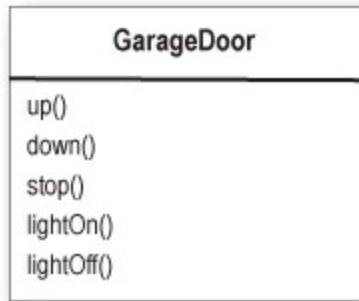
```

File Edit Window Help DinerFoodYum
%java RemoteControlTest
Light is On
%

```

SHARPEN YOUR PENCIL

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. →

```
File Edit Window Help GreenEggs&Ham  
%java RemoteControlTest
```

The Command Pattern defined

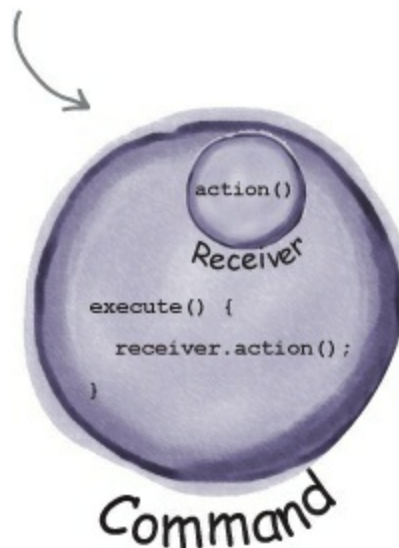
You've done your time in the Objectville Diner, you've partly implemented the remote control API, and in the process you've got a fairly good picture of how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.

Let's start with its official definition:

NOTE

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.

An encapsulated request.

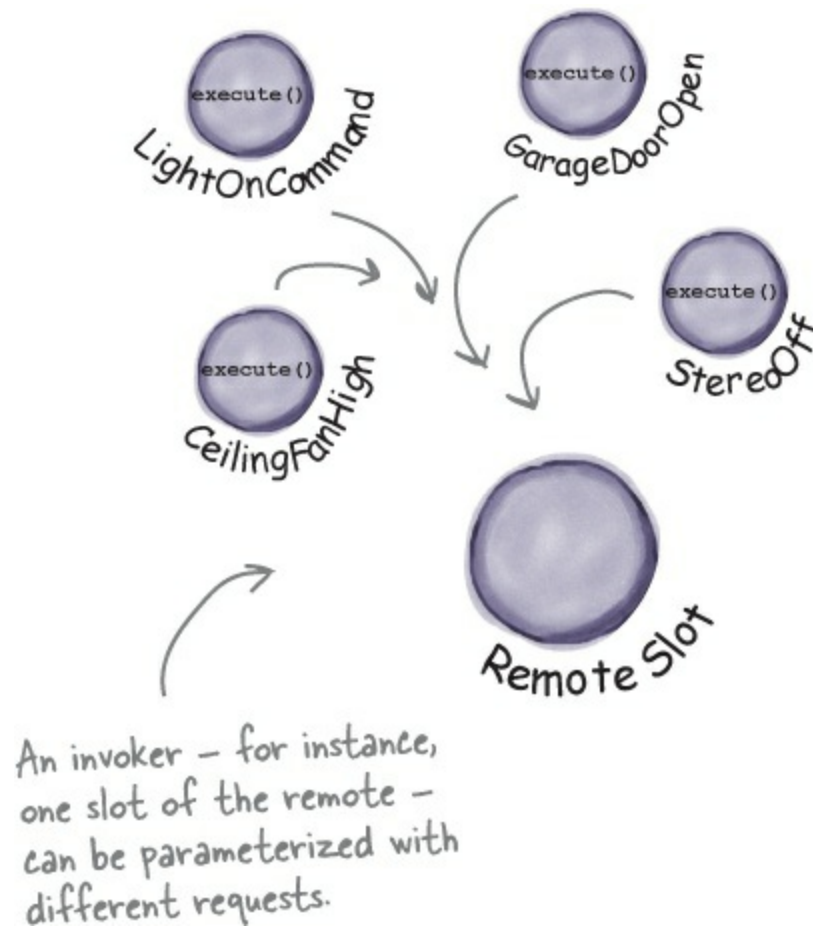


Let's step through this. We know that a command object *encapsulates a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, `execute()`. When called, `execute()` causes the actions 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.

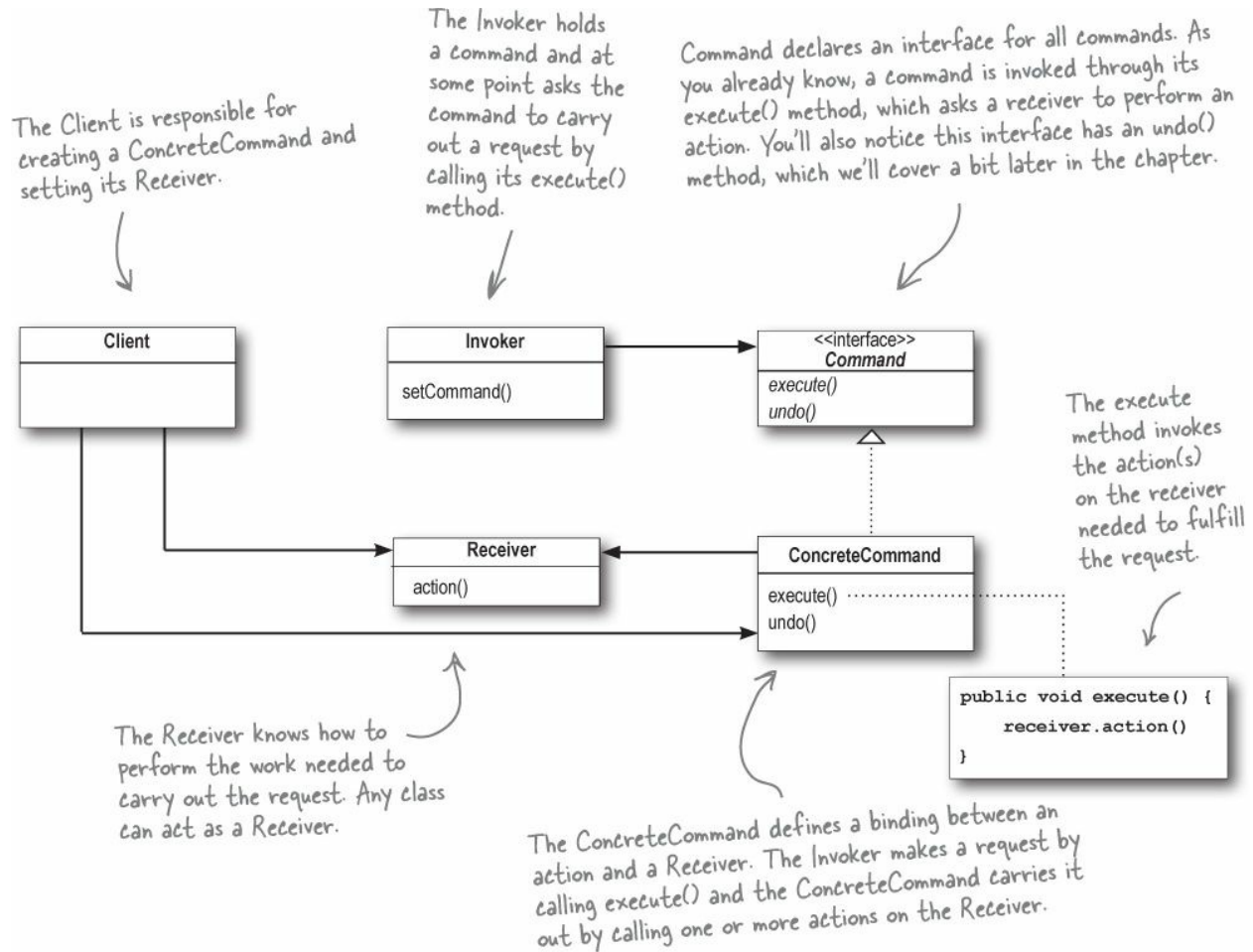
We've also seen a couple examples of *parameterizing an object* with a command. Back at the diner, the Waitress was parameterized with multiple

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.



The Command Pattern defined: the class diagram



BRAIN POWER

How does the design of the Command Pattern support the decoupling of the invoker of a request and the receiver of the request?

Okay, I think I've got a good feel for the Command Pattern now. Great tip, Joe, I think we are going to look like superstars after finishing off the Remote Control API.



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!

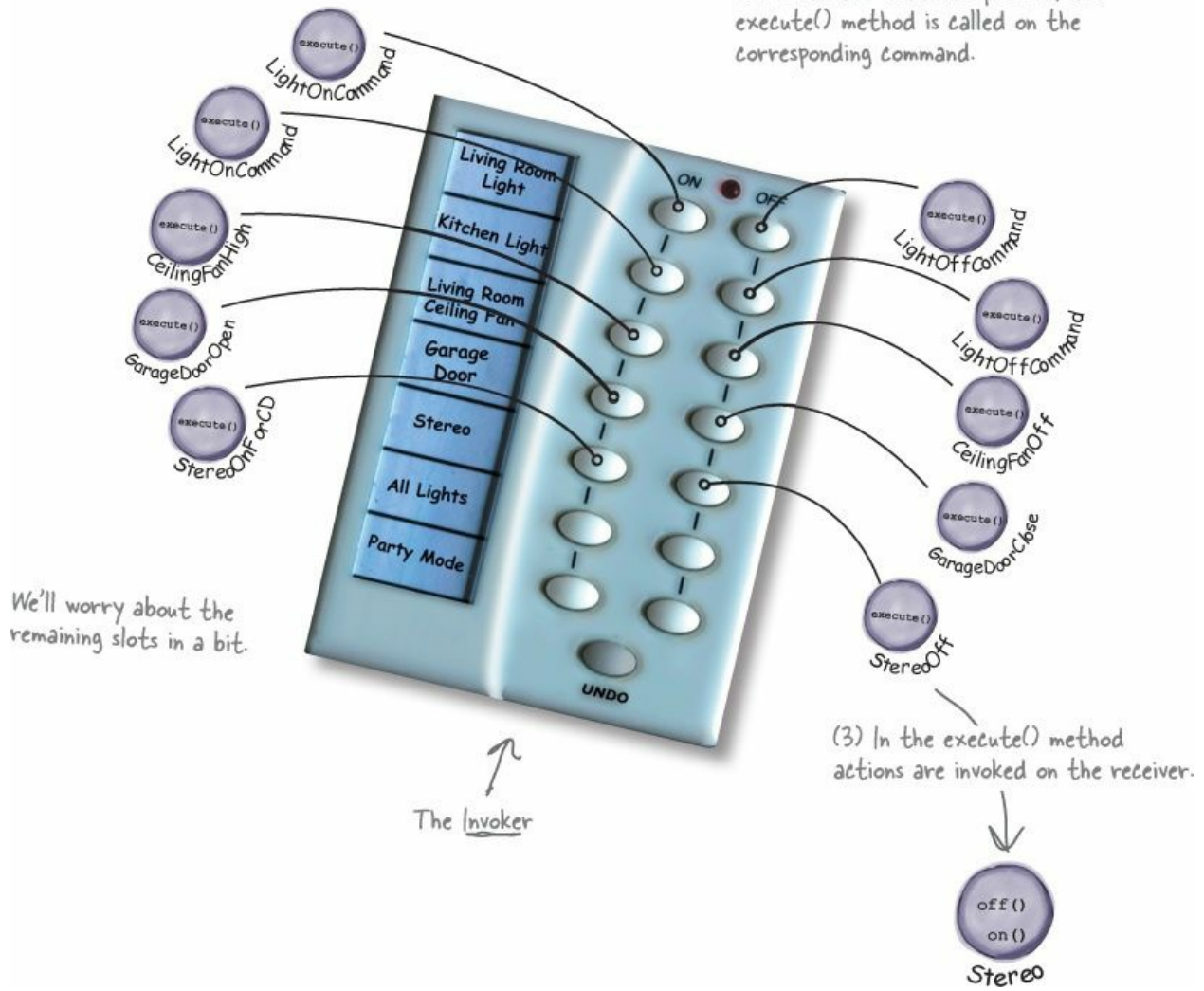
Sue: Sounds good. Let's give it a shot...

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.

(2) When the button is pressed, the execute() method is called on the corresponding command.



Implementing the Remote Control

```

public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append("\n----- Remote Control -----\n");
        for (int i = 0; i < onCommands.length; i++) {
            stringBuffer.append("[slot " + i + " ] " + onCommands[i].getClass().getName()
                + "      " + offCommands[i].getClass().getName() + "\n");
        }
        return stringBuffer.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 overridden 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

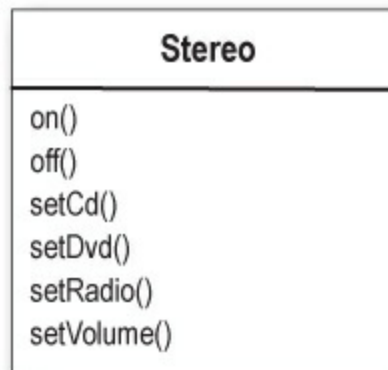
Well, we've already gotten our feet wet implementing the LightOnCommand for the SimpleRemoteControl. We can plug that same code in here and everything works beautifully. Off commands are no different; in fact, the

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 StereoOnWithCDCommand...



```

public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}

```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 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");
```

} Create all the devices in their proper locations.

```
        LightOnCommand livingRoomLightOn =  
            new LightOnCommand(livingRoomLight);  
        LightOffCommand livingRoomLightOff =  
            new LightOffCommand(livingRoomLight);  
        LightOnCommand kitchenLightOn =  
            new LightOnCommand(kitchenLight);  
        LightOffCommand kitchenLightOff =  
            new LightOffCommand(kitchenLight);
```

} Create all the Light Command objects.

```
        CeilingFanOnCommand ceilingFanOn =  
            new CeilingFanOnCommand(ceilingFan);  
        CeilingFanOffCommand ceilingFanOff =  
            new CeilingFanOffCommand(ceilingFan);
```

} Create the On and Off for the ceiling fan.

```
        GarageDoorUpCommand garageDoorUp =  
            new GarageDoorUpCommand(garageDoor);  
        GarageDoorDownCommand garageDoorDown =  
            new GarageDoorDownCommand(garageDoor);
```

} Create the Up and Down commands for the Garage.

```
        StereoOnWithCDCommand stereoOnWithCD =  
            new StereoOnWithCDCommand(stereo);  
        StereoOffCommand stereoOff =  
            new StereoOffCommand(stereo);
```

} 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);
```

} Now that we've got all our commands, we can load them into the remote slots.

```
        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);
```

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."



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.

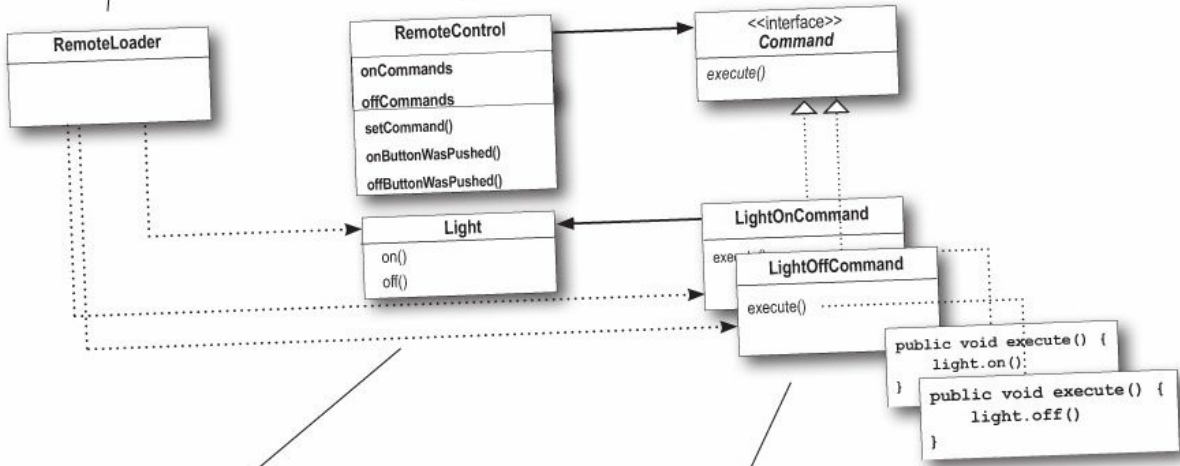
We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we 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:

The RemoteLoader creates a number of Command objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.

The RemoteControl manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home-automation work.

All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.



The Vendor Classes are used to perform the actual home-automation work of controlling devices. Here, we are using the Light class as an example.

Using the Command Interface, we implement each action that can be invoked by pressing a button on the remote with a simple Command object. The Command object holds a reference to an object that is an instance of a Vendor Class and implements an execute method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

```
public void execute() {
    light.on();
}
public void execute() {
    light.off();
}
```



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:

- ① 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");
        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.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}

```

← Create a Light, and our new undo() enabled Light On and Off Commands.

↻ Add the light Commands to the remote in slot 0.

Turn the light on, then off and then undo.

Then, turn the light off, back on and undo.

And here are the test results...

```

File Edit Window Help UndoCommandsDefyEntropy
% java RemoteLoader
Light is on
Light is off

----- 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] LightOffCommand

Light is on
Light is off
Light is 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

```

Turn the light on, then off.

Here are the Light commands.

Undo was pressed... the LightOffCommand undo() turns the light back on.

Then we turn the light off then back on.

Now undo holds the LightOffCommand, the last command invoked.

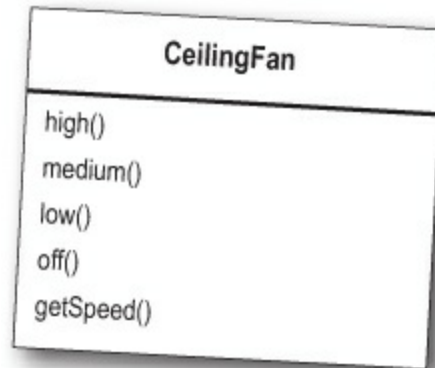
Undo was pressed, the light is back off.

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;
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

```
    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
    }
```

These methods set the speed of the ceiling fan.

```
    public void off() {
        speed = OFF;
        // code to turn fan off
    }
```

```
    public int getSpeed() {
        return speed;
    }
```

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.

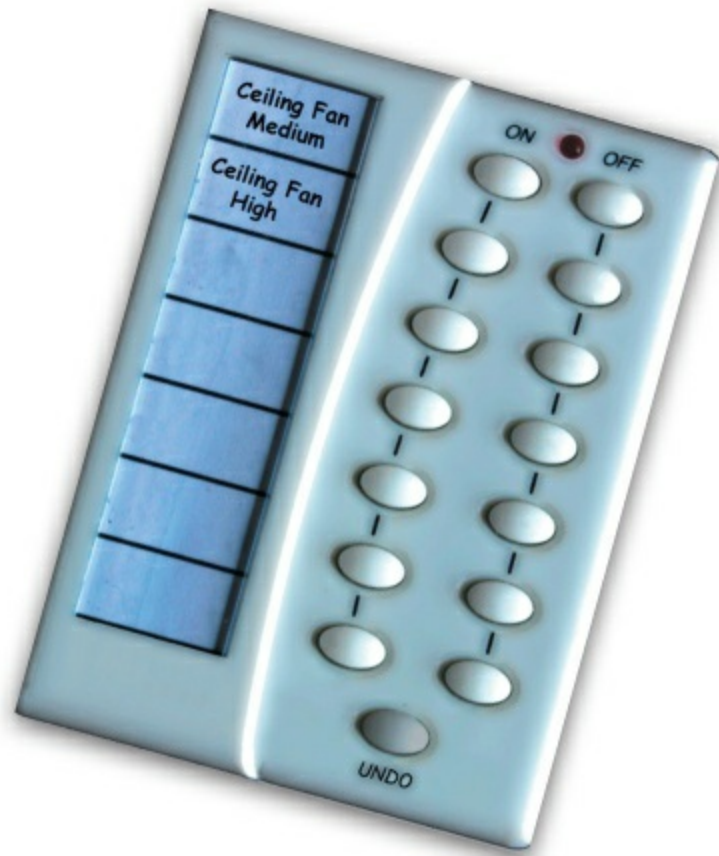
To undo, we set the speed of the fan back to its previous speed.

BRAIN POWER

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);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}

```

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
Living Room ceiling fan is 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

Living Room ceiling fan is on medium
Living Room ceiling fan is on high

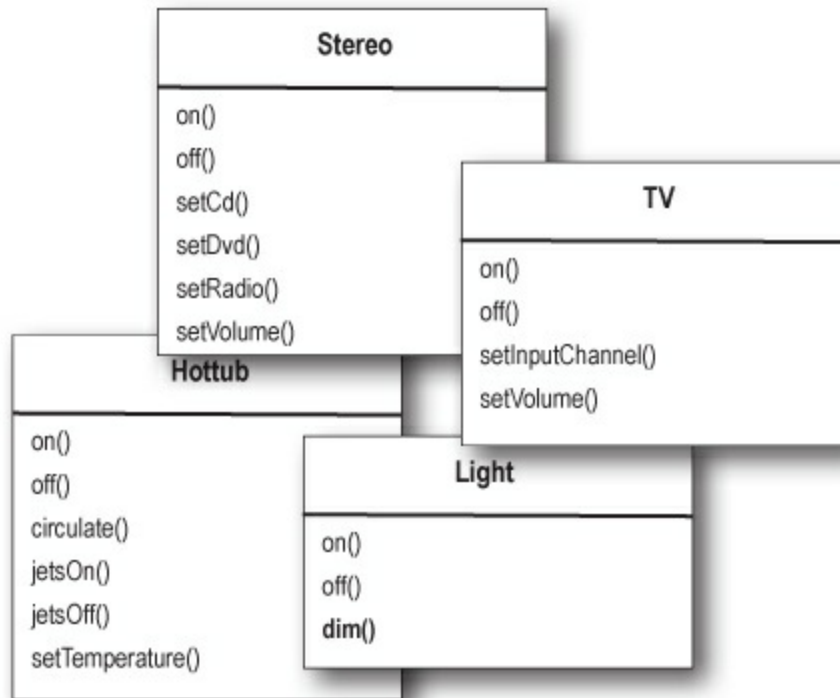
----- Remote Control -----
[slot 0] CeilingFanMediumCommand   CeilingFanOffCommand
[slot 1] CeilingFanHighCommand     CeilingFanOffCommand
[slot 2] NoCommand                 NoCommand
[slot 3] NoCommand                 NoCommand
[slot 4] NoCommand                 NoCommand
[slot 5] NoCommand                 NoCommand
[slot 6] NoCommand                 NoCommand
[undo] CeilingFanHighCommand

Living Room ceiling fan is on medium
%

Turn the ceiling fan on medium, then turn it off.
Here are the commands in the remote control...
...and undo has the last command executed, the CeilingFanOffCommand, with the previous speed of medium.
Undo the last command, and it goes back to medium.
Now, turn it on high.
Now, high is the last command executed.
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?



Hmm, our remote control would need a button for each device, I don't think we can do this.



Hold on, Sue, don't be so sure. I think we can do this without changing the remote at all!



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:

- ① First we create the set of commands we want to go into the macro:

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();
```

Create all the devices: a light, tv, stereo, and hot tub.

```
LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Now create all the On commands to control them.

SHARPEN YOUR PENCIL

We will also need commands for the off buttons. Write the code to create those here:

- ② 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.

③ 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.

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.

EXERCISE

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() {
        
    }
}

```

THERE ARE NO DUMB QUESTIONS

Q: Q: Do I always need a receiver? Why can't the command object implement the details of the execute() method?

A: 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: 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?

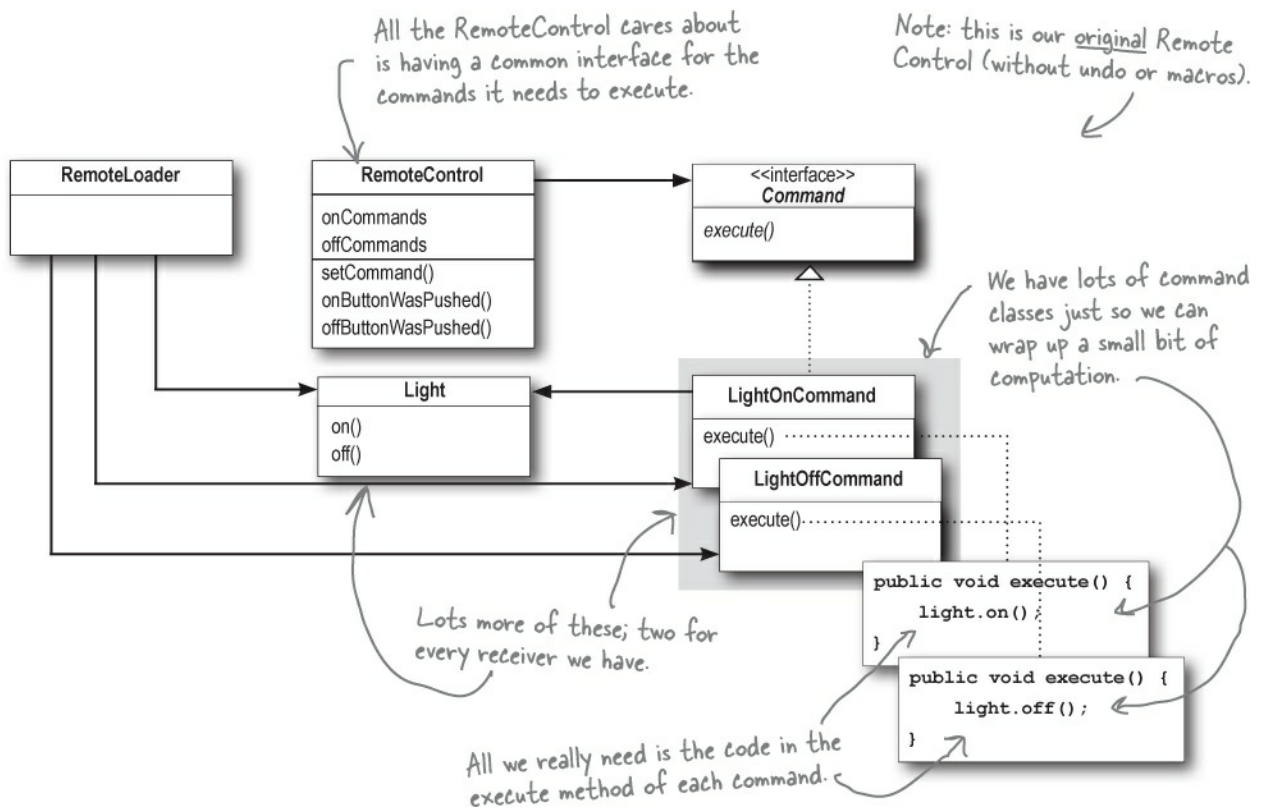
A: A: Great question. It’s pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

Q: 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: 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.

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.

NOTE

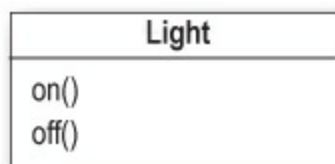
If you aren't yet familiar with lambda expressions (they were added in Java 8) they can take some getting used to. You should be able to follow along over the next few pages, but consult a Java reference to get up to speed on the syntax and semantics if you need to.

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.

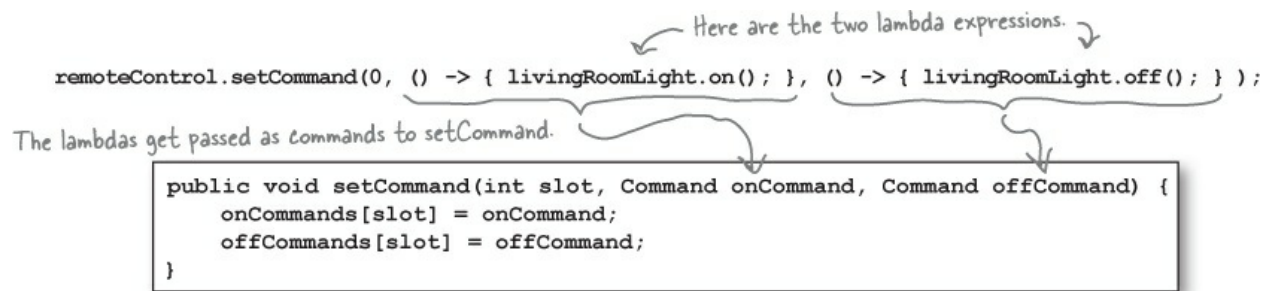
```
Light livingRoomLight = new Light("Living Room");
```



Step 2: Set the remote control's commands using lambda expressions

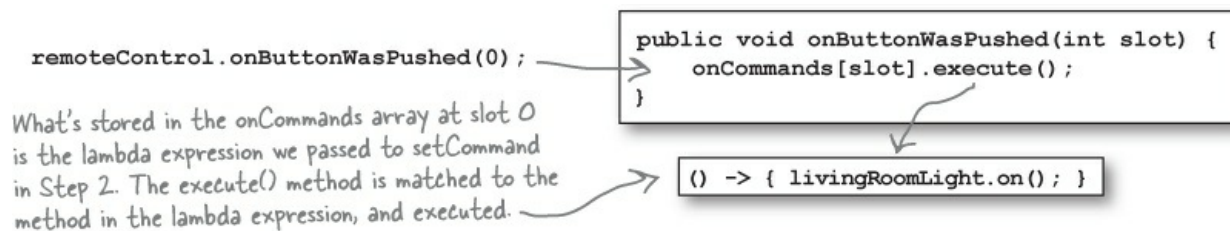
This is where the magic happens. Now, instead of creating

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:



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.



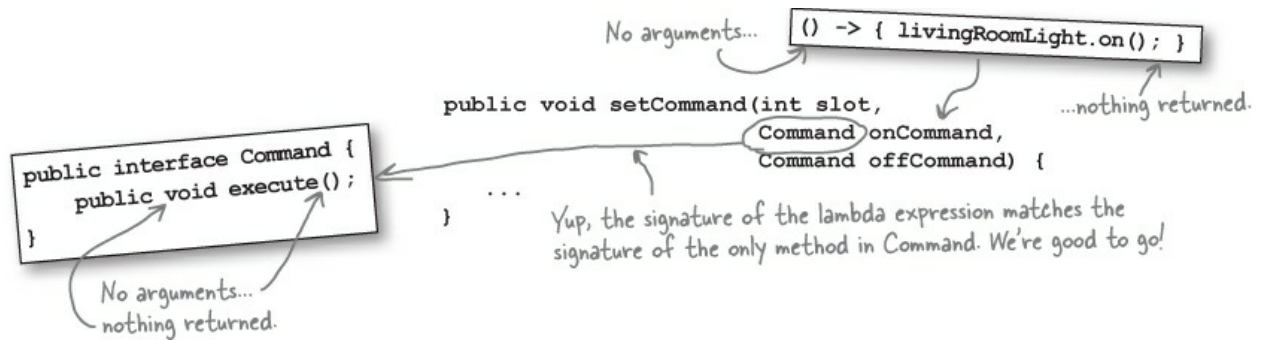


Are you trying to pull another fast one? The lambda expression we're passing into the `setCommand` method doesn't even have an `execute` method. So how does the method in the lambda ever get called?

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.

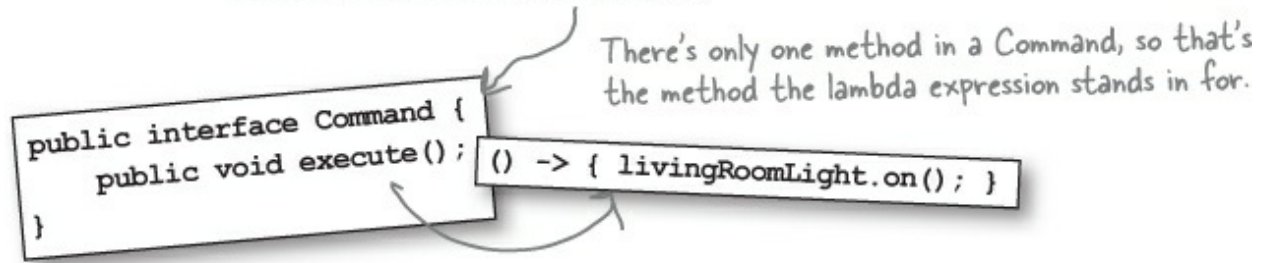
We pass the lambda expression into the Command parameter of the `setCommand()` method:



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:

```
onCommands[slot].execute();
```



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 `stereoOnWithCDCommand` 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 `stereoOnWithCDCommand` 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 `stereoOnWithCDCommand`'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:

Wow, we got that implementation from 22 classes down to 9. That's a lot easier to manage.



```
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$$Lambda$1/168423058 RemoteLoader$$Lambda$2/1247233941
[slot 1] RemoteLoader$$Lambda$3/258952499 RemoteLoader$$Lambda$4/603742814
[slot 2] RemoteLoader$$Lambda$5/1325547227 RemoteLoader$$Lambda$6/980546781
[slot 3] RemoteLoader$$Lambda$9/1706377736 RemoteLoader$$Lambda$10/1804094807
[slot 4] RemoteControl$$Lambda$1/713338599 RemoteControl$$Lambda$2/1247233941
[slot 5] RemoteControl$$Lambda$1/713338599 RemoteControl$$Lambda$2/1247233941
[slot 6] RemoteControl$$Lambda$1/713338599 RemoteControl$$Lambda$2/1247233941

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
%
```

On slots Off slots

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: Q: Can a lambda expression have parameters or return a value? Or does it always have to be a void, no-argument method?

A: 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: 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: 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: 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: 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: Q: Why do the names of on and off slots look so weird when we display the RemoteControl?

A: **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.

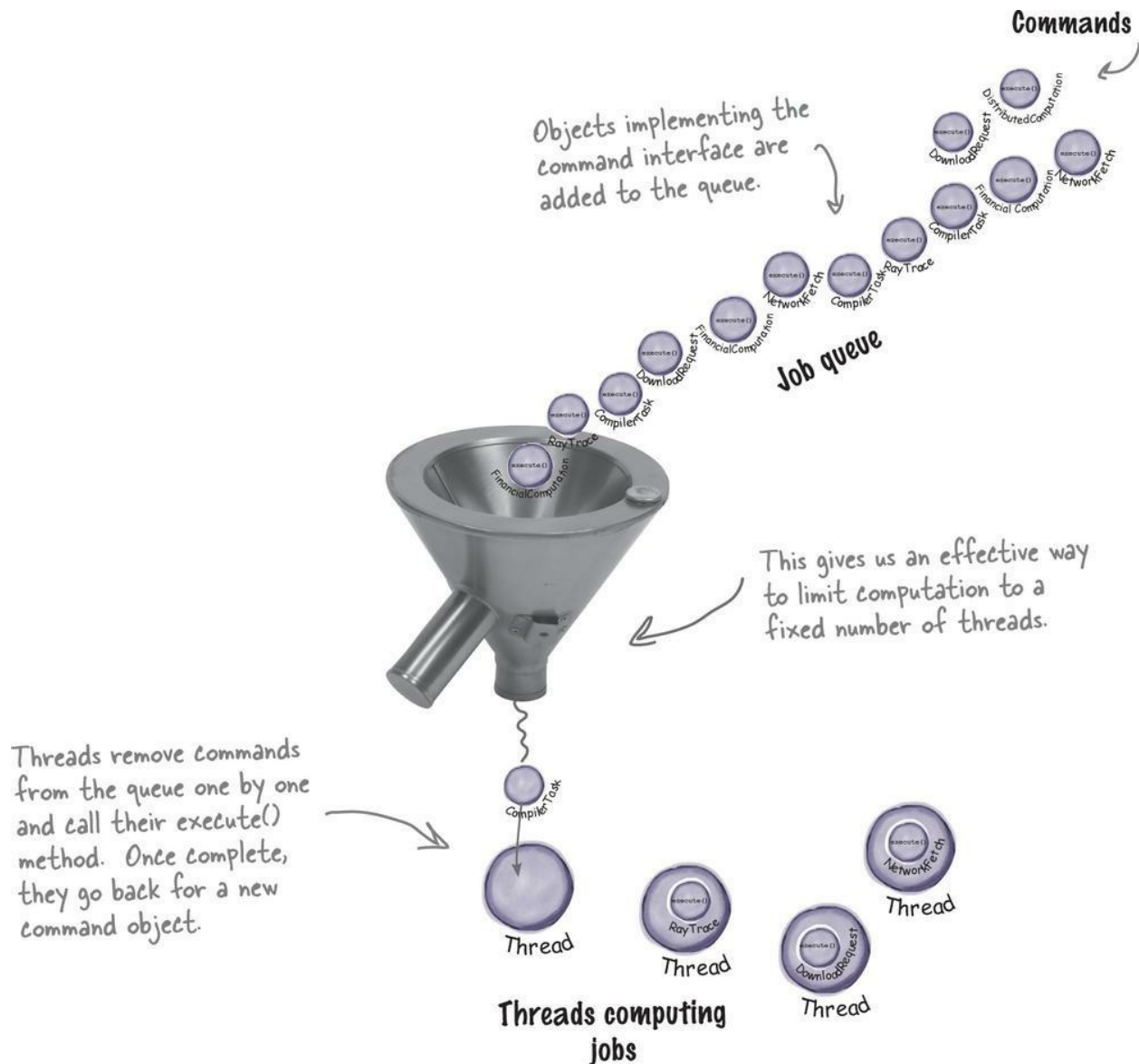
As you can see from the output, lambda expressions don't have nice class names. That's because their names are assigned internally by the Java runtime and Java has no idea what these lambda expressions mean; to Java, they're just function objects that happen to match a method in an interface.

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.

BRAIN POWER

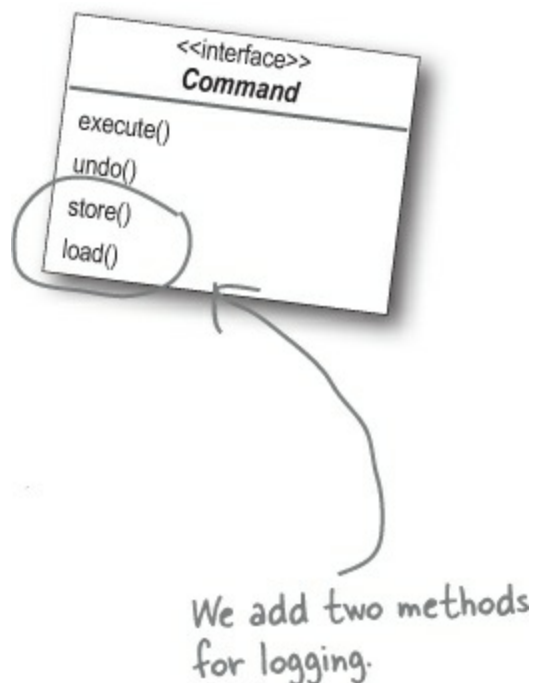
How might a web server make use of such a queue? What other applications can you think of?

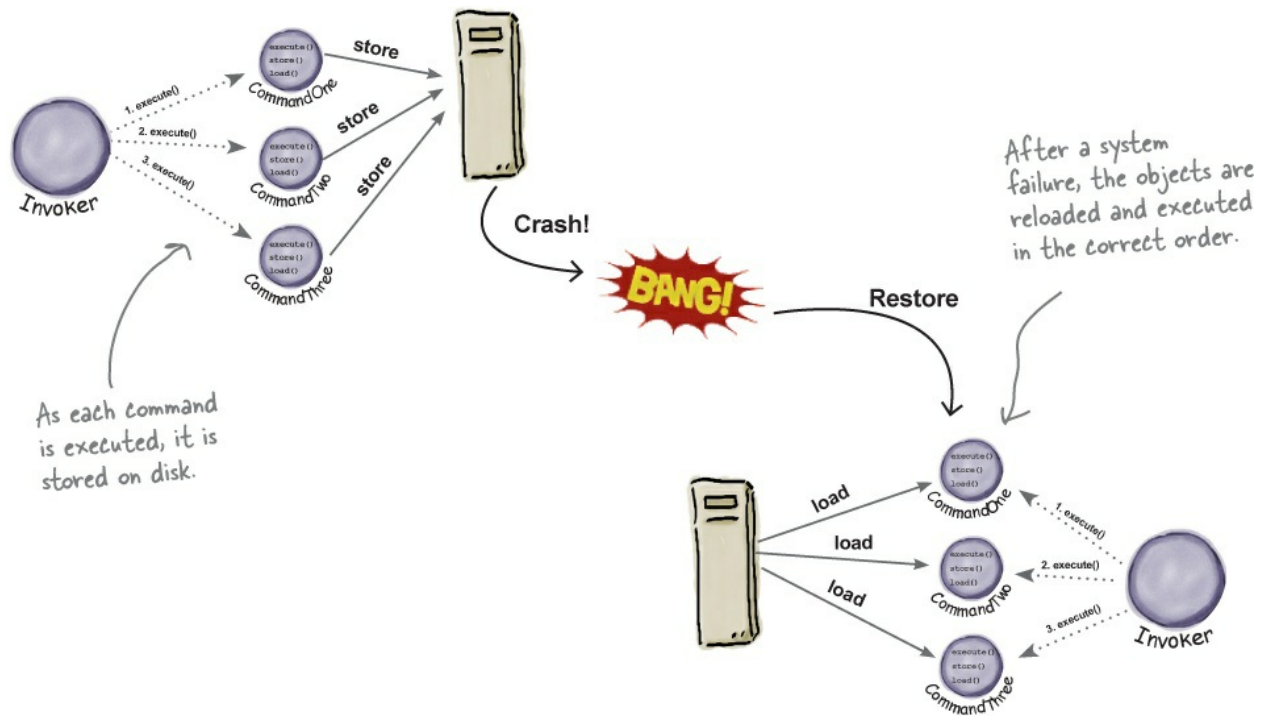
More uses of the Command Pattern: logging requests

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.

OO Basics

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.

Abstraction

Encapsulation

Polymorphism

Inheritance

When you need to decouple an object making requests from the objects that know how to perform the requests, use the Command Pattern.

OO Patterns

S

ev

in

va

va

va

va

va

va

va

va

va

va

va

va

va

va

va

va

D

A

F

S

C

D

A

S

C

D

A

S

C

D

A

S

C

D

A

Singleton - Ensure a class only has one of

Command - Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



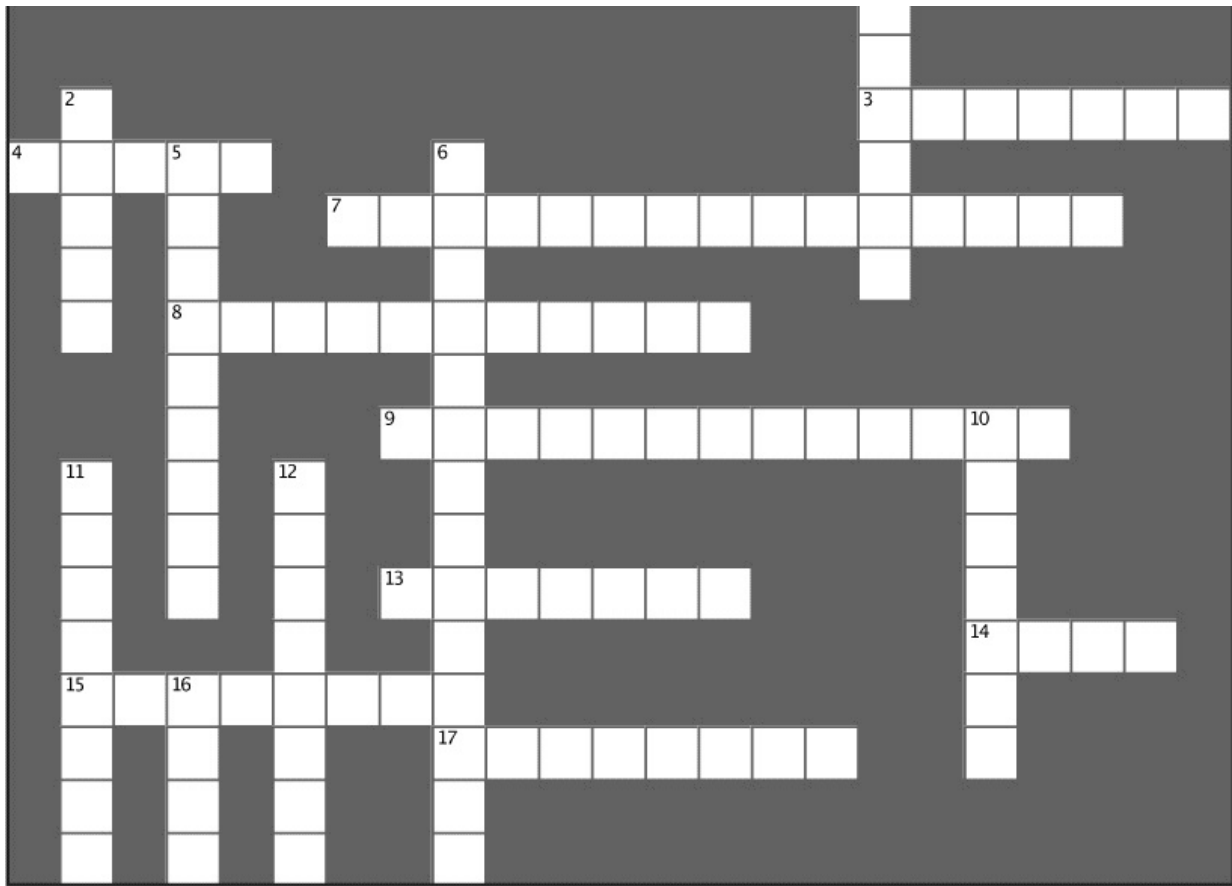
BULLET POINTS

- 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.

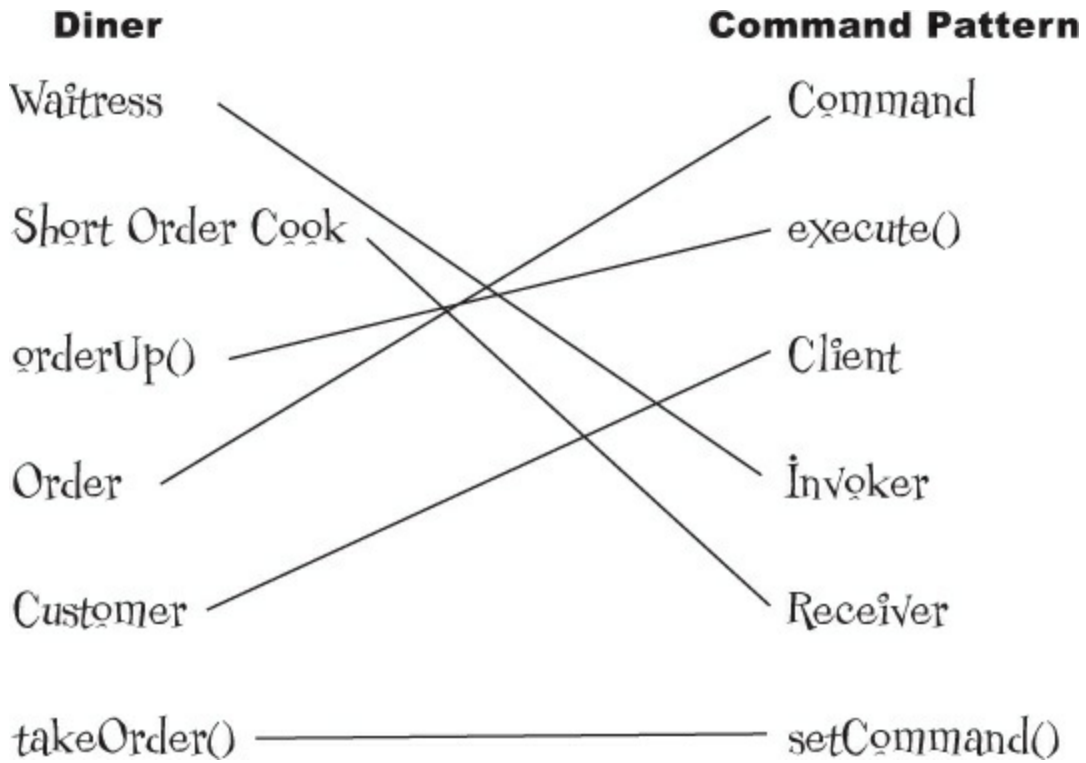


Across	Down
3. The Waitress was one.	1. Role of customer in the Command Pattern.
4. A command _____ a set of actions and a receiver.	2. Our first command object controlled this.
7. Dr. Seuss diner food.	5. Invoker and receiver are _____.
8. Our favorite city.	6. Company that got us word-of-mouth business.
9. Act as the receivers in the remote control.	10. All commands provide this.
13. Object that knows the actions and the receiver.	11. The Cook and this person were definitely decoupled.
14. Another thing Command can do.	12. Carries out a request.
15. Object that knows how to get things done.	16. Waitress didn't do this.
17. A command encapsulates 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;
    }

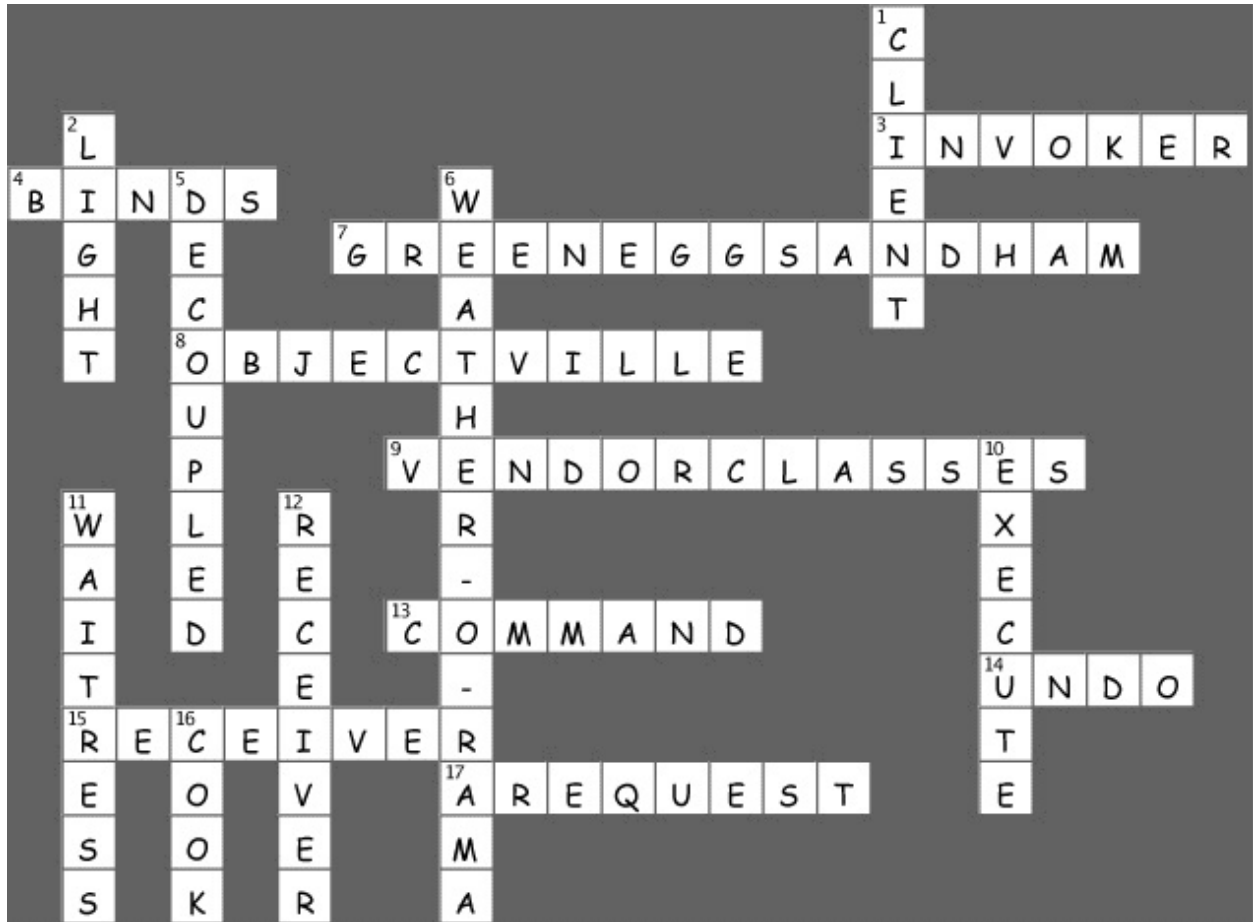
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    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);
```



Chapter 7. The Adapter and Facade Patterns: Being Adaptive



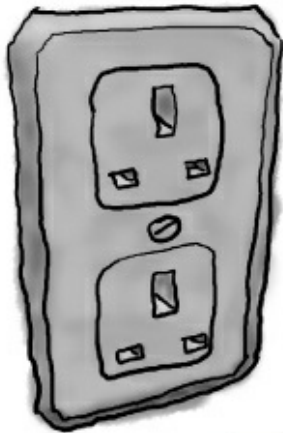
In this chapter we're going to attempt such impossible feats as putting a square peg in a round hole. Sound impossible? Not when we have Design Patterns. Remember the Decorator Pattern? We **wrapped objects** to give them new responsibilities. Now we're going to wrap some objects with a different purpose: 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...

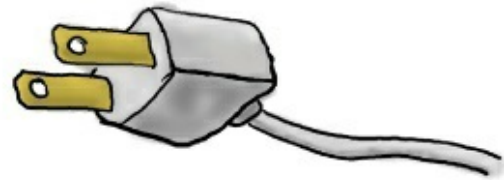
British Wall Outlet



AC Power Adapter



Standard AC Plug



The British wall outlet exposes one interface for getting power.

The adapter converts one interface into another.

The US laptop expects another interface.

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.

NOTE

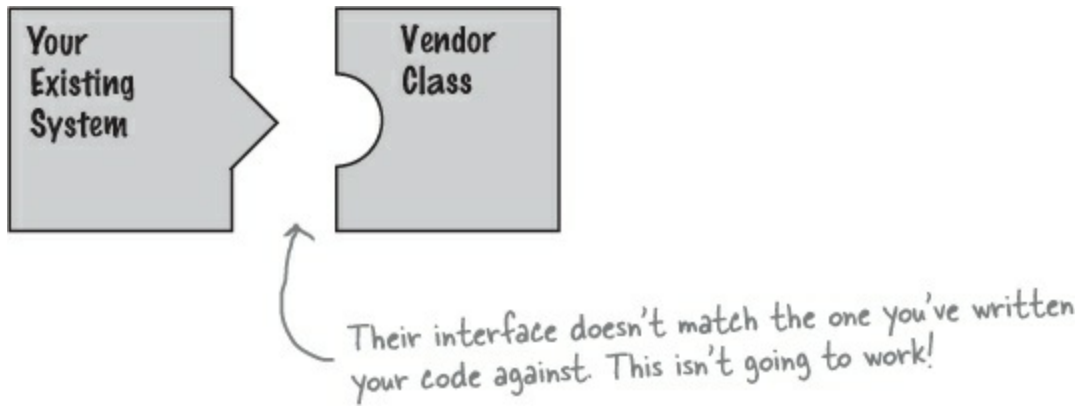
How many other real-world adapters can you think of?

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

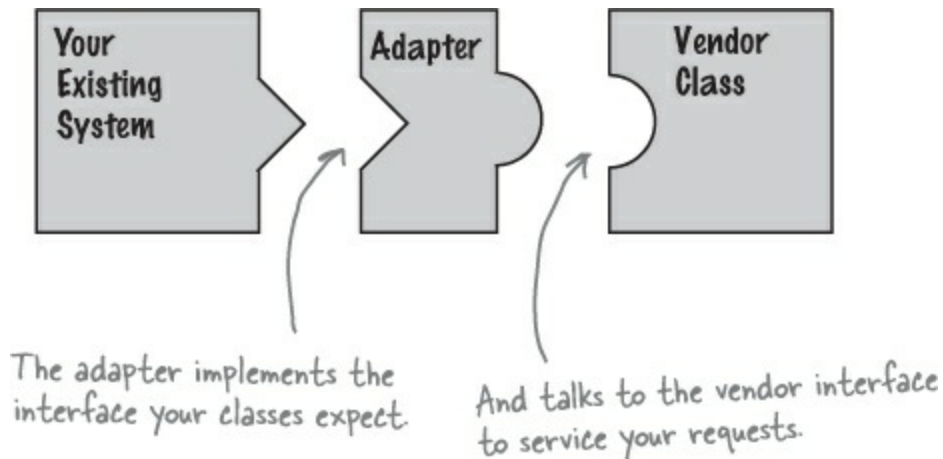
Okay, that's the real world; what about object-oriented adapters? Well, our OO adapters play the same role as their real-world counterparts: they take an interface and adapt it to one that a client is expecting.

Object-oriented adapters

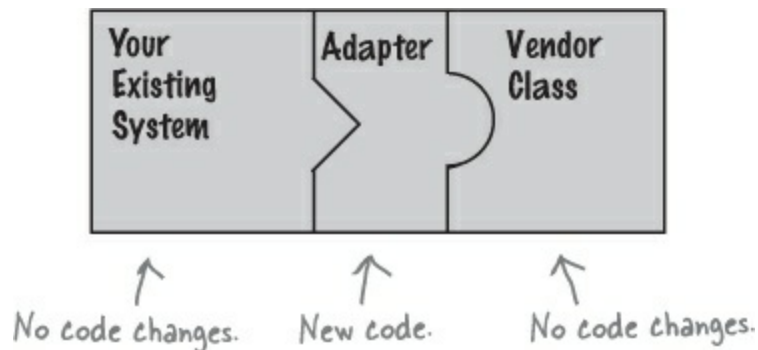
Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



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.



NOTE

Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class?

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

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



```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Here's a subclass of Duck, the MallardDuck.


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

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

Now it's time to meet the newest fowl on the block:

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Now, let's say you're short on Duck objects and you'd like to use some Turkey objects in their place. Obviously we can't use the turkeys outright because they have a different interface.

So, let's write an Adapter:

CODE UP CLOSE

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

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

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

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

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

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

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

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

Test drive the adapter

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

```

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

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

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

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

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

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

```

Let's create a Duck...

...and a Turkey.

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

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

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

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

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

Test run

```

File Edit Window Help Don'tForgetToDuck
%java 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
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance

```

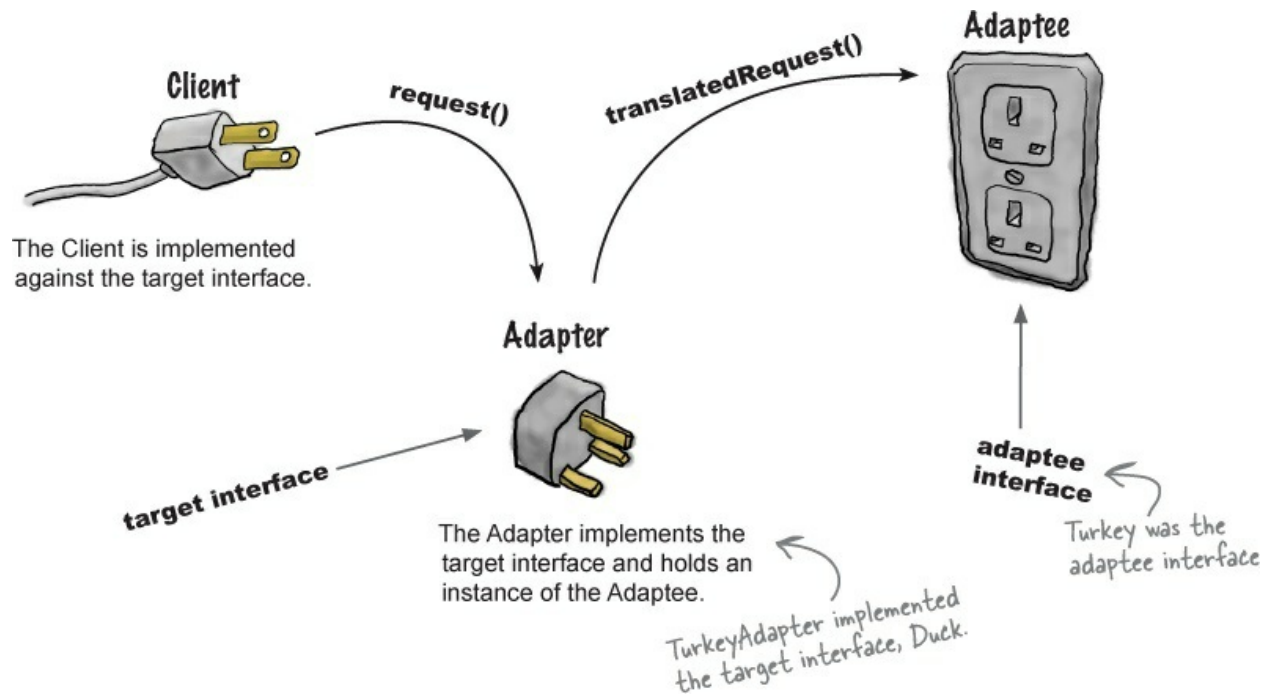
The Turkey gobbles and flies a short distance.

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

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

The Adapter Pattern explained

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



Here's how the Client uses the Adapter

- ① The client makes a request to the adapter by calling a method on it using the target interface.
- ② The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
- ③ The client receives the results of the call and never knows there is an adapter doing the translation.

NOTE

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

A: 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: 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: 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

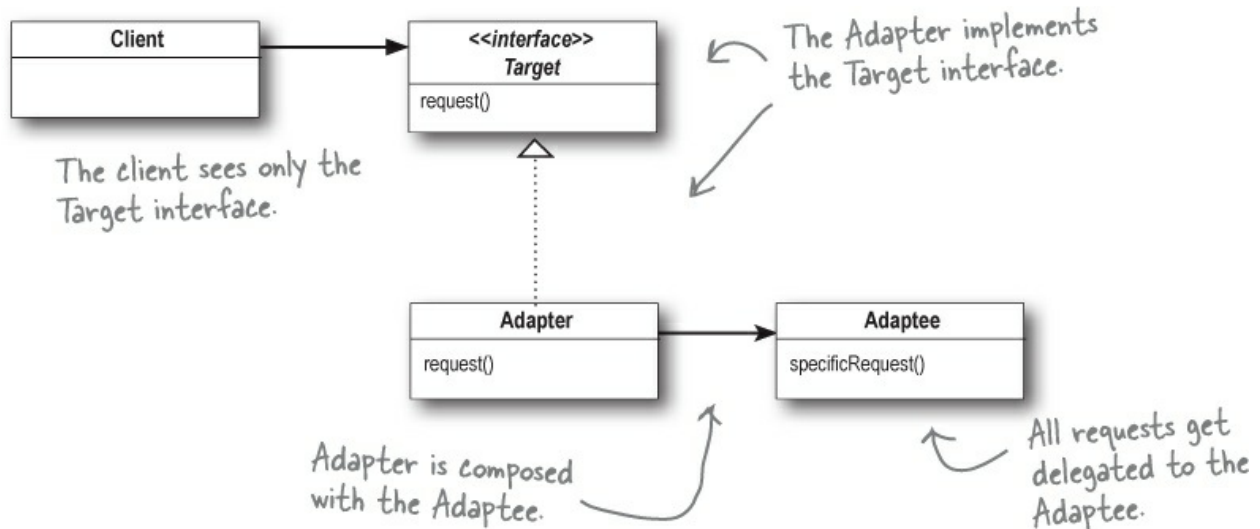
Enough ducks, turkeys, and AC power adapters; let’s get real and look at the official definition of the Adapter Pattern:

NOTE

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:



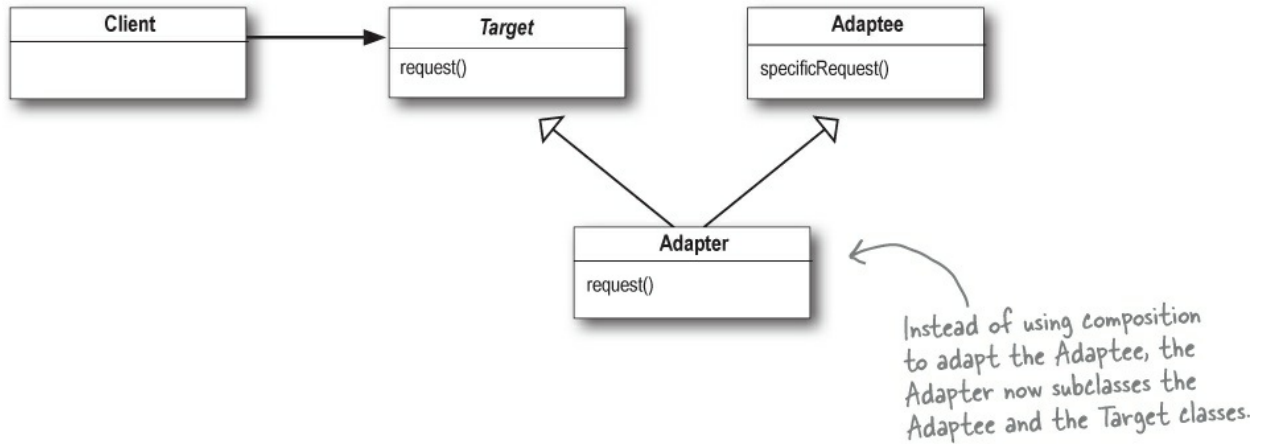
The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

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.

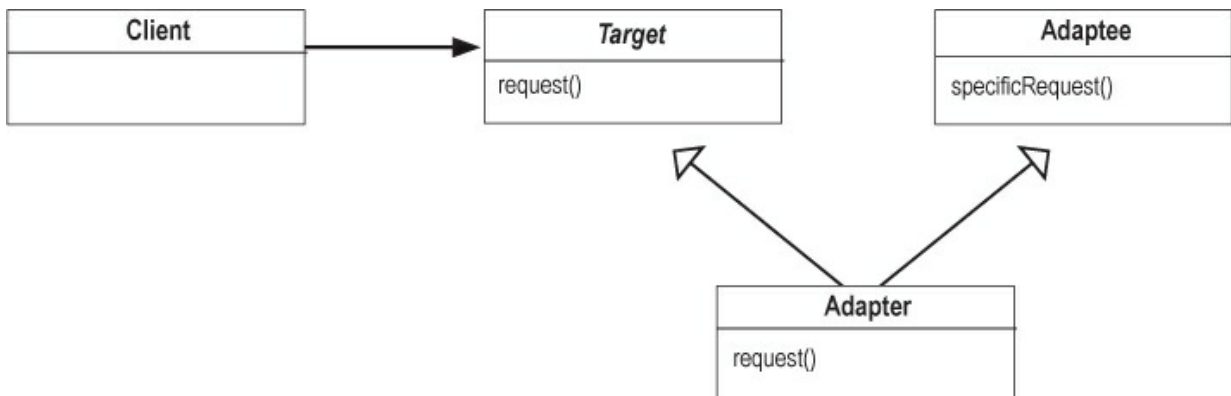
BRAIN POWER

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

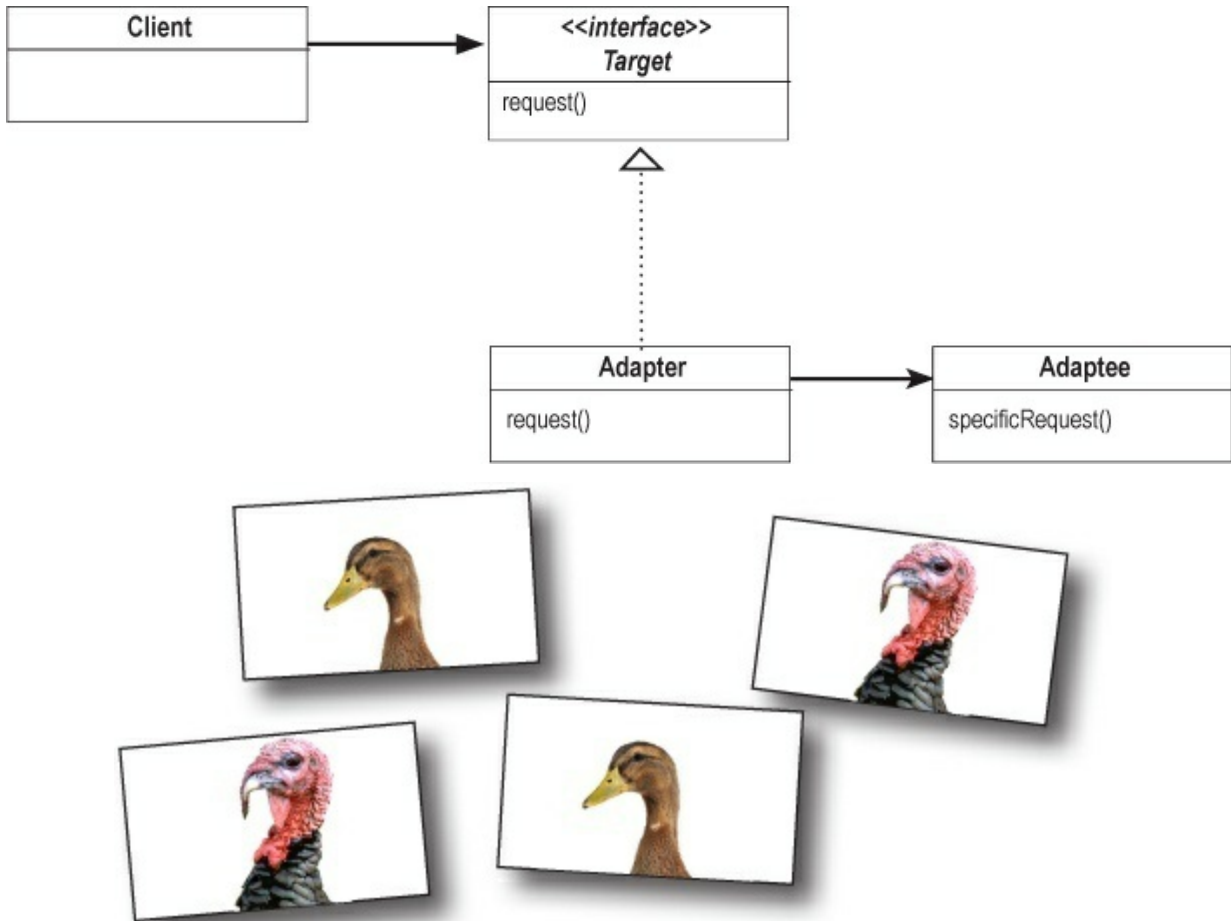
DUCK MAGNETS

Your job is to take the duck and turkey magnets and drag them over the part of the diagram that describes the role played by that bird, in our earlier example. (Try not to flip back through the pages.) Then add your own annotations to describe how it works.

Class Adapter



Object Adapter



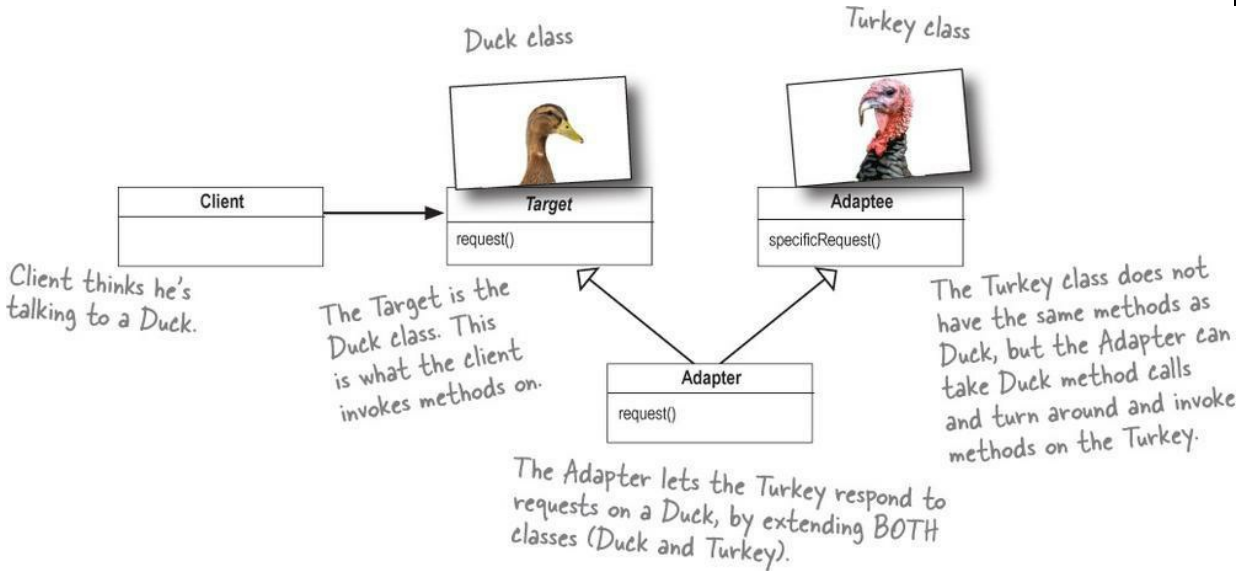
Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.

DUCK MAGNETS ANSWER

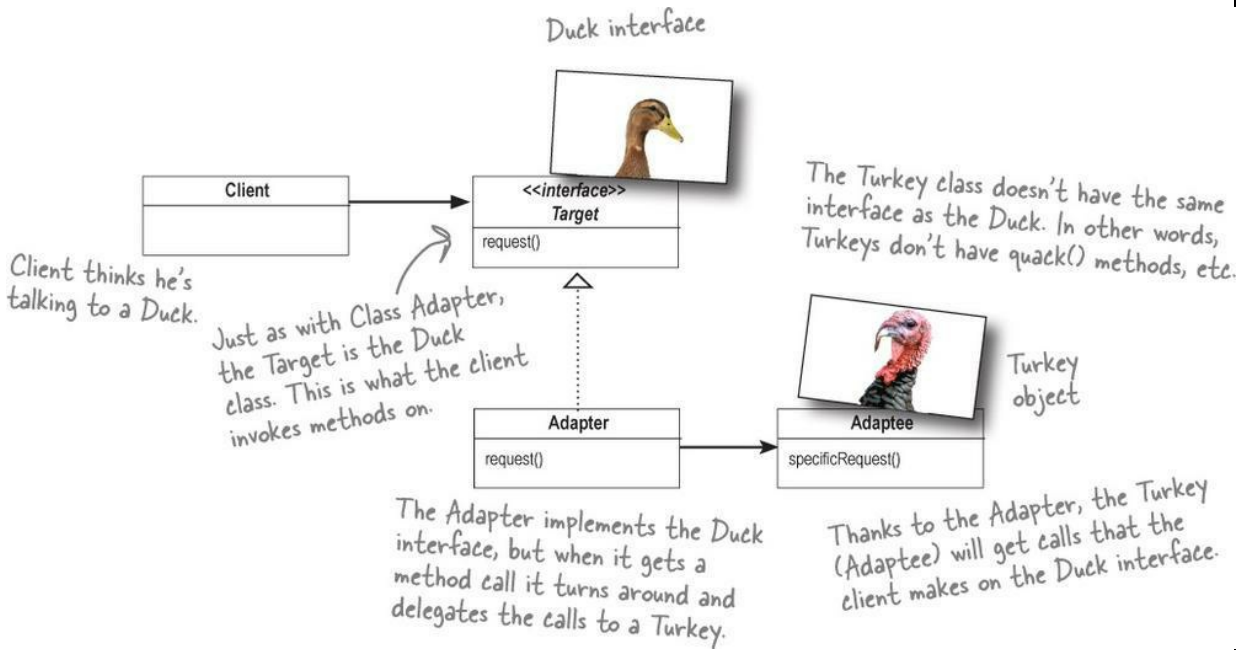
NOTE

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

Class Adapter



Object Adapter



FIRESIDE CHATS

Tonight's talk: **The Object Adapter and Class Adapter meet face to face.**

Object Adapter:	Class Adapter:
Because I use composition I've got a leg up. I can not only adapt an adaptee class, but any of its subclasses.	

	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.
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.	
	Flexible maybe, but efficient? No. Using a class adapter there is just one of me, not an adapter and an adaptee.
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 <i>and</i> all its subclasses.	
	Yeah, but what if a subclass of adaptee adds some new behavior. Then what?
Hey, come on, cut me a break, I just need to compose with the subclass to make that work.	
	Sounds messy...
You wanna see messy? Look in the mirror!	

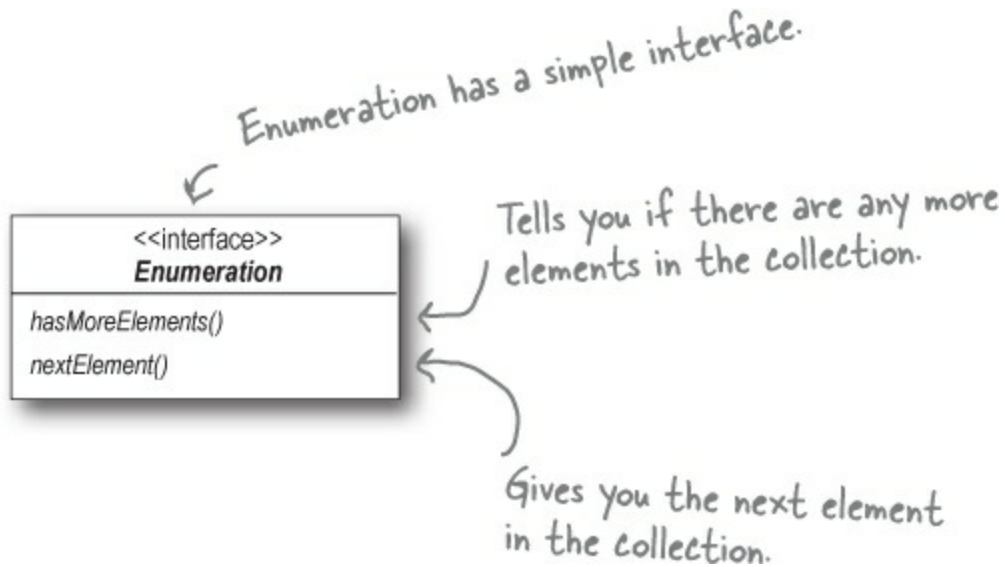
Real-world adapters

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

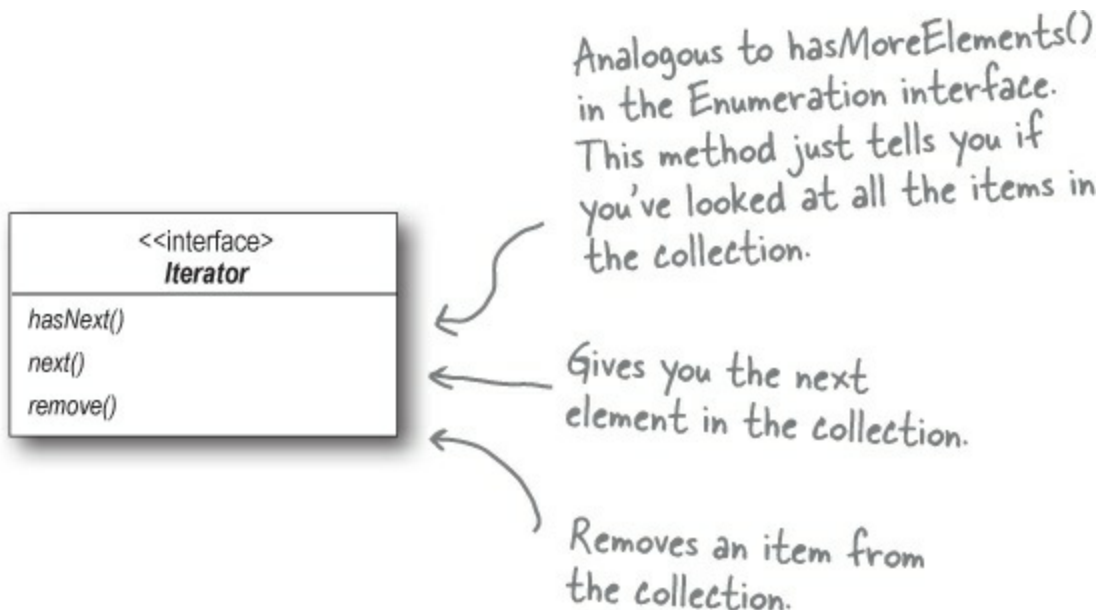
If you've been around Java for a while you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, `elements()`, which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without

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.

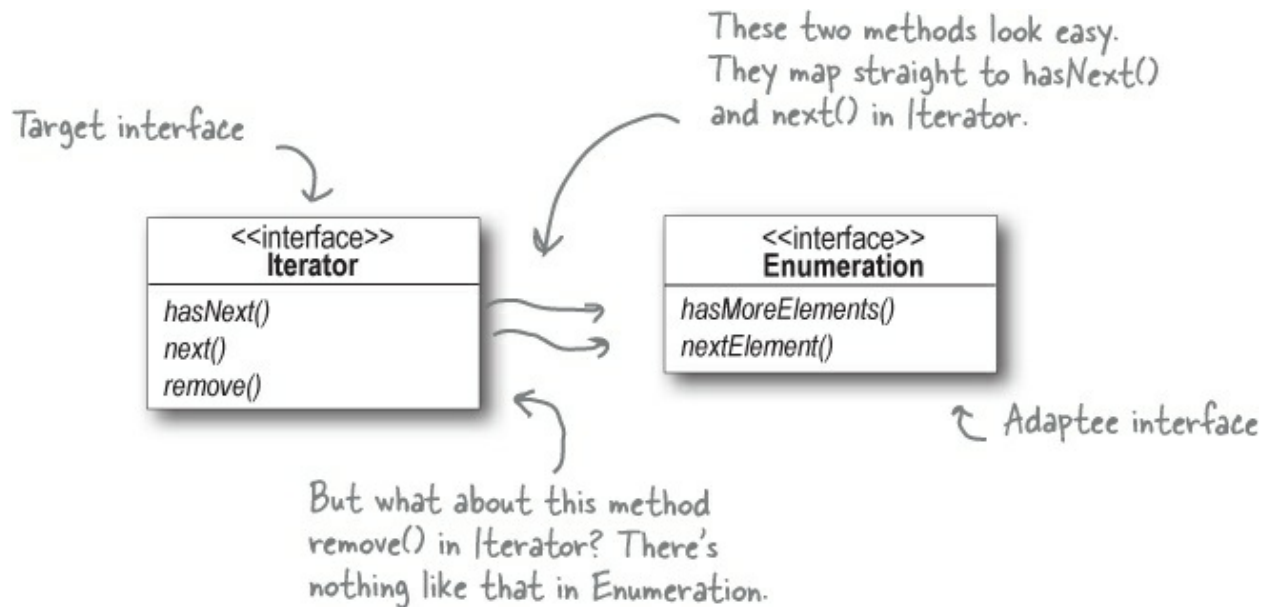


And today...

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.

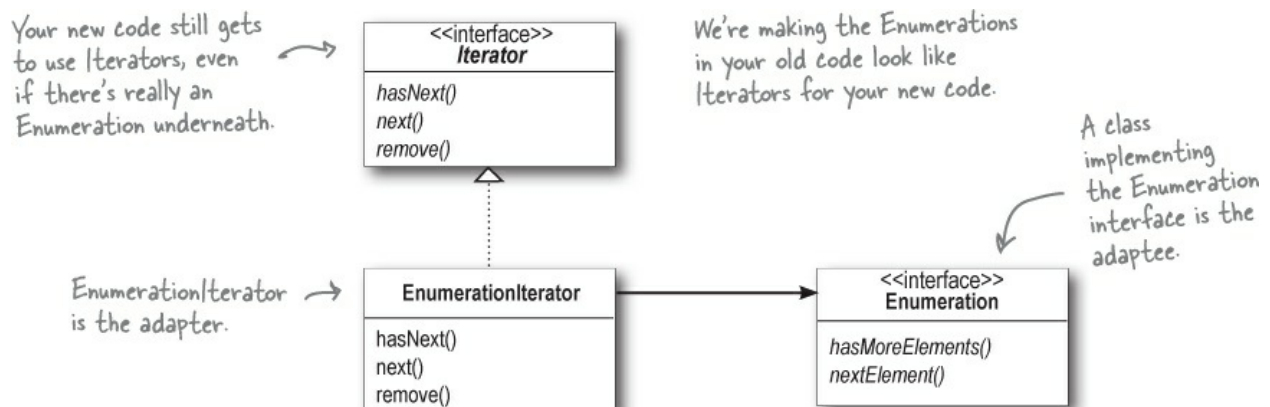
Adapting an Enumeration to an Iterator

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.

EXERCISE

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).

BRAIN POWER

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

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

FIRESIDE CHATS

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

Decorator:	Adapter:
I'm important. My job is all about <i>responsibility</i> — you know that when a Decorator is involved there's going to be some new responsibilities or behaviors added to your design.	
	You guys want all the glory while us adapters are down in the trenches doing the dirty work: converting interfaces. Our jobs may not be glamorous, but our clients sure do appreciate us making their lives simpler.
That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.	
	Try being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying: "an uncoupled client is a happy client."
Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how	

many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.	
	Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job.
	But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing <i>any</i> code; they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.
Well, us decorators do that as well, only we allow <i>new behavior</i> to be added to classes without altering existing code. I still say that adapters are just fancy decorators — I mean, just like us, you wrap an object.	
	No, no, no, not at all. We <i>always</i> convert the interface of what we wrap; you <i>never</i> do. I'd say a decorator is like an adapter; it is just that you don't change the interface!
Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap; we aren't a <i>simple pass through</i> .	
	Hey, who are you calling a simple pass through? Come on down and we'll see how long <i>you</i> last converting a few interfaces!
Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are <i>miles</i> apart in our <i>intent</i> .	
	Oh yeah, I'm with you there.

And now for something different...

There's another pattern in this chapter.

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by

wrapping the object that has an incompatible interface with an object that implements the correct one.

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

WHO DOES WHAT?

Match each pattern with its intent:

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

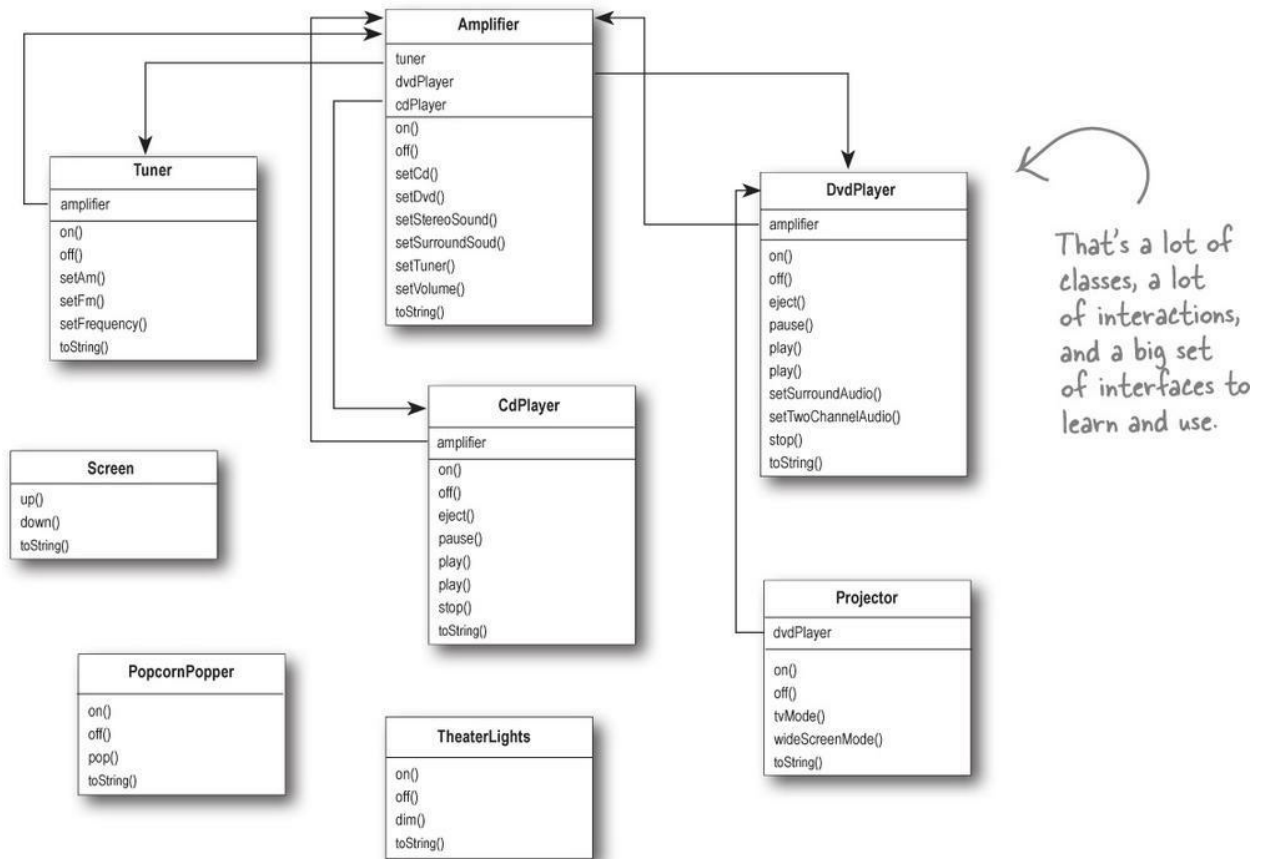
Home Sweet Home Theater

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

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



Check out all the components you've put together:



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:

- ① Turn on the popcorn popper
- ② Start the popper popping
- ③ Dim the lights
- ④ Put the screen down
- ⑤ Turn the projector on
- ⑥ Set the projector input to DVD
- ⑦ Put the projector on wide-screen mode
- ⑧ Turn the sound amplifier on
- ⑨ Set the amplifier to DVD input

- ⑩ Set the amplifier to surround sound
- ⑪ Set the amplifier volume to medium (5)
- ⑫ Turn the DVD player on
- ⑬ Start the DVD player playing



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

Six different classes involved!

```
popper.on();  
popper.pop();
```

Turn on the popcorn popper and start popping...

```
lights.dim(10);
```

Dim the lights to 10%...

```
screen.down();
```

Put the screen down...

```
projector.on();  
projector.setInput(dvd);  
projector.wideScreenMode();
```

Turn on the projector and put it in wide screen mode for the movie...

```
amp.on();  
amp.setDvd(dvd);  
amp.setSurroundSound();  
amp.setVolume(5);
```

Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...

```
dvd.on();  
dvd.play(movie);
```

Turn on the DVD player... and FINALLY, play the movie!

But there's more...

- When the movie is over, how do you turn everything off? Wouldn't you have to do all of this over again, in reverse?
- Wouldn't it be as complex to listen to a CD or the radio?
- If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure.

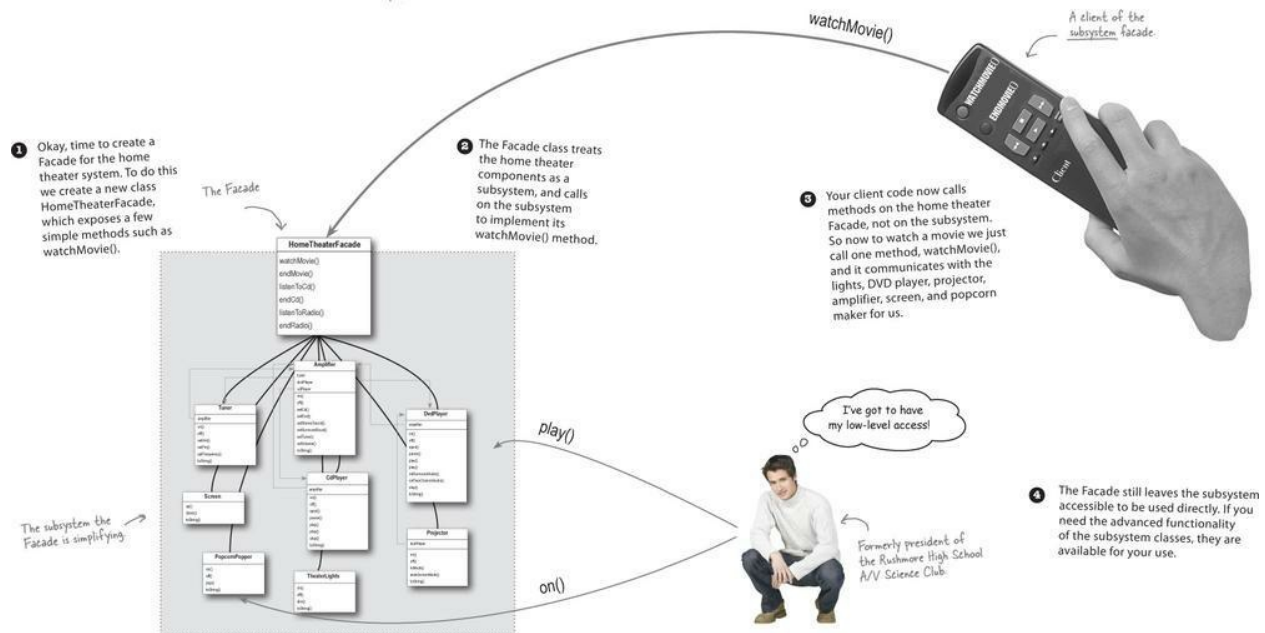
So what to do? The complexity of using your home theater is becoming apparent!

Let's see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

Lights, Camera, Facade!

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

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



THERE ARE NO DUMB QUESTIONS

Q: Q: If the facade encapsulates the subsystem classes, how does a client that needs lower-level functionality gain access to them?

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

A: 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 high movie showing).

Q: Q: Does each subsystem have only one facade?

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

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

A: 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: 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: 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;

```

Here's the composition; these are all the components of the subsystem we are going to use.

```

    public HomeTheaterFacade(Amplifier amp,
        Tuner tuner,
        DvdPlayer dvd,
        CdPlayer cd,
        Projector projector,
        Screen screen,
        TheaterLights lights,
        PopcornPopper popper) {

```

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.

```

        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

```

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);
}

```

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.

```

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}

```

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

BRAIN POWER

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

Time to watch a movie (the easy way)

It's SHOWTIME!



```

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

        HomeTheaterFacade homeTheater =
            new HomeTheaterFacade(amp, tuner, dvd, cd,
                projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}

```

Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Here's the output.
Calling the Facade's watchMovie() does all this work for us...

...and here, we're done watching the movie, so calling endMovie() turns everything off.

```

File Edit Window Help SnakesWhy dItHaveToBeSnakes?
%java HomeTheaterTestDrive
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%

```

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

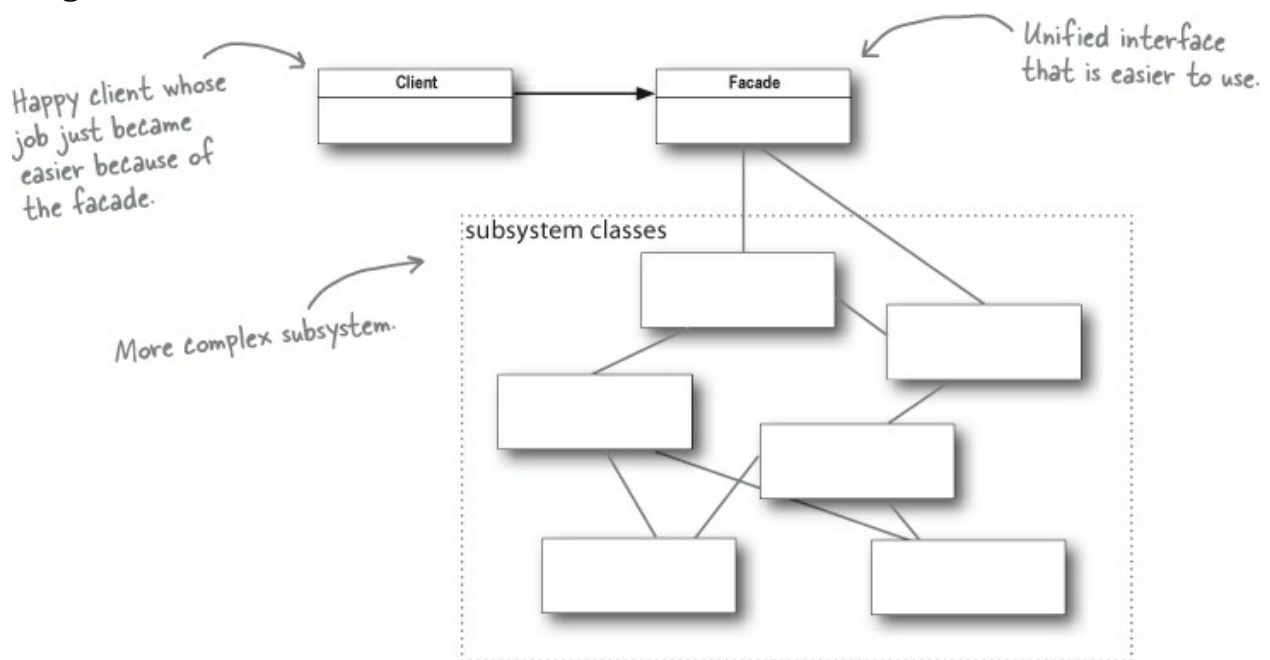
clients and subsystems, and, as you will see shortly, also helps us adhere to a new object-oriented principle.

Before we introduce that new principle, let's take a look at the official definition of the pattern:

NOTE

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:

DESIGN PRINCIPLE

Principle of Least Knowledge: talk only to your immediate friends.

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

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

BRAIN POWER

How many classes is this code coupled to?

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

How NOT to Win Friends and Influence Objects

Okay, but how do you keep from doing this? The principle provides some guidelines: take any object; now from any method in that object, the principle tells us that we should only invoke methods that belong to:

- The object itself
- Objects passed in as a parameter to the method
- Any object the method creates or instantiates

NOTE

Notice that these guidelines tell us not to call methods on objects that were returned from calling other methods!!

- Any components of the object

NOTE

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
    }
}

```

Here's a component of this class. We can call its methods.

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

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

You can call a method on a component of the object.

You can call a local method within the object.

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

THERE ARE NO DUMB QUESTIONS

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

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

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

SHARPEN YOUR PENCIL

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

```
public House {
    WeatherStation station;

    // other methods and constructor

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

public House {
    WeatherStation station;

    // other methods and constructor

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

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



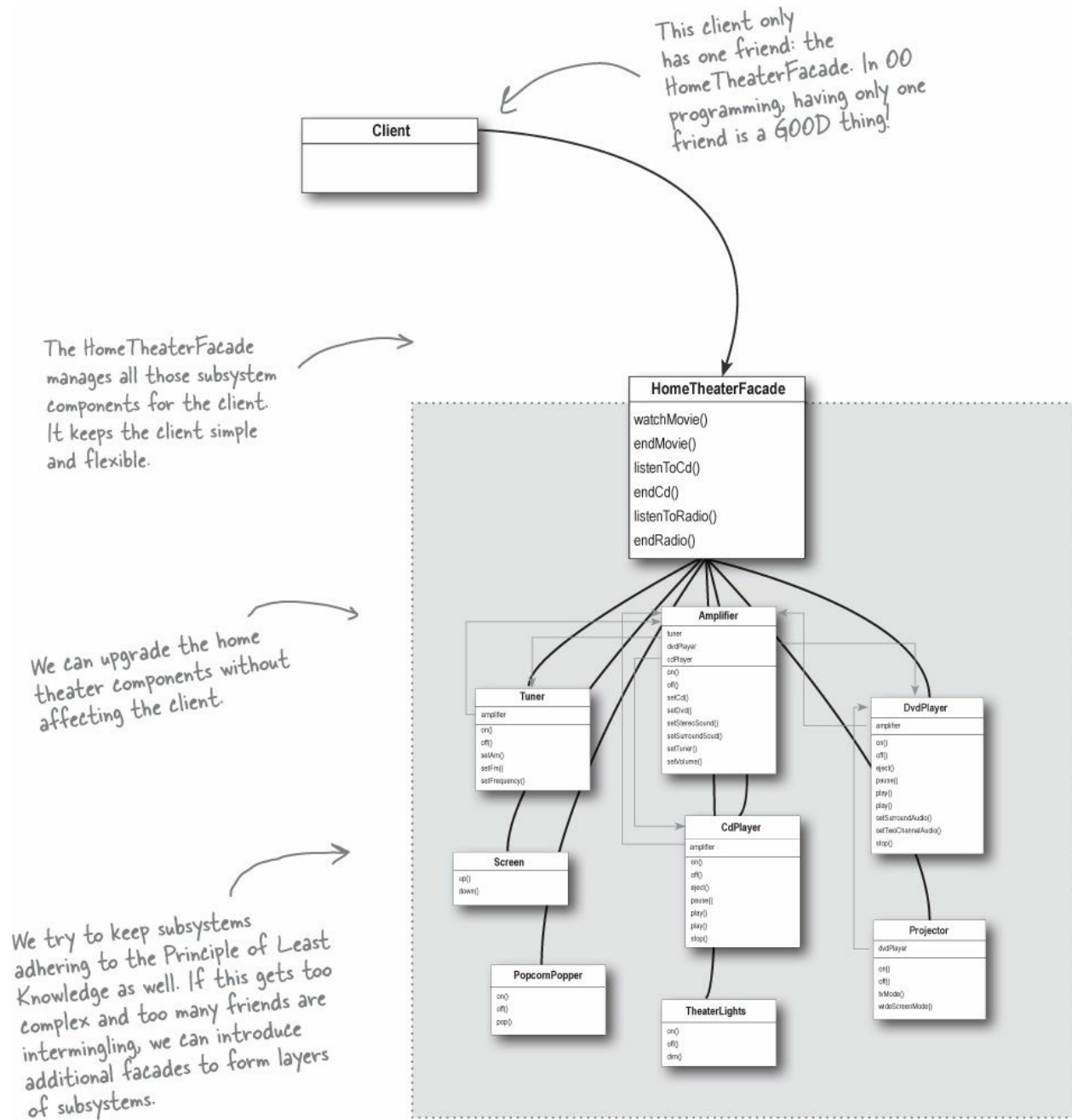
**HARD HAT AREA.
WATCH OUT FOR
FALLING ASSUMPTIONS**

BRAIN POWER

Q: Can you think of a common use of Java that violates the Principle of Least Knowledge? Should you care?

A: 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.

OO Basics

OO Principles

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

abstraction
encapsulation
polymorphism
inheritance

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

OO Patterns

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

Facade - Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

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

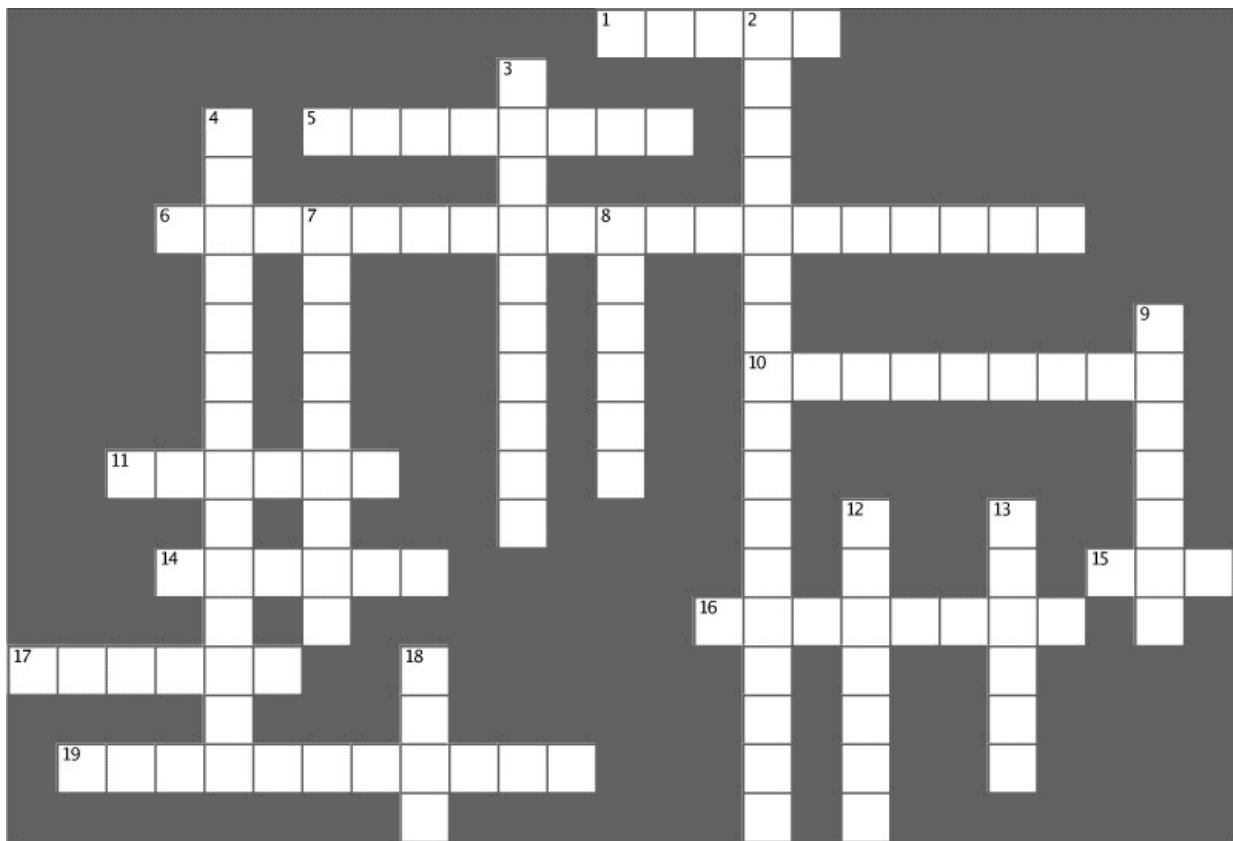


BULLET POINTS

- 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	Down
<p>1. True or false? Adapters can wrap only one object.</p> <p>5. An Adapter _____ an interface.</p> <p>6. Movie we watched (five words).</p> <p>10. If in Britain, you might need one of these (two words).</p> <p>11. Adapter with two roles (two words).</p> <p>14. Facade still _____ low-level access.</p> <p>15. Ducks do it better than Turkeys.</p> <p>16. Disadvantage of the Principle of Least Knowledge: too many _____.</p> <p>17. A _____ simplifies an interface.</p> <p>19. New American dream (two words).</p>	<p>2. Decorator called Adapter this (three words).</p> <p>3. One advantage of Facade.</p> <p>4. Principle that wasn't as easy as it sounded (two words).</p> <p>7. A _____ adds new behavior.</p> <p>8. Masquerading as a Duck.</p> <p>9. Example that violates the Principle of Least Knowledge: System.out._____.</p> <p>12. No movie is complete without this.</p> <p>13. Adapter client uses the _____ interface.</p> <p>18. An Adapter and a Decorator can be said to _____ an object.</p>

SHARPEN YOUR PENCIL SOLUTION

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Here's our 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();
        }
    }
}
```

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

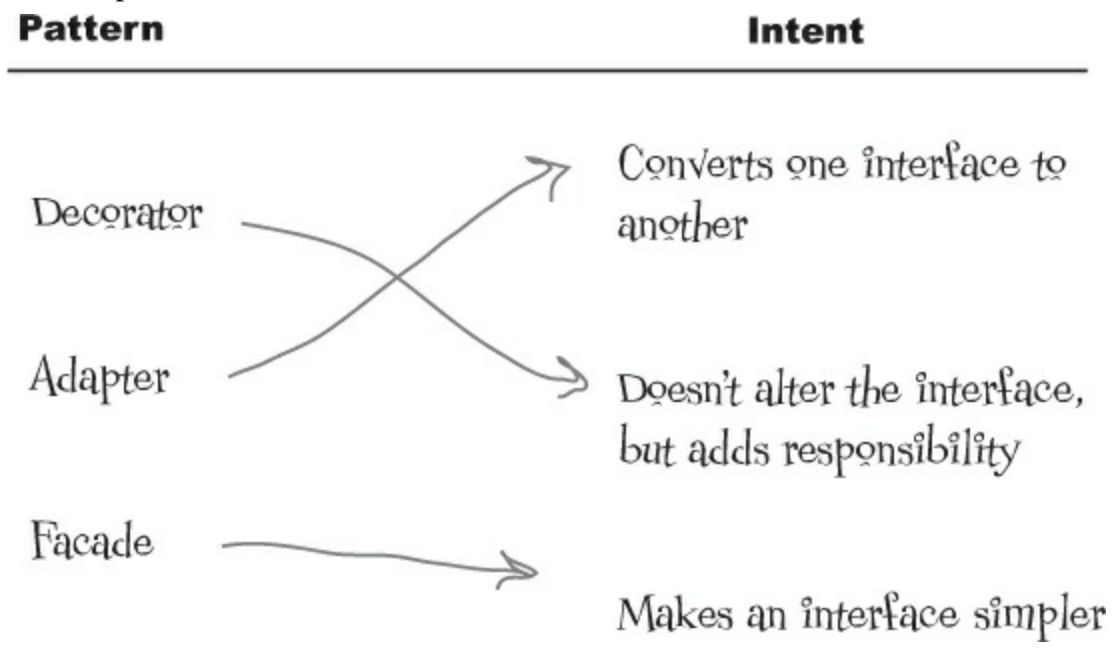
You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumeration.

Notice we keep the type parameter generic so this will work for any type of object.

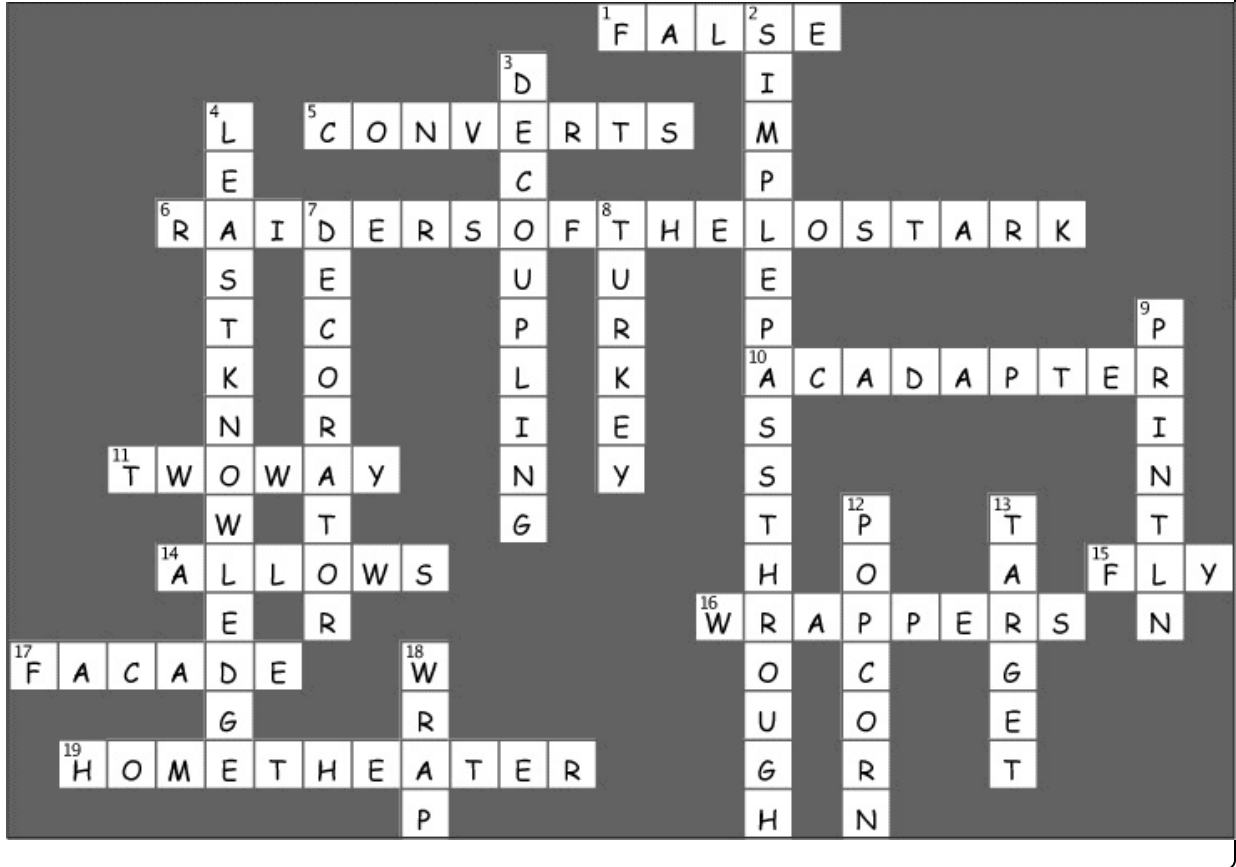
```
public class IteratorEnumeration implements Enumeration<Object> {  
    Iterator<?> iterator;  
  
    public IteratorEnumeration(Iterator<?> iterator) {  
        this.iterator = iterator;  
    }  
  
    public boolean hasMoreElements() {  
        return iterator.hasNext();  
    }  
  
    public Object nextElement() {  
        return iterator.next();  
    }  
}
```

WHO DOES WHAT? SOLUTION

Match each pattern with its intent:



DESIGN PATTERNS CROSSWORD SOLUTION



Chapter 8. The Template Method Pattern: Encapsulating Algorithms

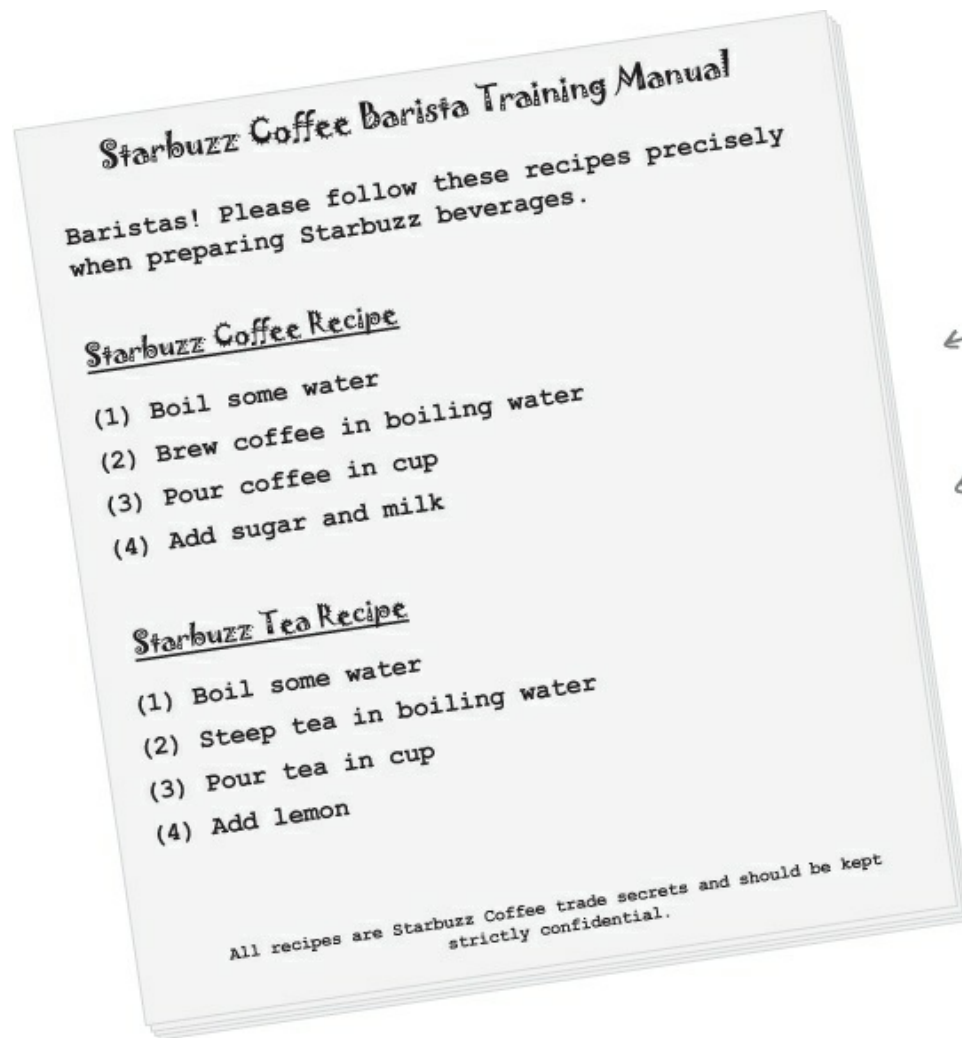


We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas...what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.

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:



← The recipe for coffee looks a lot like the recipe for tea, doesn't it?
←

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:

```
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");  
    }  
}
```

Handwritten annotations:

- Here's our Coffee class for making coffee. (points to the class declaration)
- Here's our recipe for coffee, straight out of the training manual. (points to the prepareRecipe method)
- Each of the steps is implemented as a separate method. (points to the individual methods within prepareRecipe)
- 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. (points to each of the four individual methods)

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?

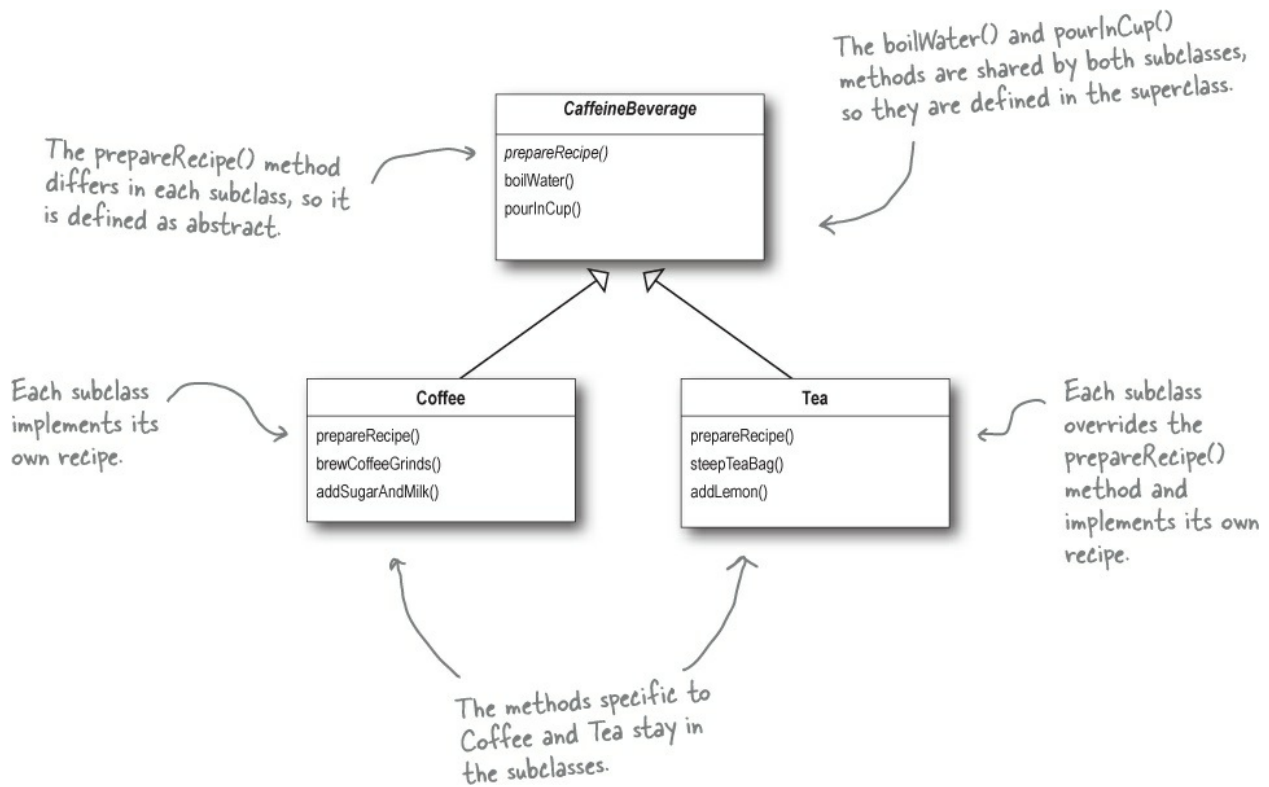


DESIGN PUZZLE

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:

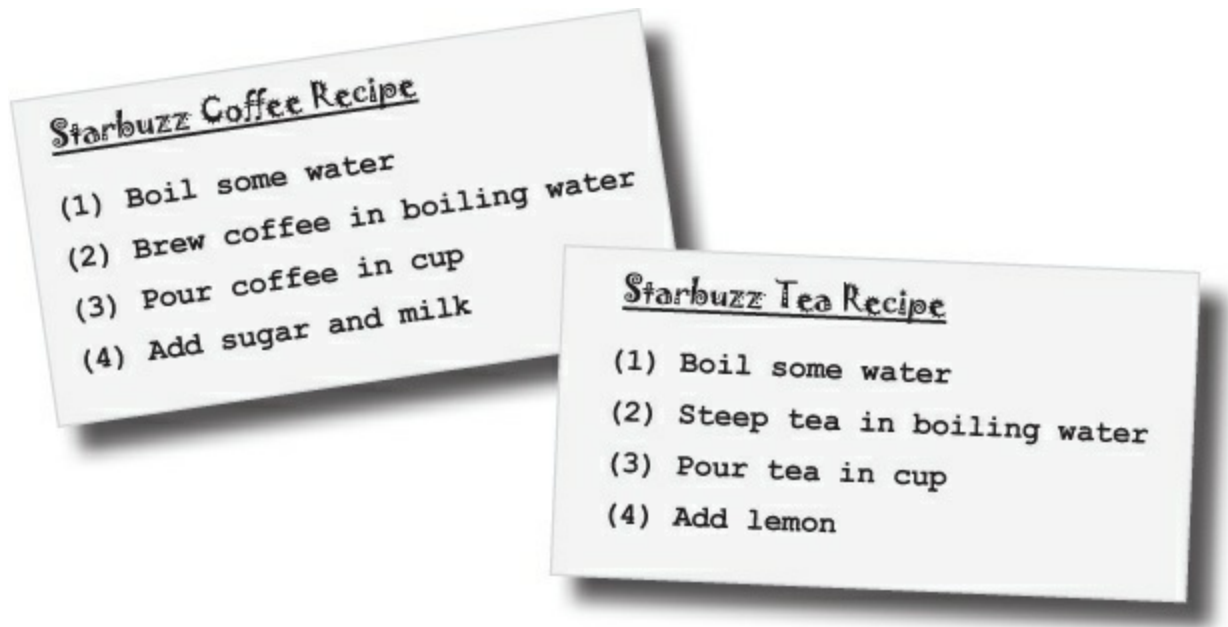


BRAIN POWER

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

Taking the design further...

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



Notice that both recipes follow the same algorithm:

① **Boil some water.**

NOTE

These two are already abstracted into the base class.

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

NOTE

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

③ **Pour the resulting beverage into a cup.**

④ **Add the appropriate condiments to the beverage.**


So, can we find a way to abstract `prepareRecipe()` too? Yes, let's find out...

Abstracting `prepareRecipe()`

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

- ① The first problem we have is that Coffee uses `brewCoffeeGrinds()` and `addSugarAndMilk()` methods, while Tea uses `steepTeaBag()` and

addLemon() methods.

Coffee		Tea
<pre>void prepareRecipe() { boilWater(); brewCoffeeGrinds(); pourInCup(); addSugarAndMilk(); }</pre>		<pre>void prepareRecipe() { boilWater(); steepTeaBag(); pourInCup(); addLemon(); }</pre>

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

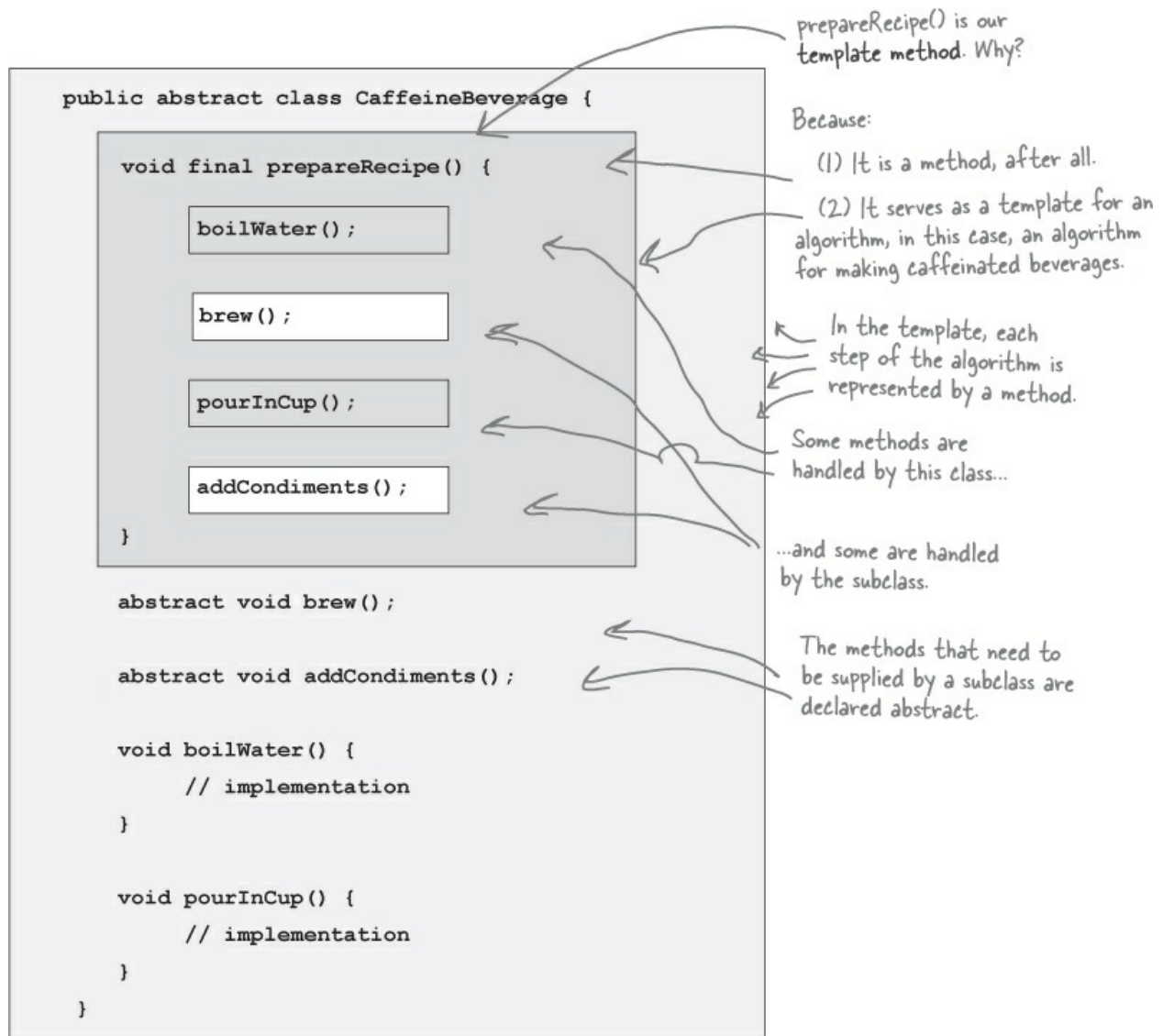
Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

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...



Behind the Scenes

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...

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

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

- ② Then we call the template method:

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

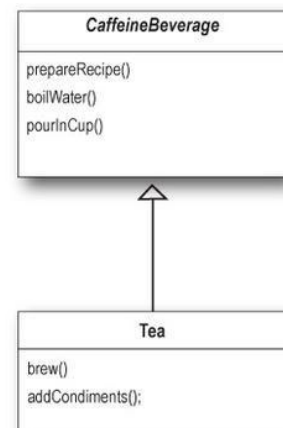
```
boilWater();  
brew();  
pourInCup();  
addCondiments();
```

The prepareRecipe() method controls the algorithm. No one can change this, and it counts on subclasses to provide some or all of the implementation.

which follows the algorithm for making caffeine beverages...

- ③ First we boil water:

```
boilWater();
```



which happens in CaffeineBeverage.

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

```
brew();
```

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

```
pourInCup();
```

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

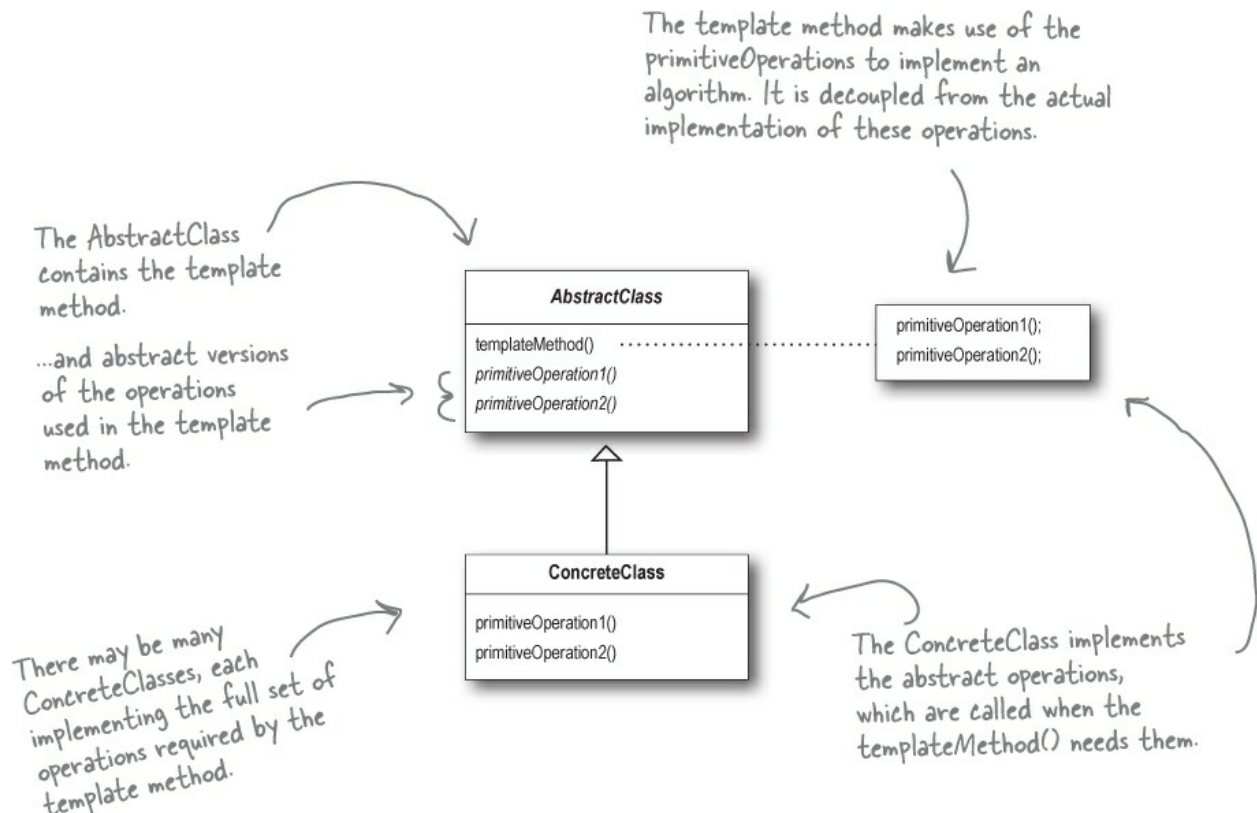
You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

NOTE

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.

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

```
abstract class AbstractClass {  
  
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
    }  
  
    abstract void primitiveOperation1();  
  
    abstract void primitiveOperation2();  
  
    void concreteOperation() {  
        // implementation here  
    }  
}
```

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

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

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

CODE WAY UP CLOSE

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

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

```
abstract class AbstractClass {  
  
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
        hook();  
    }  
  
    abstract void primitiveOperation1();  
  
    abstract void primitiveOperation2();  
  
    final void concreteOperation() {  
        // implementation here  
    }  
  
    void hook() {}  
  
}
```

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

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

↑
A concrete method, but it does nothing!

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

Hooked on Template Method...

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

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.



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();  
        }  
    }
```

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()`.

```
    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;  
    }
```

```
}
```

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();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

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

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

```

← Create a tea.
 ← A coffee.
 ← And call prepareRecipe() on both!

And let's give it a run...

```

File Edit Window Help send-more-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

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

```

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

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



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

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

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

A: 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: 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: 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:



NOTE

The Hollywood Principle

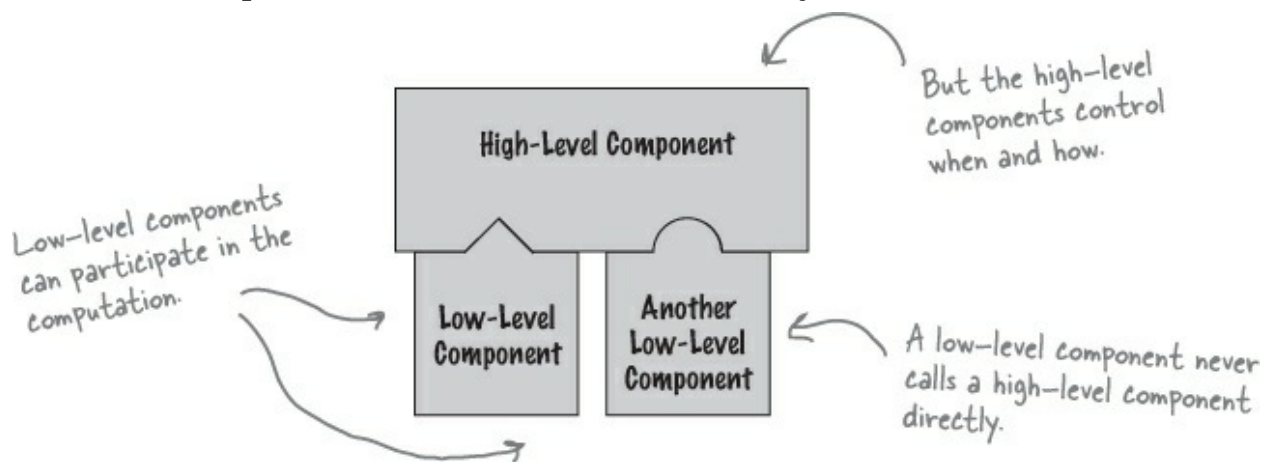
Don't call us, we'll call you.

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

The Hollywood Principle gives us a way to prevent “dependency rot.” Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on

sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

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

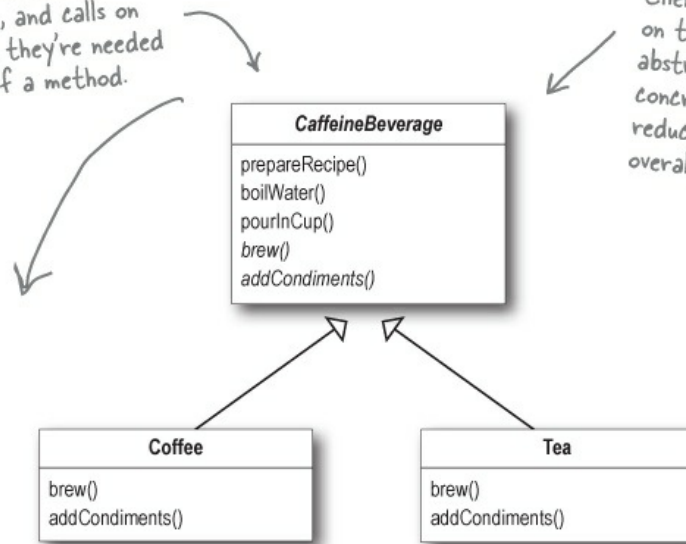


The Hollywood Principle and Template Method

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

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

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



The subclasses are used simply to provide implementation details.

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

BRAIN POWER

What other patterns make use of the Hollywood Principle?

The Factory Method, Observer; any others?

THERE ARE NO DUMB QUESTIONS

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

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

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

WHO DOES WHAT?

Match each pattern with its description:

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

NOTE

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...

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

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

```
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.

Think of this as the template method.

```
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  
}
```

`compareTo()` is the method we need to implement to "fill out" the template method.

This is a concrete method, already defined in the `Arrays` class.

We've got some ducks to sort...

Let's say you have an array of ducks that you'd like to sort. How do you do it? Well, the sort template method in `Arrays` gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the `compareTo()` method... Make sense?



We've got an array of
Ducks we need to sort.

No, it doesn't.
Aren't we supposed to be
subclassing something? I thought
that was the point of Template
Method. An array doesn't subclass
anything, so I don't get how we'd
use sort().



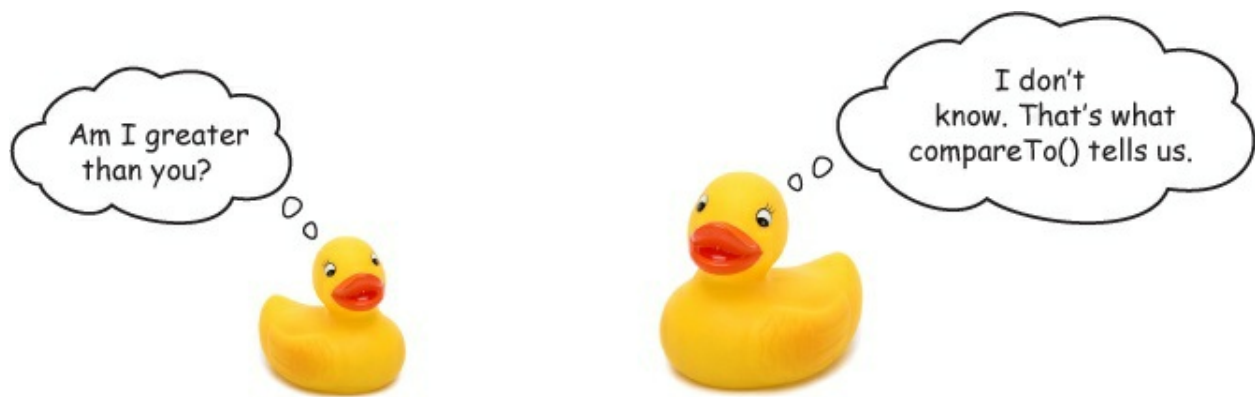
Good point. Here's the deal: the designers of `sort()` wanted it to be useful across all arrays, so they had to make `sort()` a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because `sort()` really isn't defined in our superclass, the `sort()` method needs to know that you've implemented the

compareTo() method, or else you don't have the piece needed to complete the sort algorithm.

To handle this, the designers made use of the Comparable interface. All you have to do is implement this interface, which has one method (surprise): compareTo().

What is compareTo()?

The compareTo() method compares two objects and returns whether one is less than, greater than, or equal to the other. sort() uses this as the basis of its comparison of objects in the array.



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...