```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // code to set fan to high
    }

    public void medium() {
        speed = MEDIUM;
        // code to set fan to medium
    }

    public void low() {
        speed = LOW;
        // code to set fan to low
    }

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

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

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

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().

Adding Undo to the CeilingFan commands

Now let's tackle adding undo to the various CeilingFan commands. To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting. Here's the code for the CeilingFanHighCommand:

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

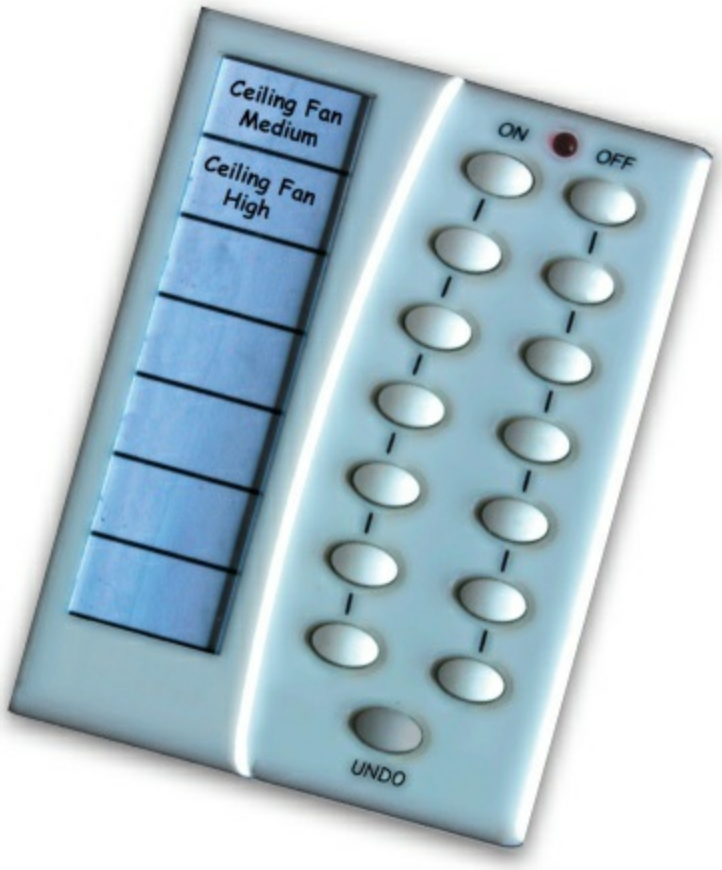
To undo, we set the speed of the fan back to its previous speed.

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.

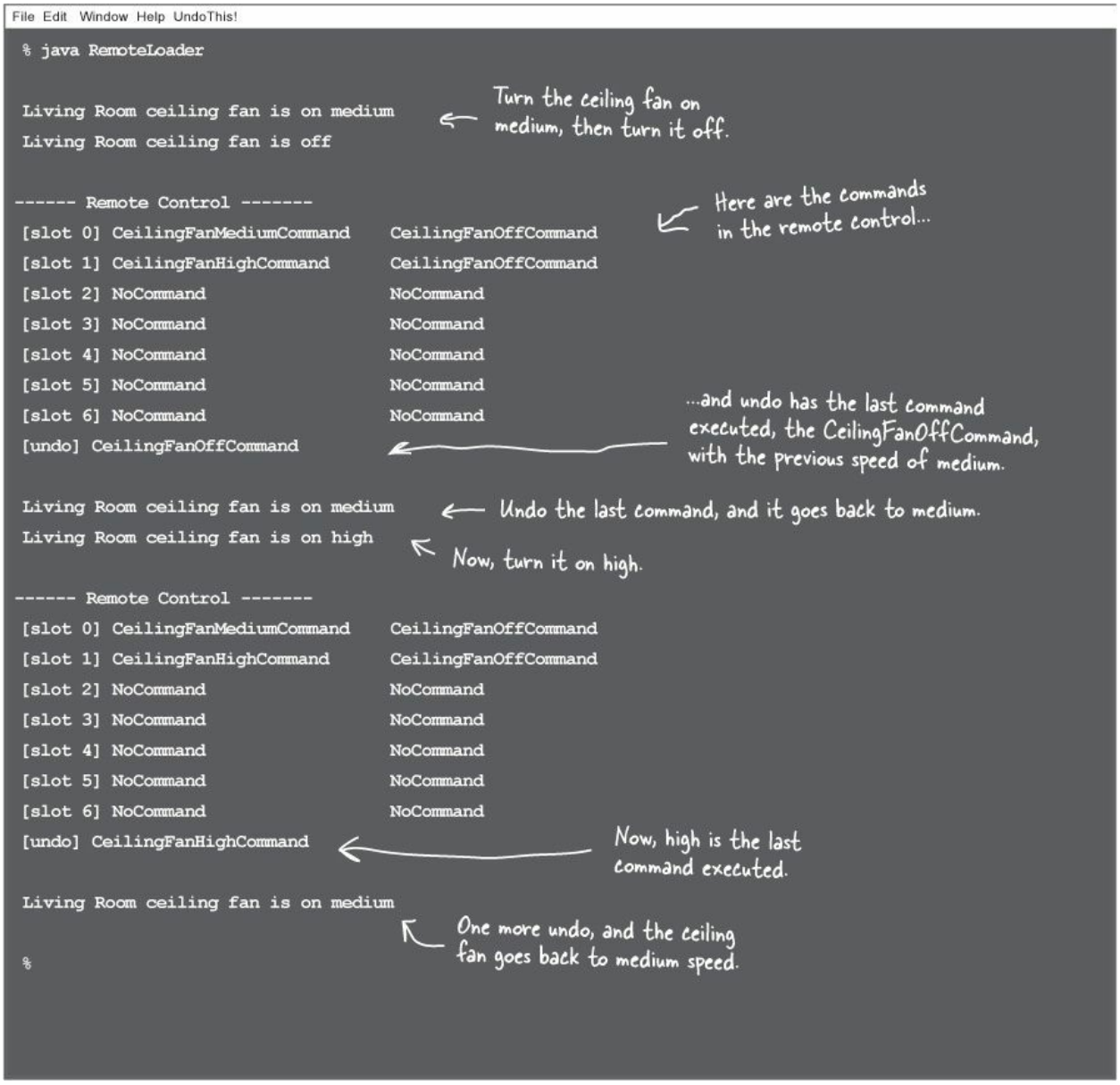
Undo! It should go back to medium...

Turn it on to high this time.

And, one more undo; it should go back to medium.

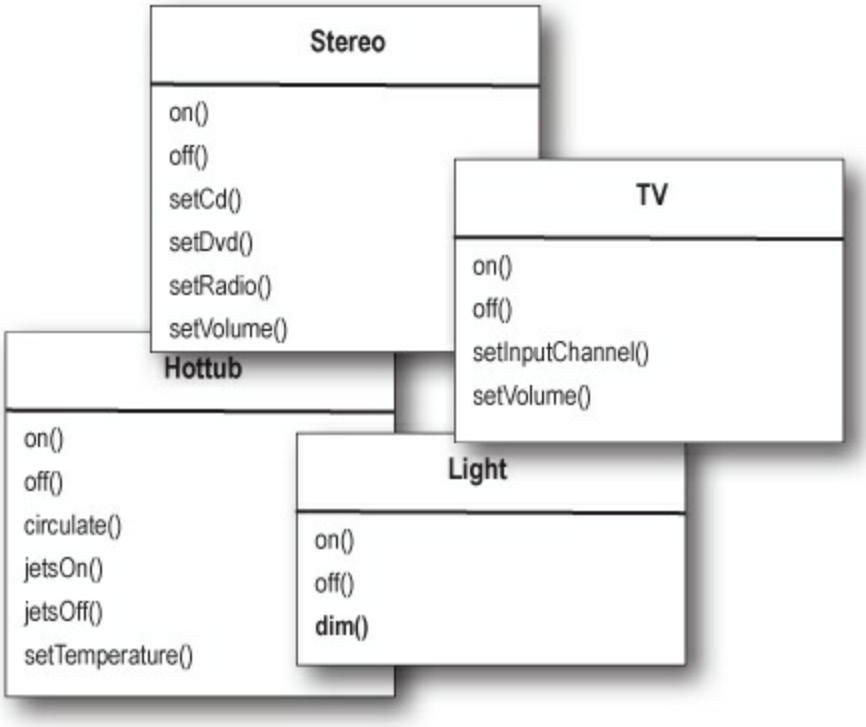
Testing the ceiling fan...

Okay, let’s fire up the remote, load it with commands, and push some buttons!



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?


Stereo
on()
off()
setCd()
setDvd()
setRadio()
setVolume()
TV
on()
off()
setInputChannel()
setVolume()
Hottub
on()
off()
circulate()
jetsOn()
jetsOff()
setTemperature()
Light
on()
off()
dim()


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();

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

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

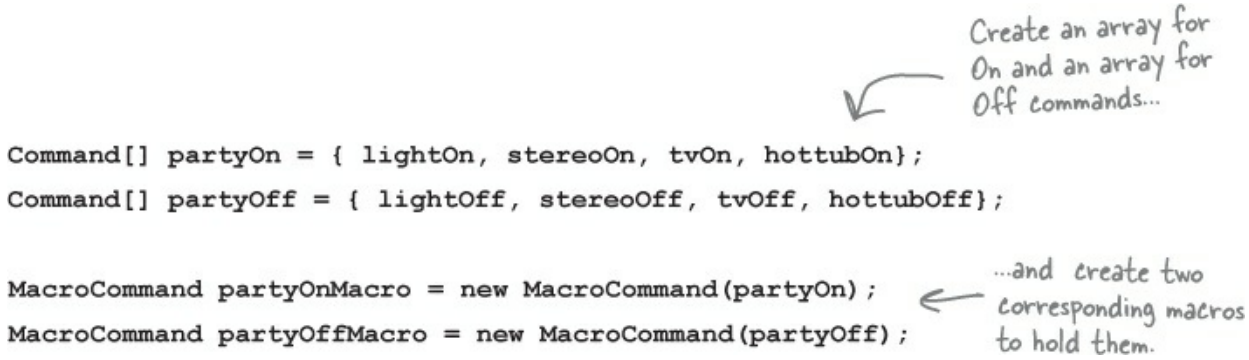
Now create all the On commands to control them.

SHARPEN YOUR PENCIL

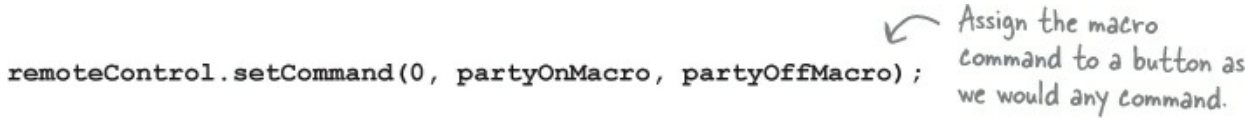
We will also need commands for the off buttons. Write the code to create those here:

② Next we create two arrays, one for the On commands and one for the

Off commands, and load them with the corresponding commands:



③ Then we assign MacroCommand to a button like we always do:



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?**

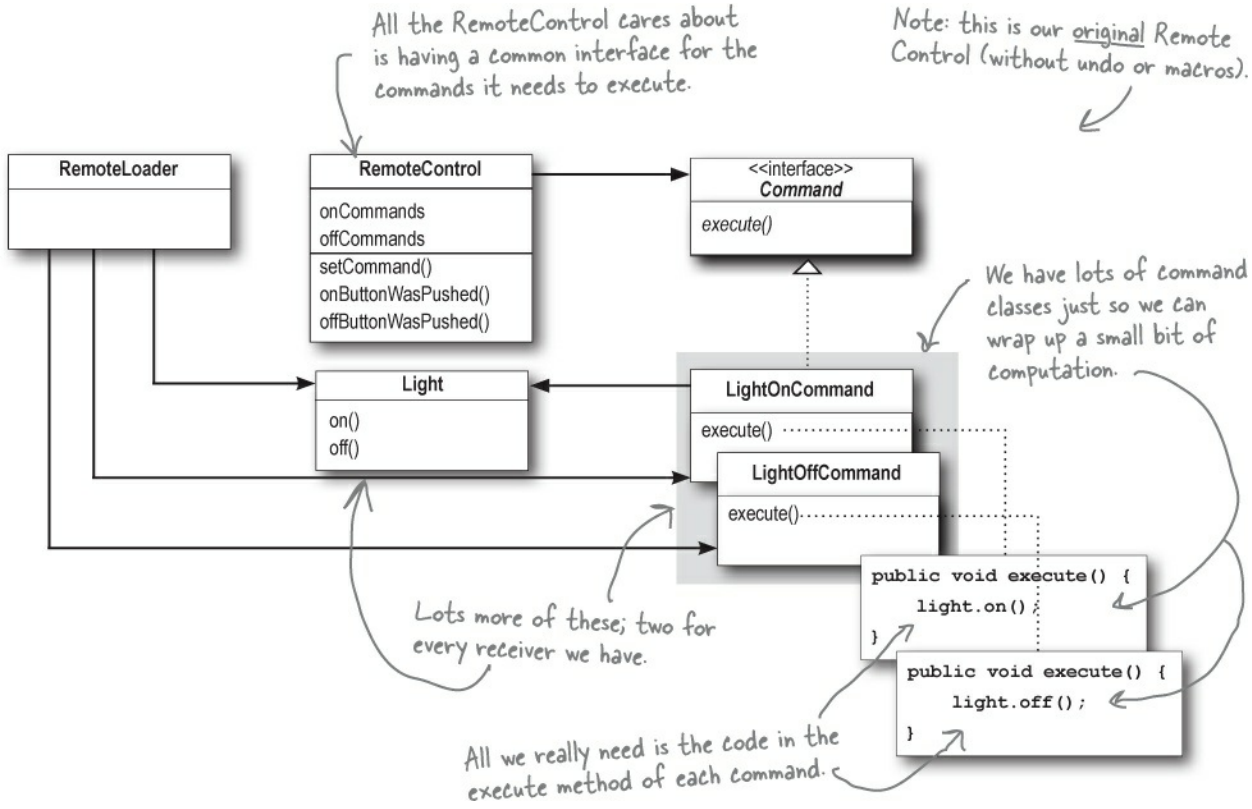
A: **A:** Great question. It's pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

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.

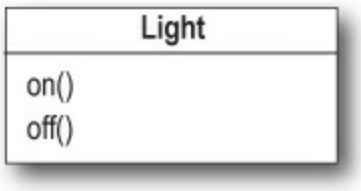
> **NOTE**
>
> If you aren’t yet familiar with lambda expressions (they were added in Java 8) they can take some getting used to. You should be able to follow along over the next few pages, but consult a Java reference to get up to speed on the syntax and semantics if you need to.

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.

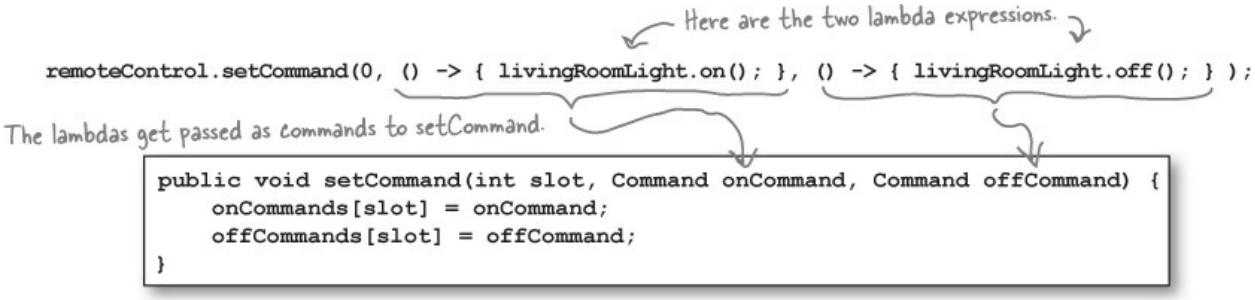
```
Light livingRoomLight = new Light("Living Room");
```



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

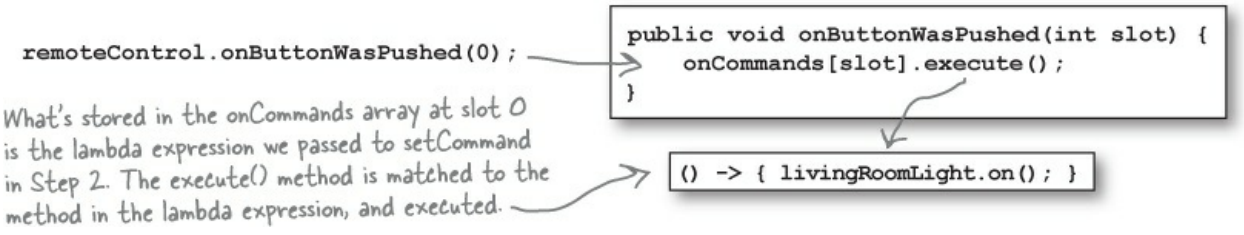
This is where the magic happens. Now, instead of creating

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

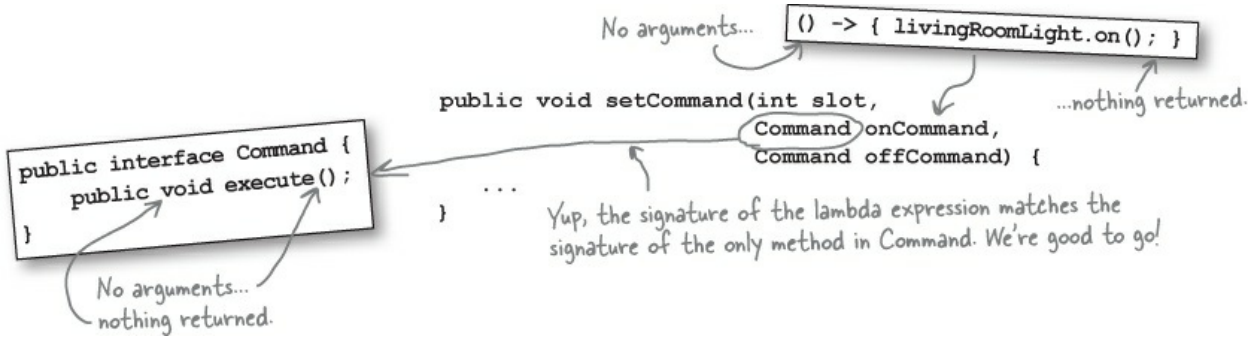
This step doesn't change either. Except now, when we call the remote's onButtonWasPushed(0) method, the command that's in slot 0 is a function object (created by the lambda expression). When we call execute() on the command, that method is matched up with the method defined by the lambda expression, which is then executed.



Well, we did say "magic" didn't we?

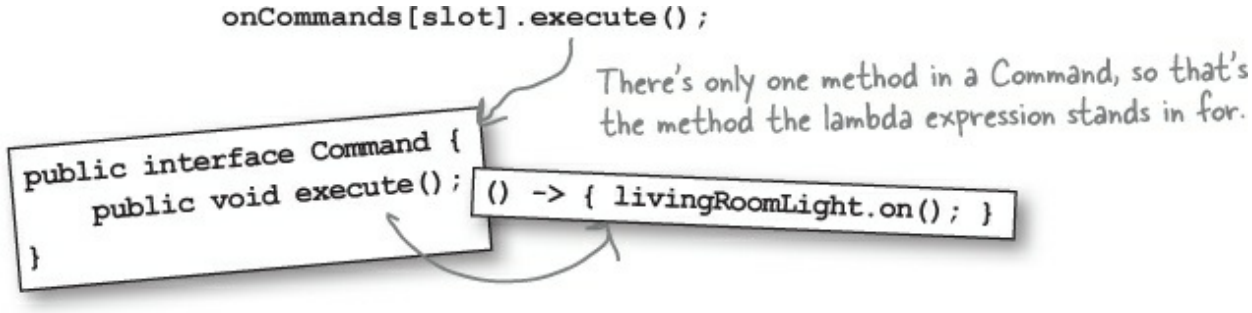
Just kidding... it's actually not all that magical. We're using lambda expressions to stand in for Command objects, and the Command interface has just one method: execute(). The lambda expression we use must have a compatible signature with this method — and it does: execute() takes no arguments (neither does our lambda expression), and returns no value (neither does our lambda expression), so the compiler is happy.

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



The compiler checks to see if the Command interface has one method that matches the lambda expression, and it does: execute().

Then, when we call execute() on that command, the method in the lambda expression is called:



Just remember: as long as the interface of the parameter we're passing the lambda expression to has one (and only one!) method, and that method has a compatible signature with the lambda expression, this will work.

Simplifying even more with method references

We can simplify our code even more using *method references*. When the lambda expression you're passing calls just one method, you can pass a method reference in place of the lambda expression. Like this:

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:

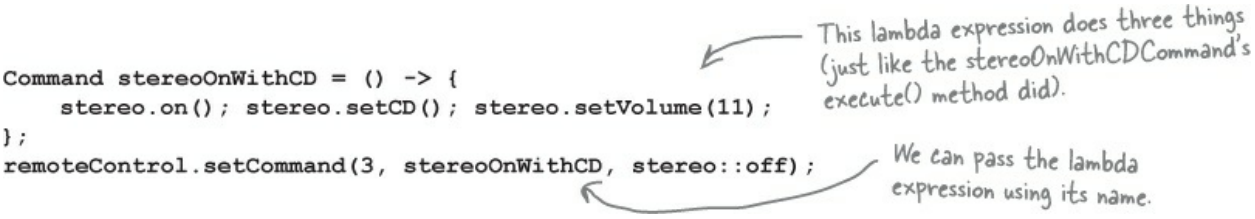
```
stereo.off();
```

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:



Notice that we use Command as the type of the lambda expression. The lambda expression will match the Command interface's execute() method, and the Command parameter we're passing it to in the setCommand() method.

Test the remote control with lambda expressions

To use lambda expressions to simplify the code for the original Remote Control implementation (without undo), we're going to change the code in the RemoteLoader to replace the concrete Command objects with lambda expressions, and change the RemoteControl constructor to use lambda expressions instead of a NoCommand object. Once we've done that, we can

delete all the concrete Command classes (LightOnCommand, LightOffCommand, HottubOnCommand, HottubOffCommand, and so on). And that's it. Everything else stays the same. Make sure you *don't* delete the Command interface; you still need that to match the type of the function objects created by the lambda expressions that get stored in the remote control, and passed to the various methods.

Here's the new code for the RemoteLoader class:

```
public class RemoteLoader {
    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan = new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("Main house");
        Stereo stereo = new Stereo("Living Room");

        remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);
        remoteControl.setCommand(1, kitchenLight::on, kitchenLight::off);
        remoteControl.setCommand(2, ceilingFan::high, ceilingFan::off);

        Command stereoOnWithCD = () -> {
            stereo.on(); stereo.setCD(); stereo.setVolume(11);
        };
        remoteControl.setCommand(3, stereoOnWithCD, stereo::off);
        remoteControl.setCommand(4, garageDoor::up, garageDoor::down);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

We've removed all the code to create concrete Command objects (and we deleted all those classes too). Now our code's a lot more concise (and we've gone from 22 classes to 9).

We're using method references everywhere we have simple one-method commands, and a full lambda expression for where we need to do more than one method call.

(You can think of a method reference as a compact lambda expression. They're really the same thing; a method reference is just shorthand for a lambda expression that calls just one method.)

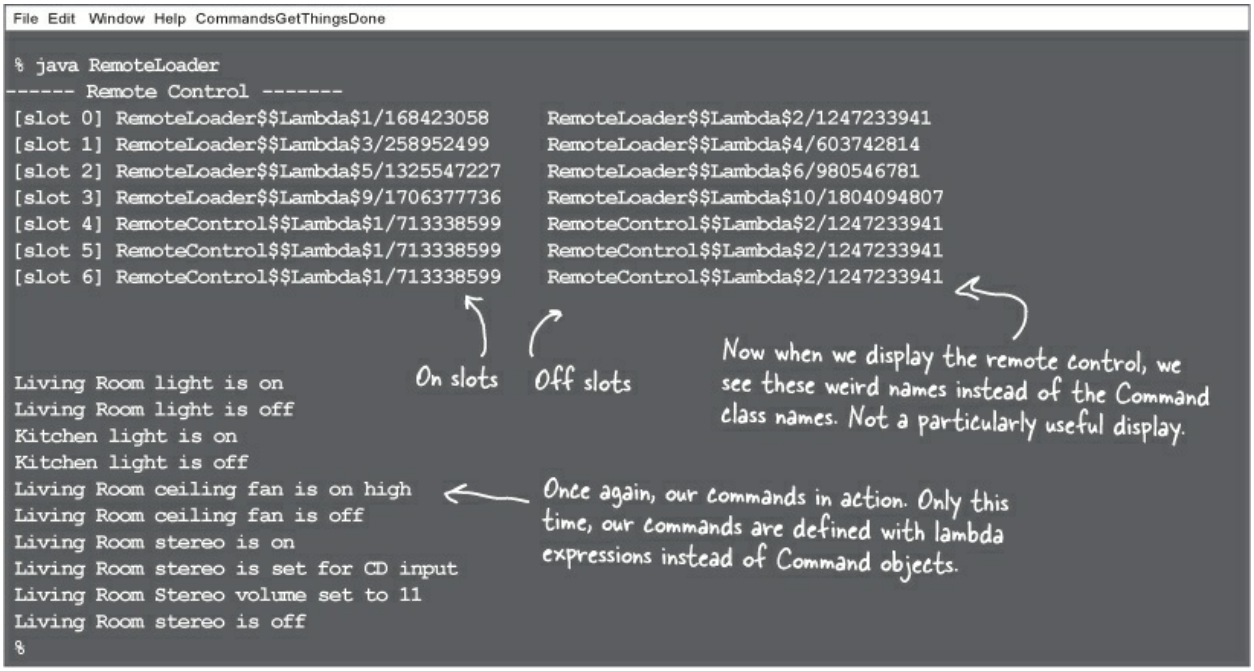
And don't forget, we need to modify the RemoteControl constructor to remove the code to construct NoCommand objects, and replace those with lambda expressions too:


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

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

        for (int i = 0; i < 7; i++) {
            onCommands[i] = () -> { };
            offCommands[i] = () -> { };
        }
    }
    // rest of the code here
}
```

We've removed the code to create a NoCommand object.

Instead of a NoCommand object, we use a lambda expression that does nothing! (Just like the execute() method of the NoCommand object did nothing.)

Check out the results of all those lambda expression commands...



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.

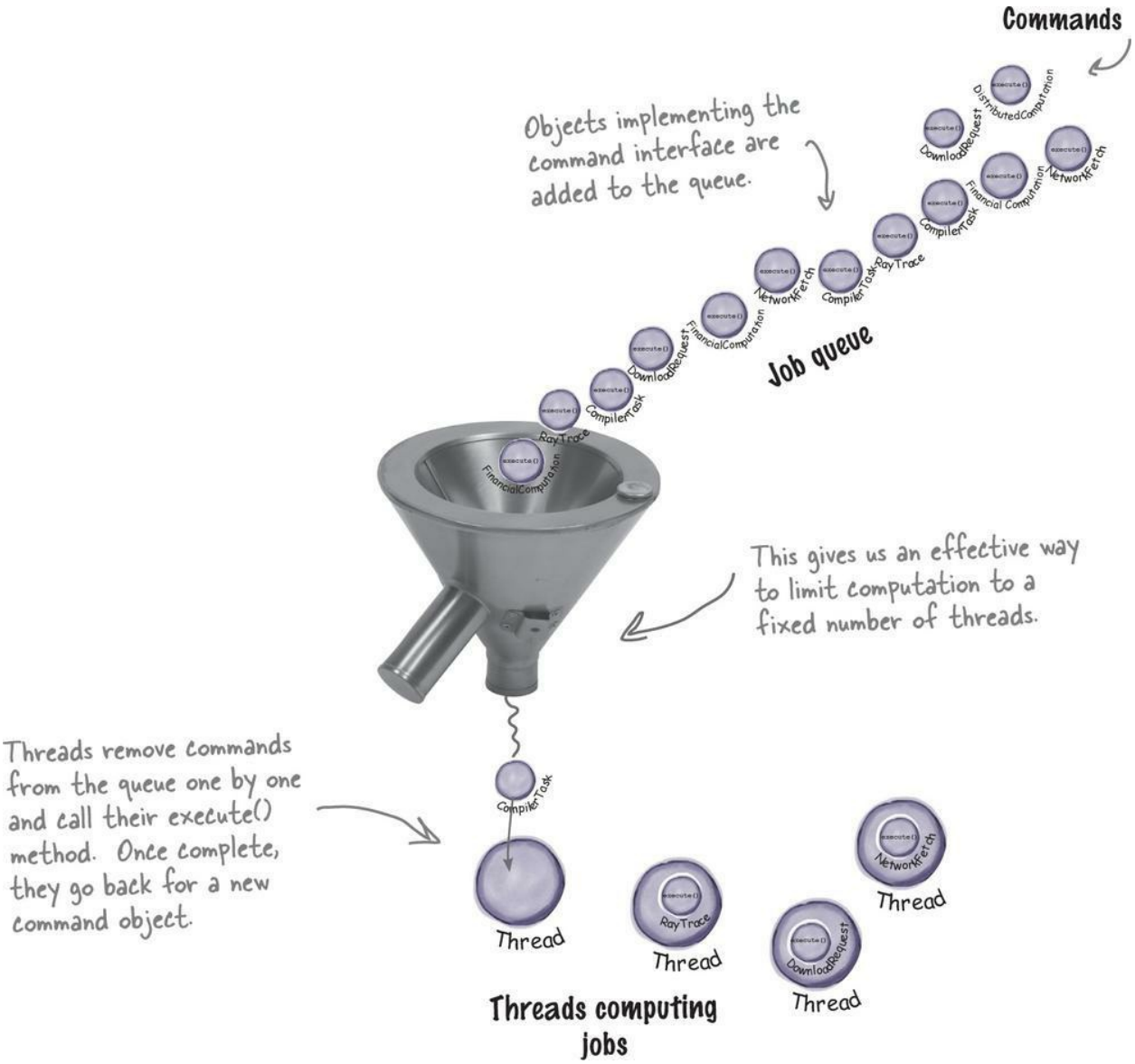
As you can see from the output, lambda expressions don't have nice class names. That's because their names are assigned internally by the Java runtime and Java has no idea what these lambda expressions mean; to Java, they're just function objects that happen to match a method in an interface.

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

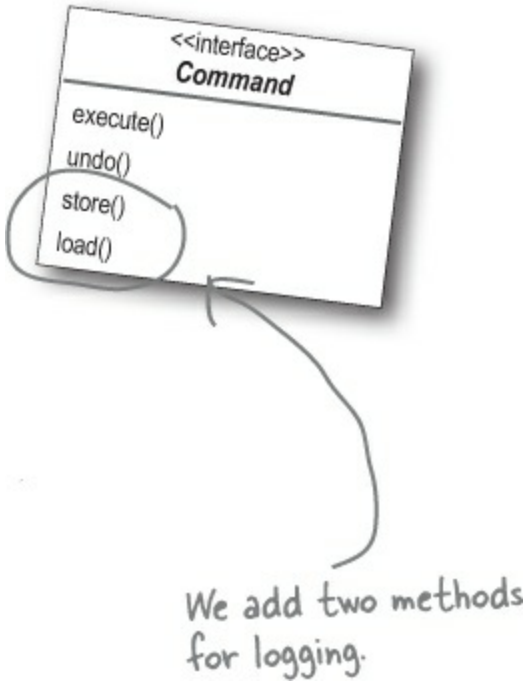
How might a web server make use of such a queue? What other applications can you think of?

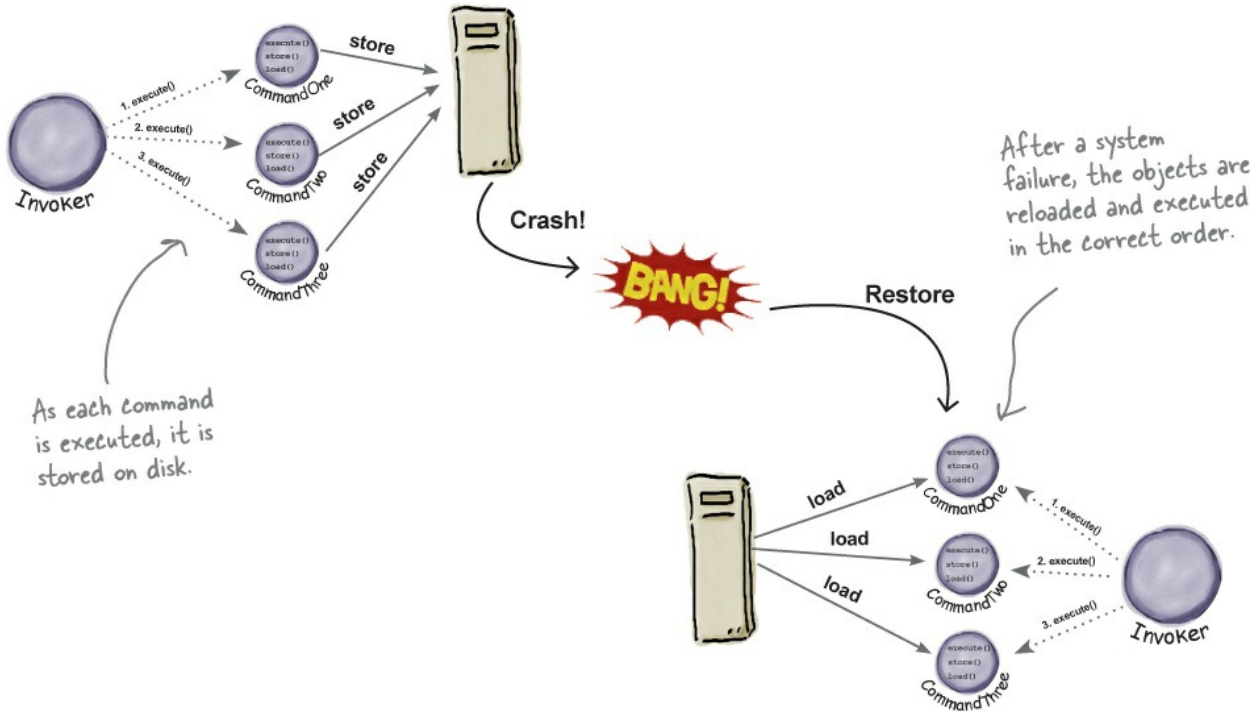
More uses of the Command Pattern: logging requests

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.

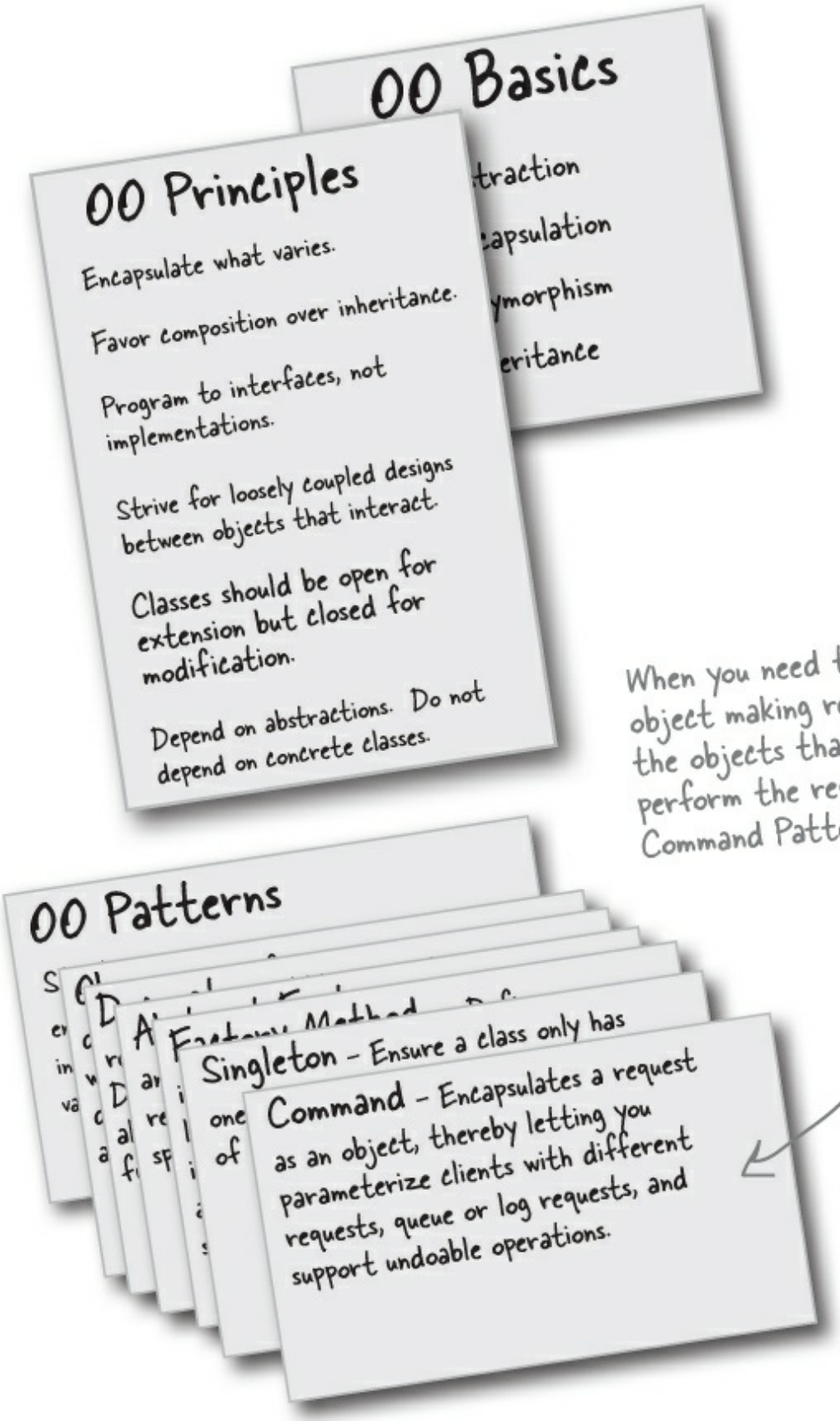
Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last check point, and if there is a 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
traction
apsulation
ymorphism
eritance
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
When you need to decouple an object making requests from the objects that know how to perform the requests, use the Command Pattern.
OO Patterns
Singleton – Ensure a class only has
Command – Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

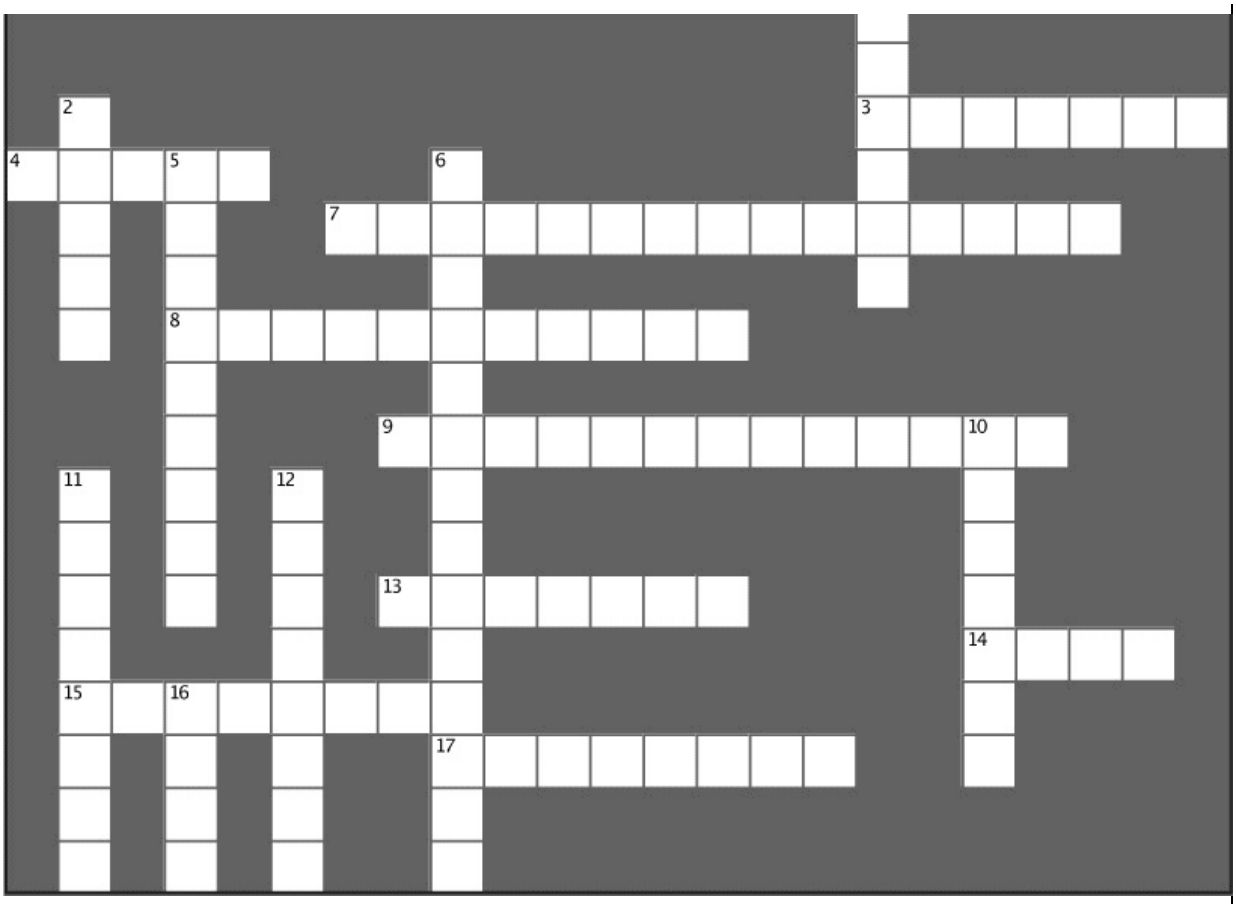
BULLET POINTS

- 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.

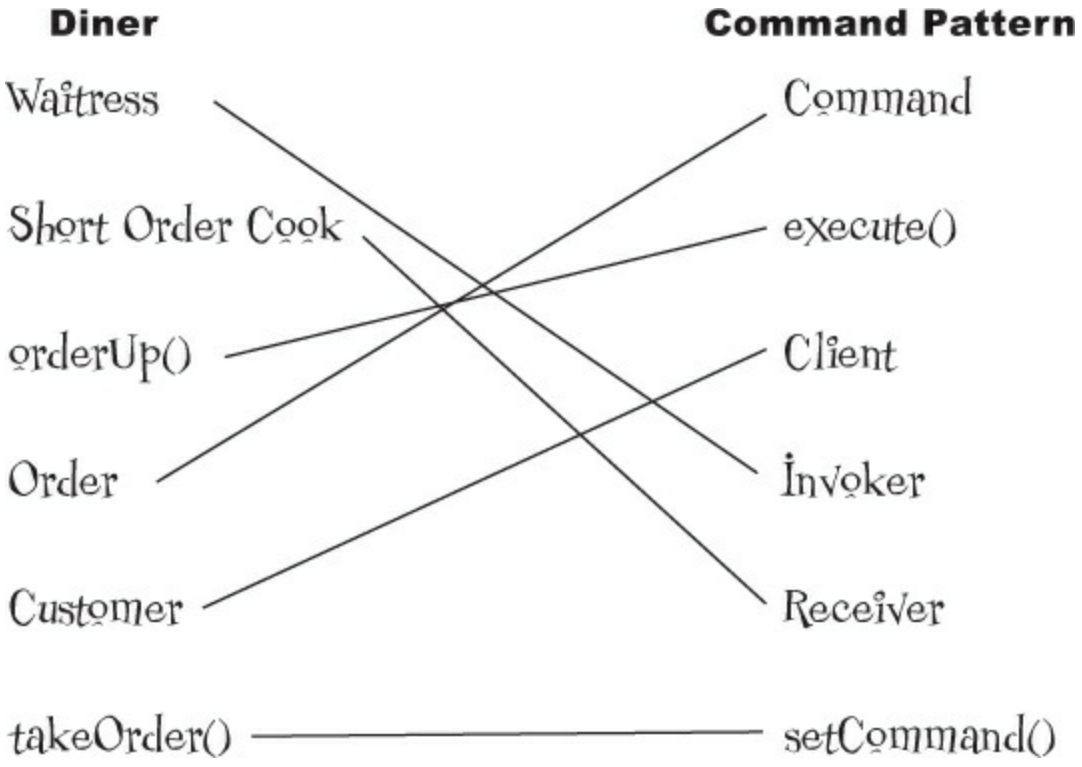


Across	Down
3. The Waitress was one. 4. A command _____ a set of actions and a receiver. 7. Dr. Seuss diner food. 8. Our favorite city. 9. Act as the receivers in the remote control. 13. Object that knows the actions and the receiver. 14. Another thing Command can do. 15. Object that knows how to get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

WHO DOES WHAT? SOLUTION

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

Pattern



SHARPEN YOUR PENCIL SOLUTION

Here's the code for the GarageDoorOpenCommand class.

```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;

    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }
    public void execute() {
        garageDoor.up();
    }
}
```

Here's the output:

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

Light is on
Garage Door is Open
%
```

EXERCISE SOLUTION

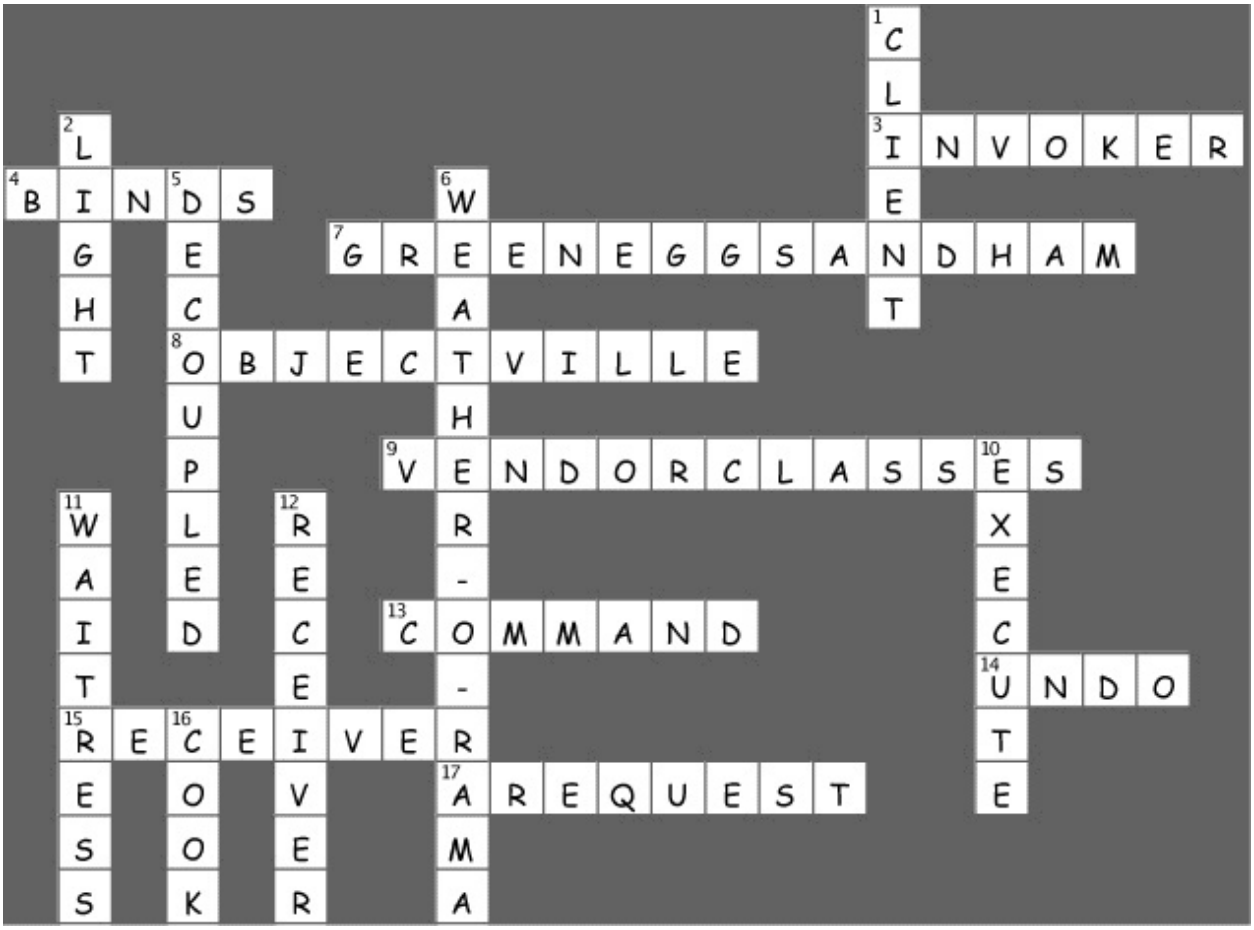
Here is the undo() method for the MacroCommand.

```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

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

    public void undo() {
        for (int i = commands.length - 1; i > = 0; i--) {
            commands[i].undo();
        }
    }
}
```

SHARPEN YOUR PENCIL SOLUTION

Here is the code to create commands for the off button.

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```


1 C L
2 L
3 I N V O K E R
4 B I N D S
5 D E C
6 W E A T
7 G R E E N E G G S A N D H A M
H
T
8 O B J E C T V I L L E
U
P
9 V E N D O R C L A S S
10 E S
11 W A I T
L E D
12 R E C E
R - C O M M A N D
X E C
14 U N D O T E
15 R E C E I V E R
E S S
O O K
V E R
17 A R E Q U E S T
M A
E

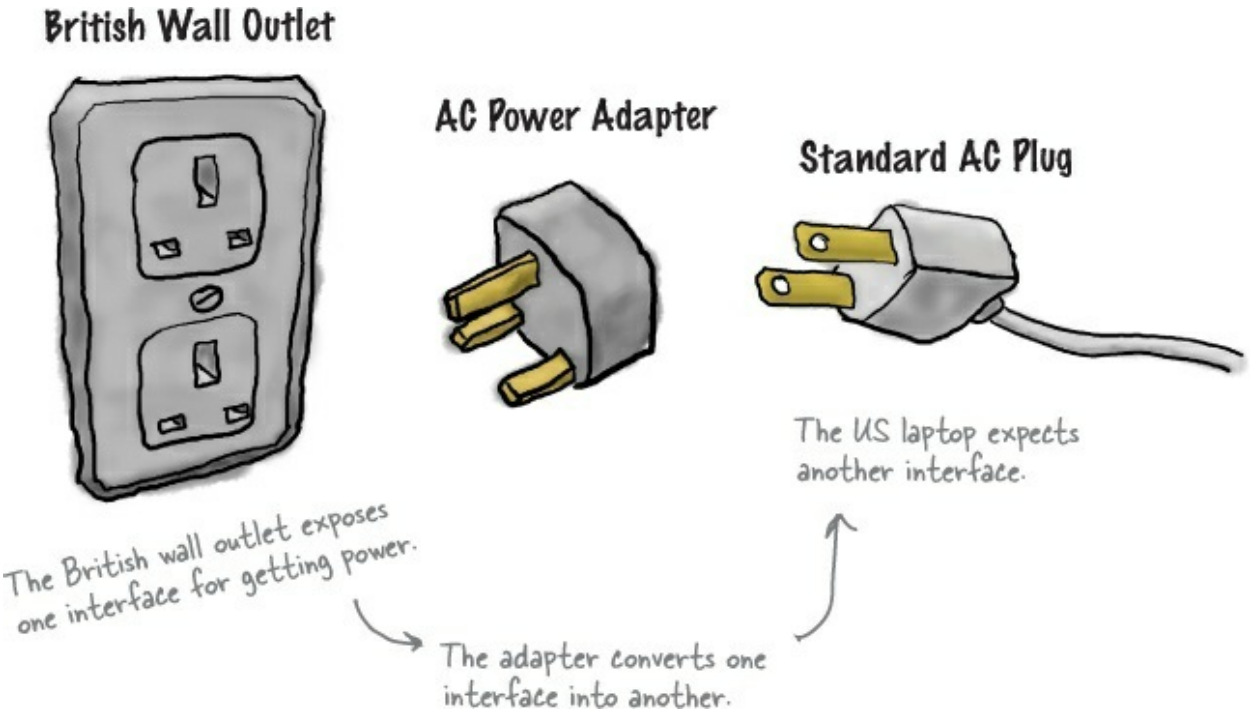
Chapter 7. The Adapter and Facade Patterns: Being Adaptive

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...



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.

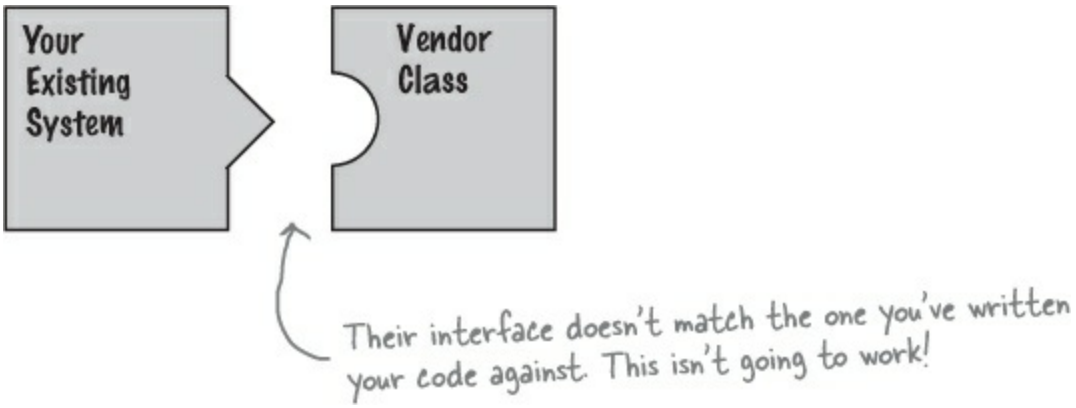
> **NOTE**
>
> How many other real-world adapters can you think of?

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.

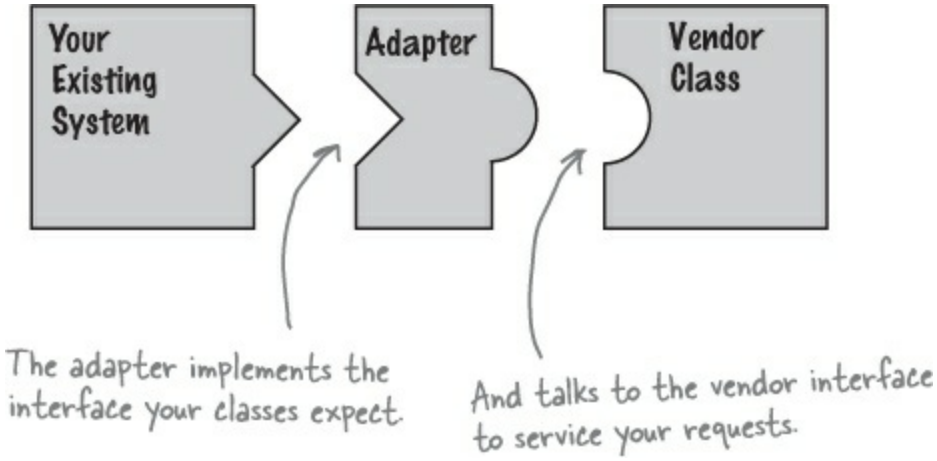
Okay, that's the real world; what about object-oriented adapters? Well, our OO adapters play the same role as their real-world counterparts: they take an interface and adapt it to one that a client is expecting.

Object-oriented adapters

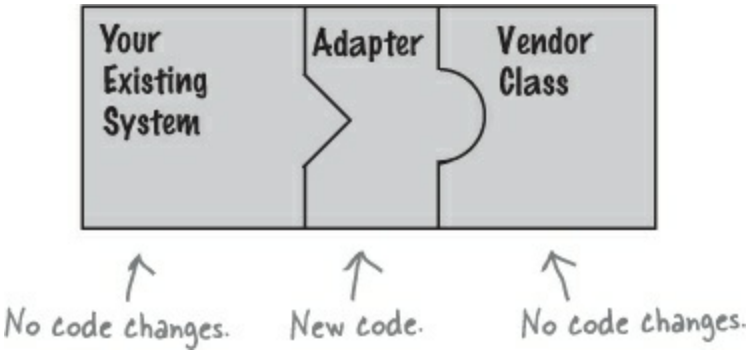
Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



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:

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

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

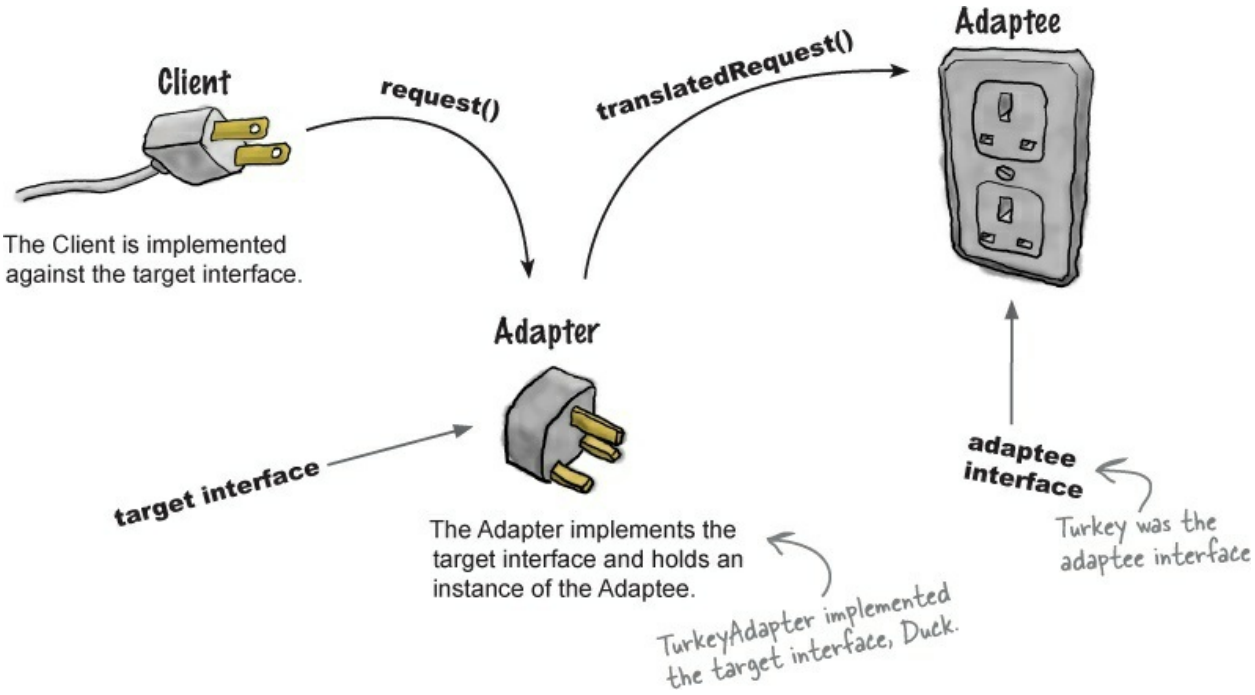
The Turkey gobbles and flies a short distance.

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

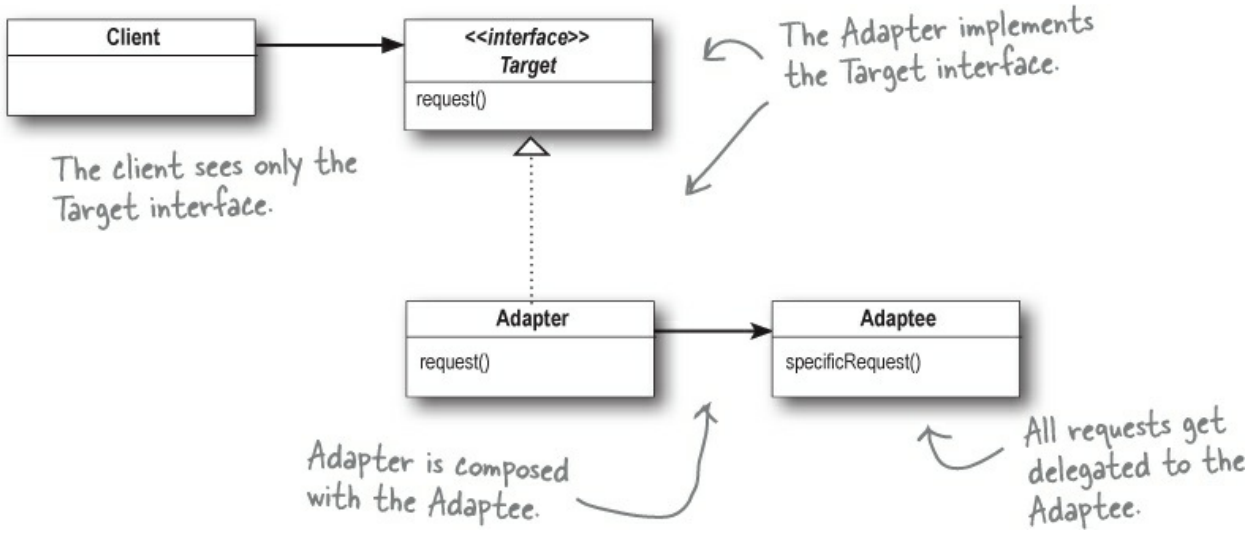
Enough ducks, turkeys, and AC power adapters; let’s get real and look at the official definition of the Adapter Pattern:

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:



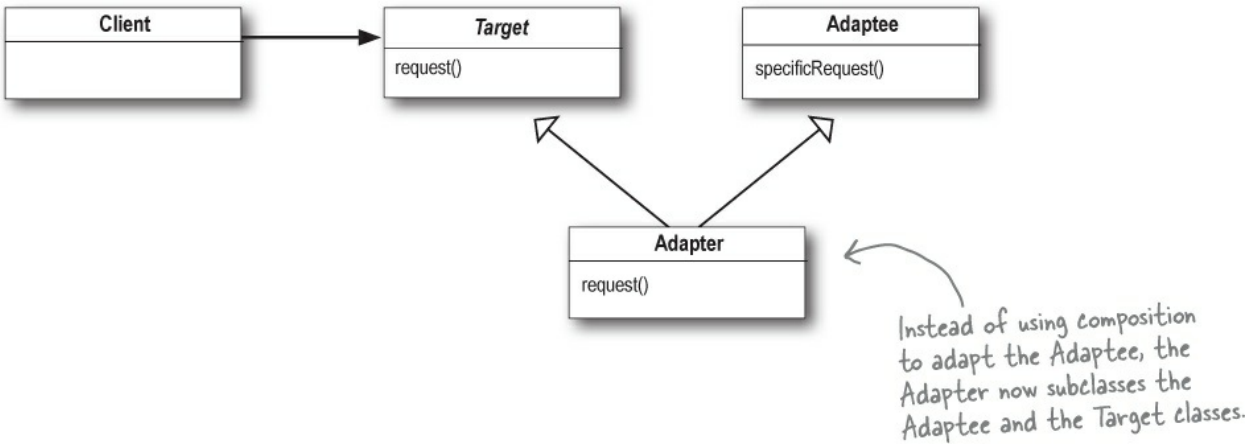
The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

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.

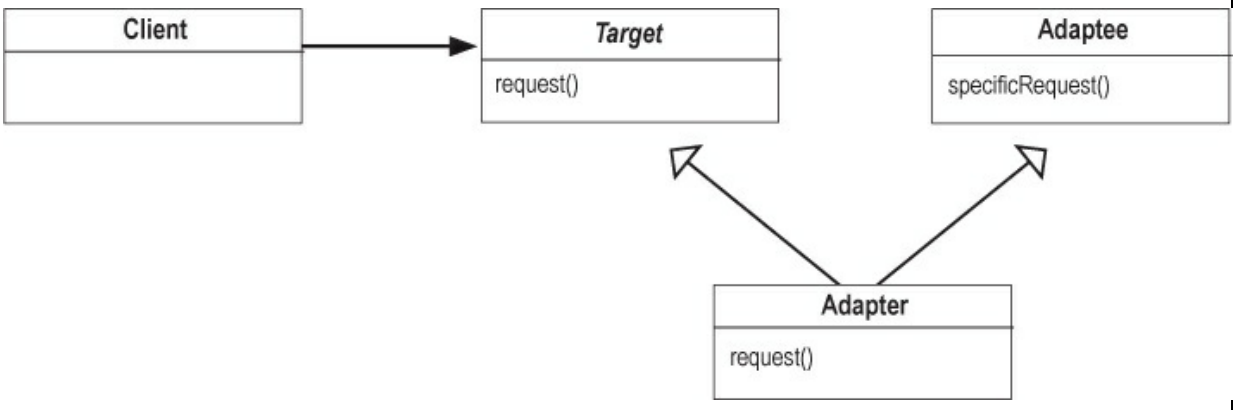
BRAIN POWER

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?

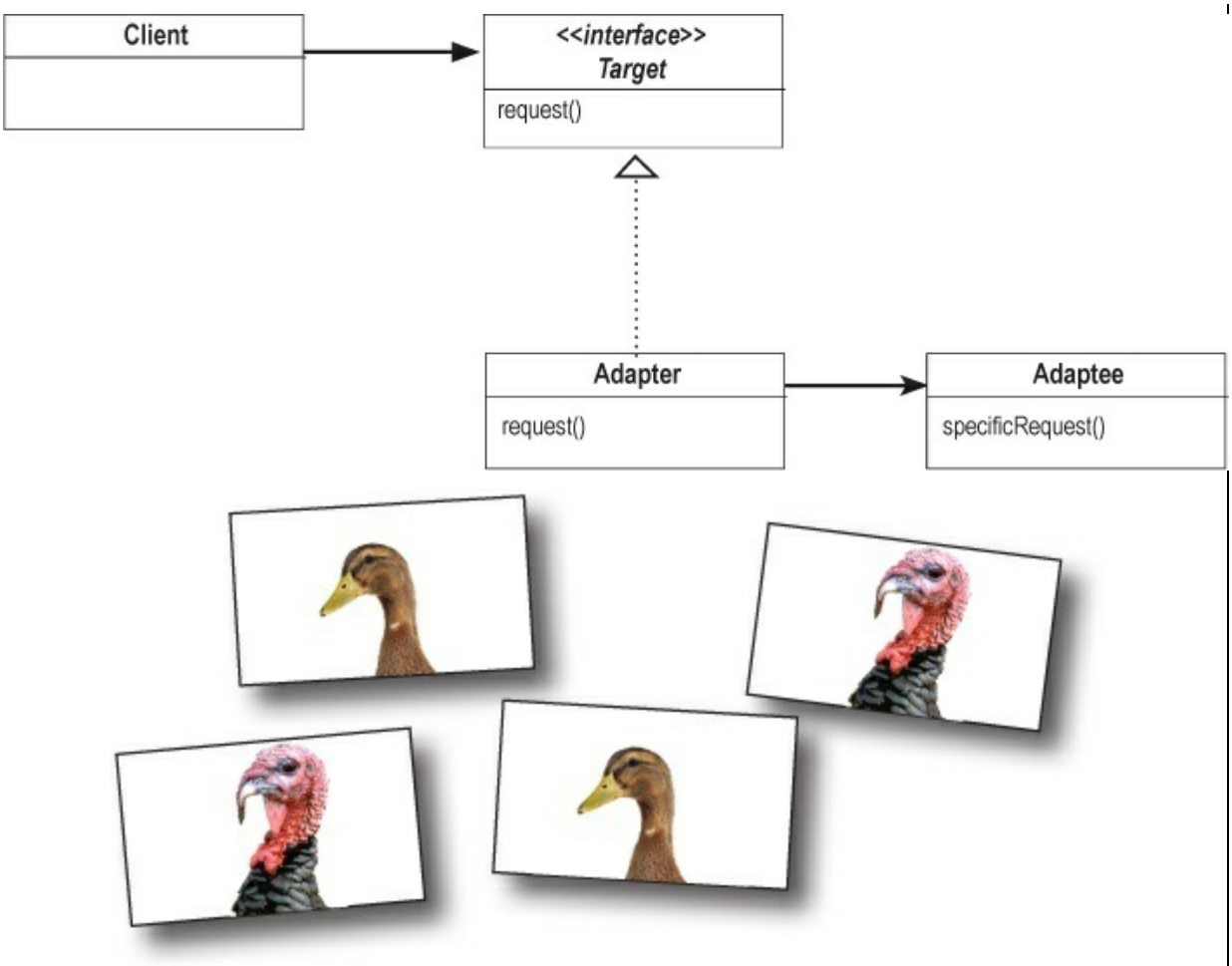
DUCK MAGNETS

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



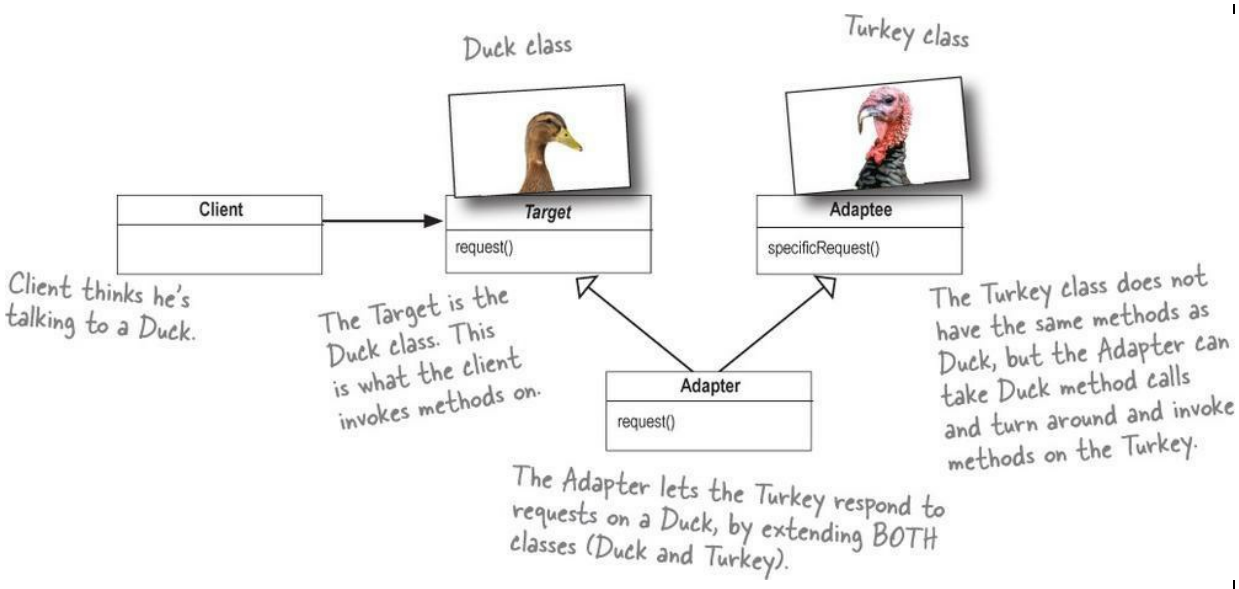
Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.

DUCK MAGNETS ANSWER

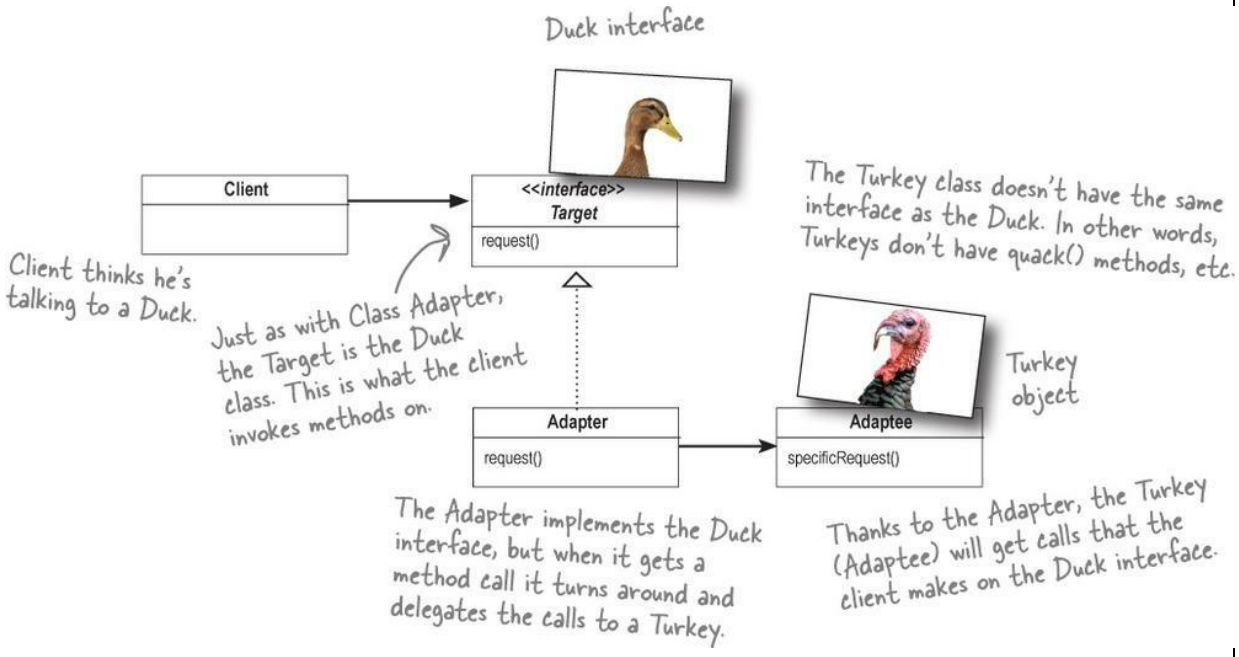
NOTE

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 *and* 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!	

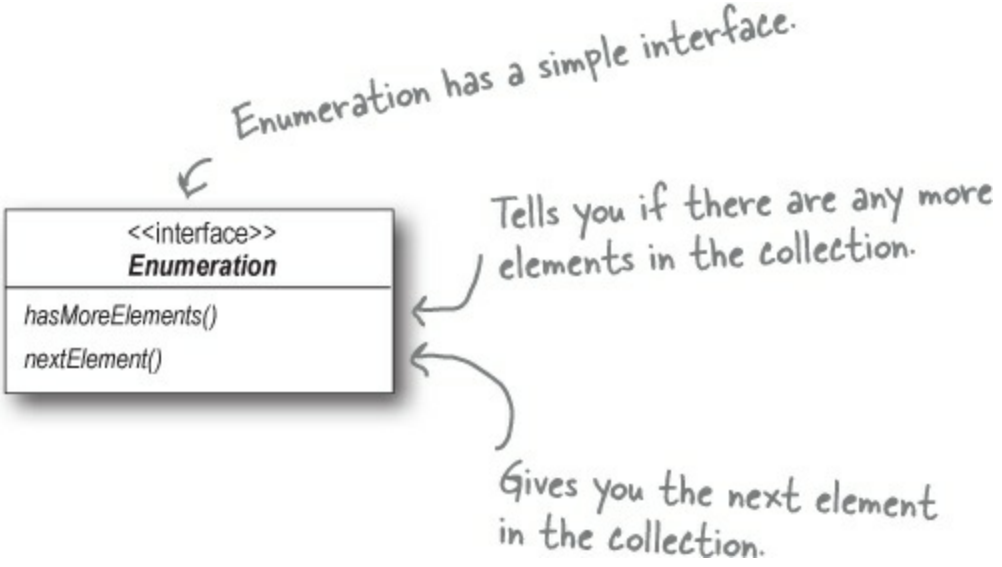
Real-world adapters

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

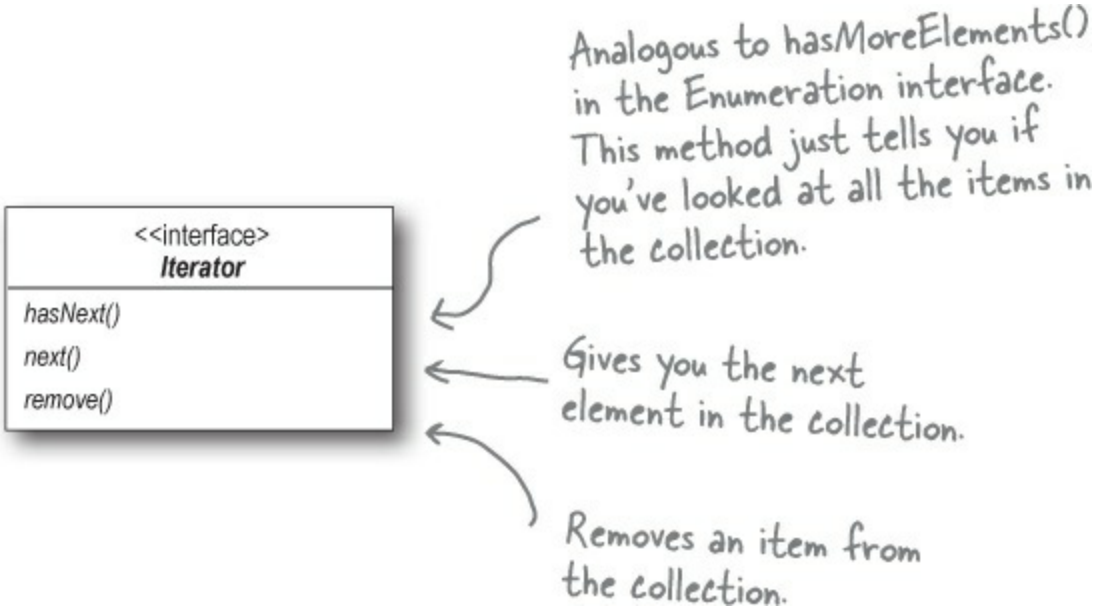
If you’ve been around Java for a while you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, elements(), which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without

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.

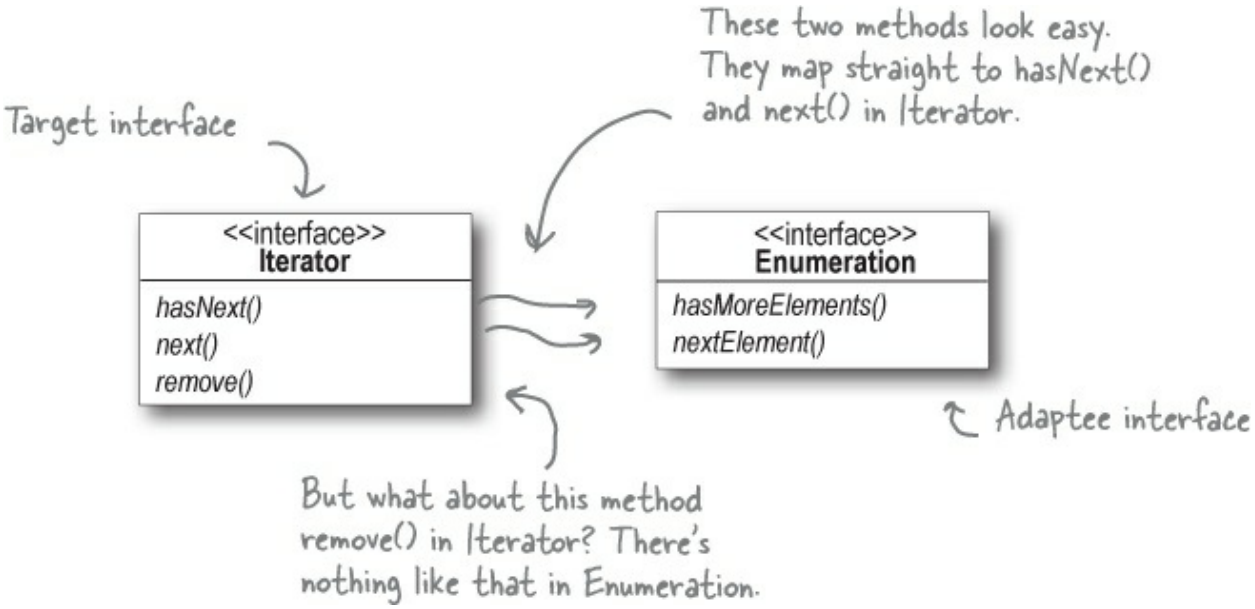


And today...

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.

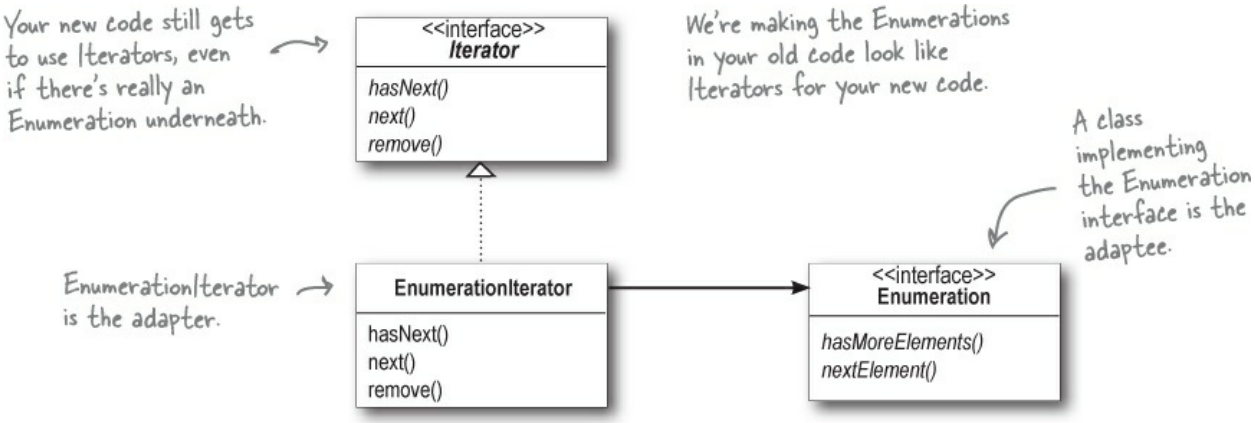
Adapting an Enumeration to an Iterator

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 Enumerations's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.

EXERCISE

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

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

And now for something different...

There's another pattern in this chapter.

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

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

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

WHO DOES WHAT?

Match each pattern with its intent:

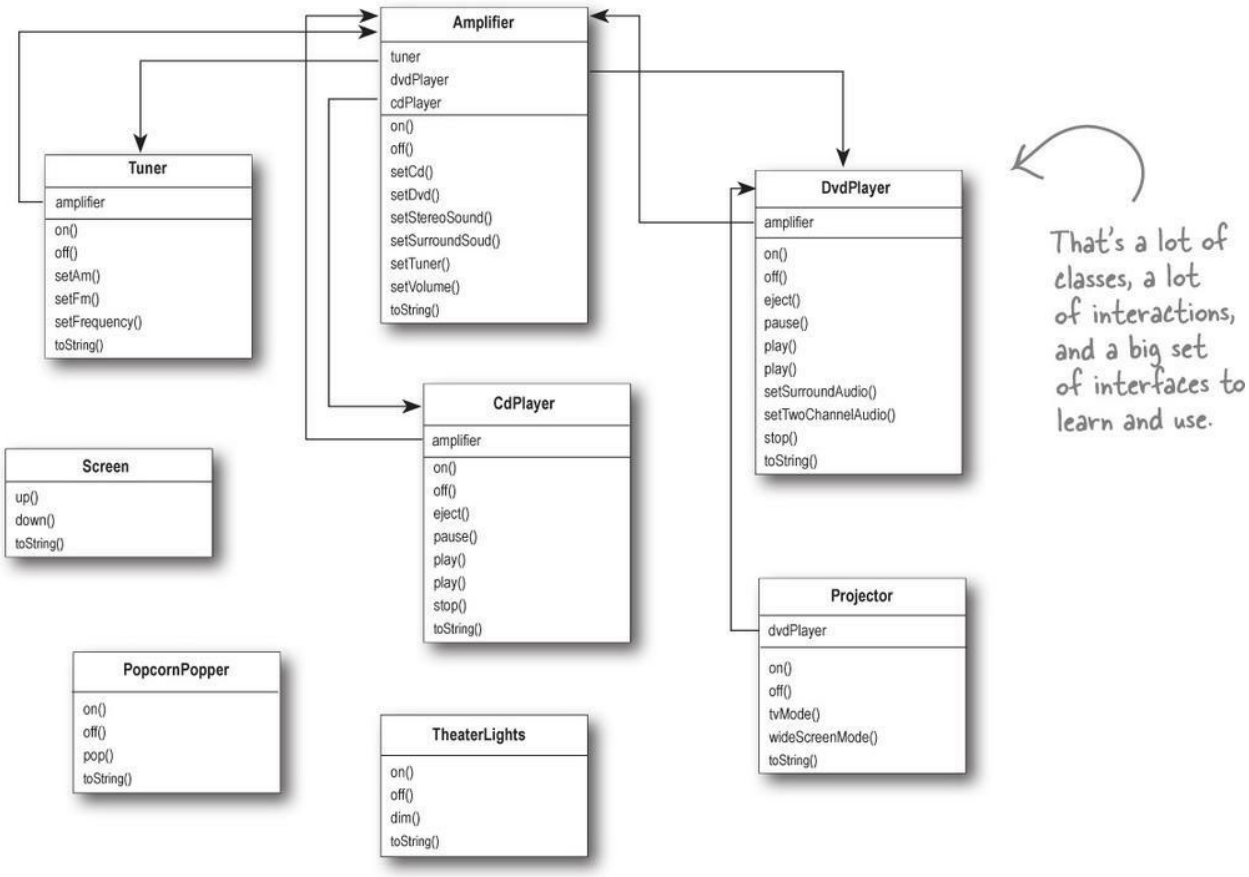
Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

Home Sweet Home Theater

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

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

Check out all the components you've put together:



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...

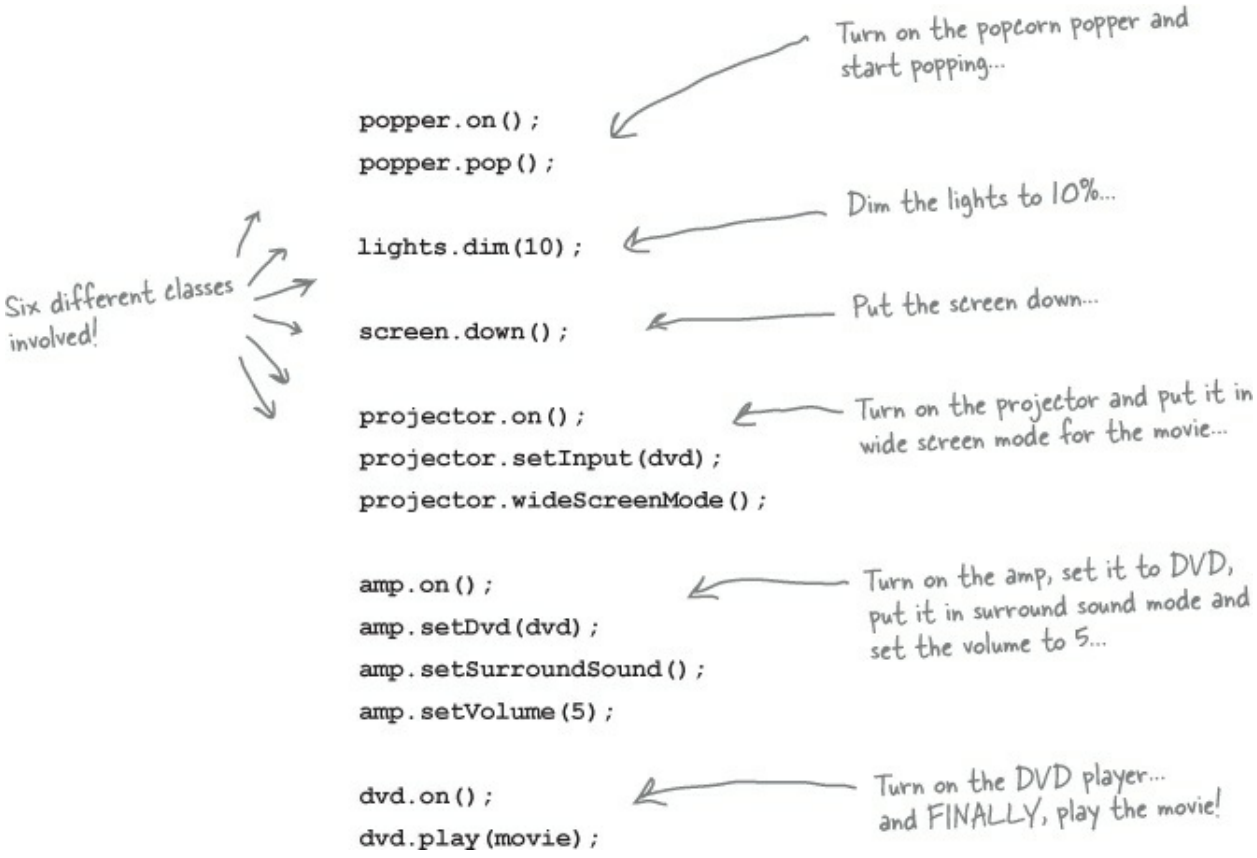
Watching a movie (the hard way)

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:



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.

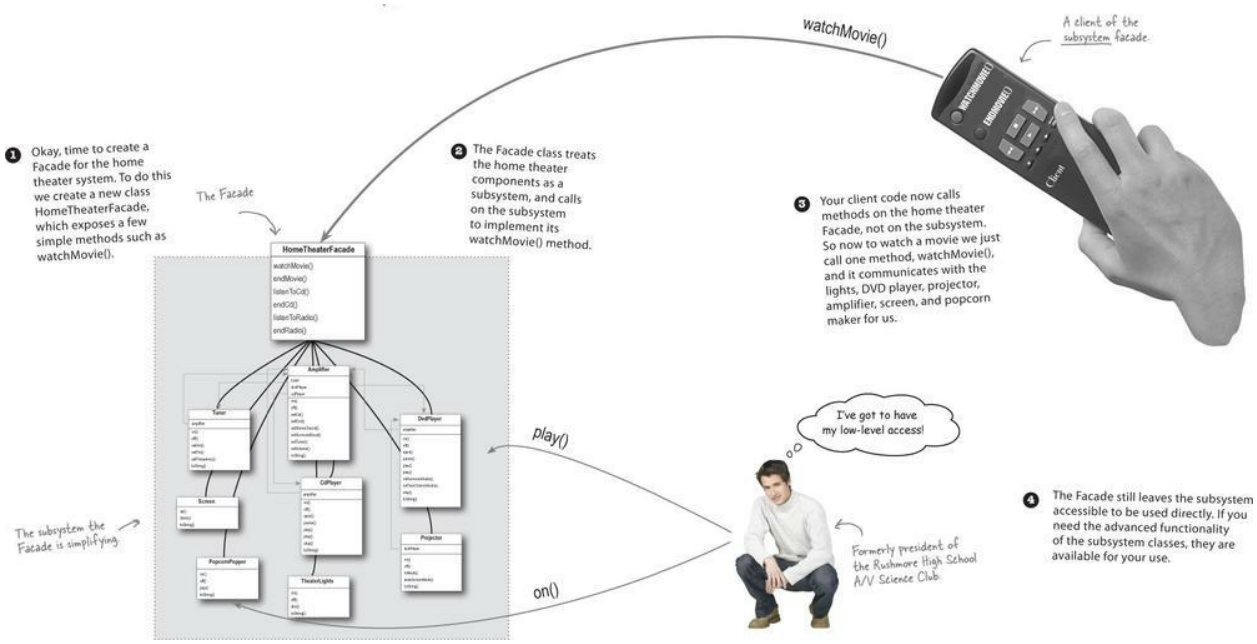
So what to do? The complexity of using your home theater is becoming apparent!

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 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;

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

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

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

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementing the simplified interface

Now it's time to bring the components of the subsystem together into a unified interface. Let's implement the watchMovie() and endMovie() methods:

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

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

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

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

BRAIN POWER

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

Time to watch a movie (the easy way)

It's SHOWTIME!

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

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

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

}
```

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

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

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

Here's the output.

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

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

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

Facade Pattern defined

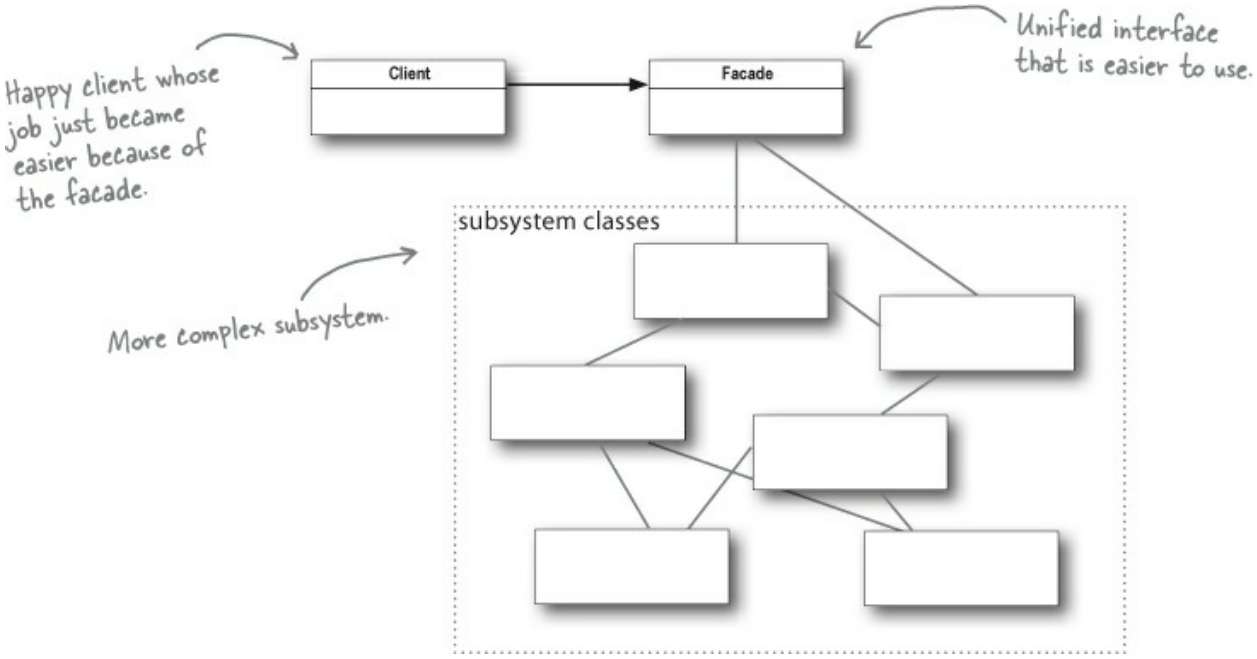
To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind-bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between

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

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

> **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.

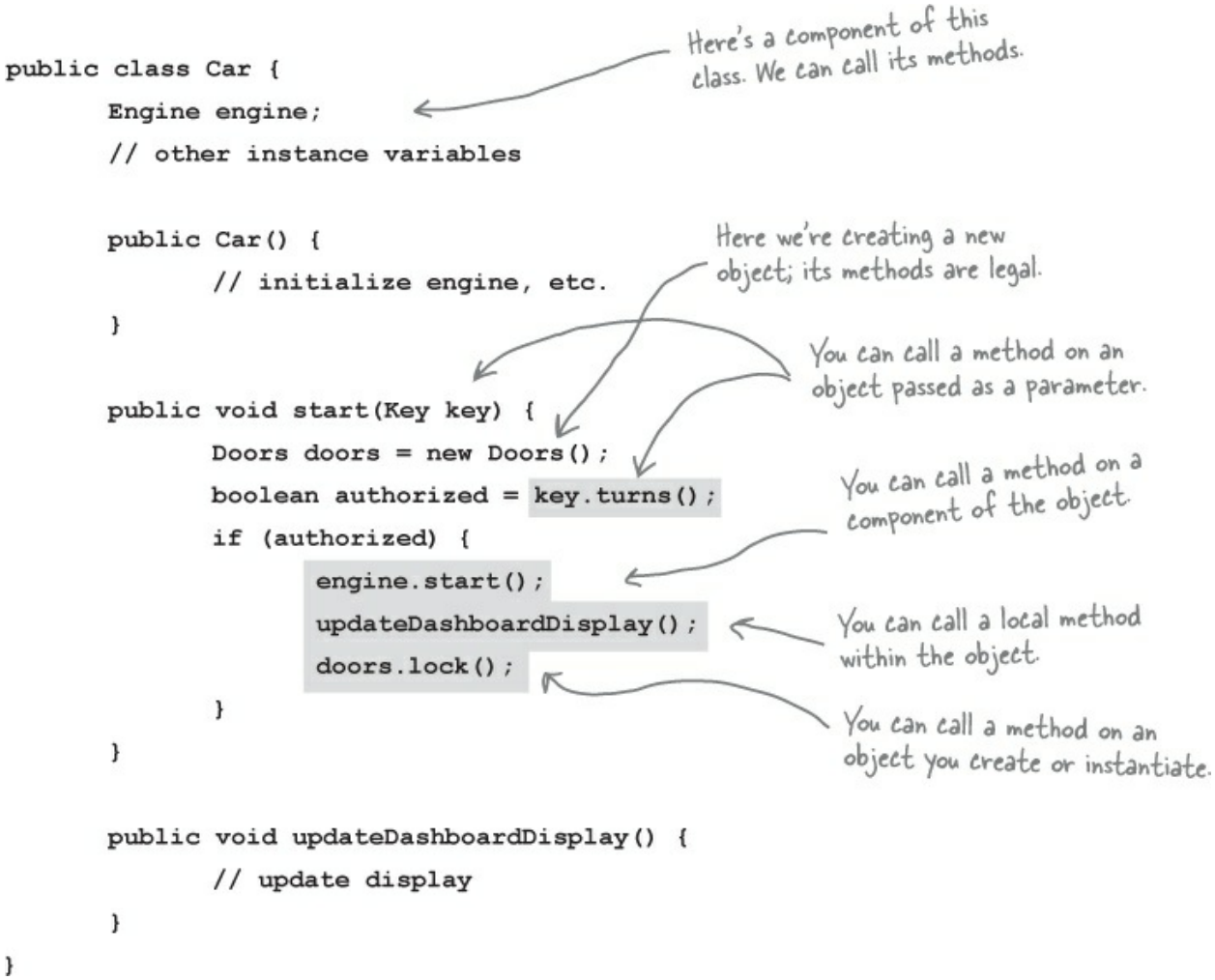
With the Principle

```
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
        }
}
```

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();
    }
}
```

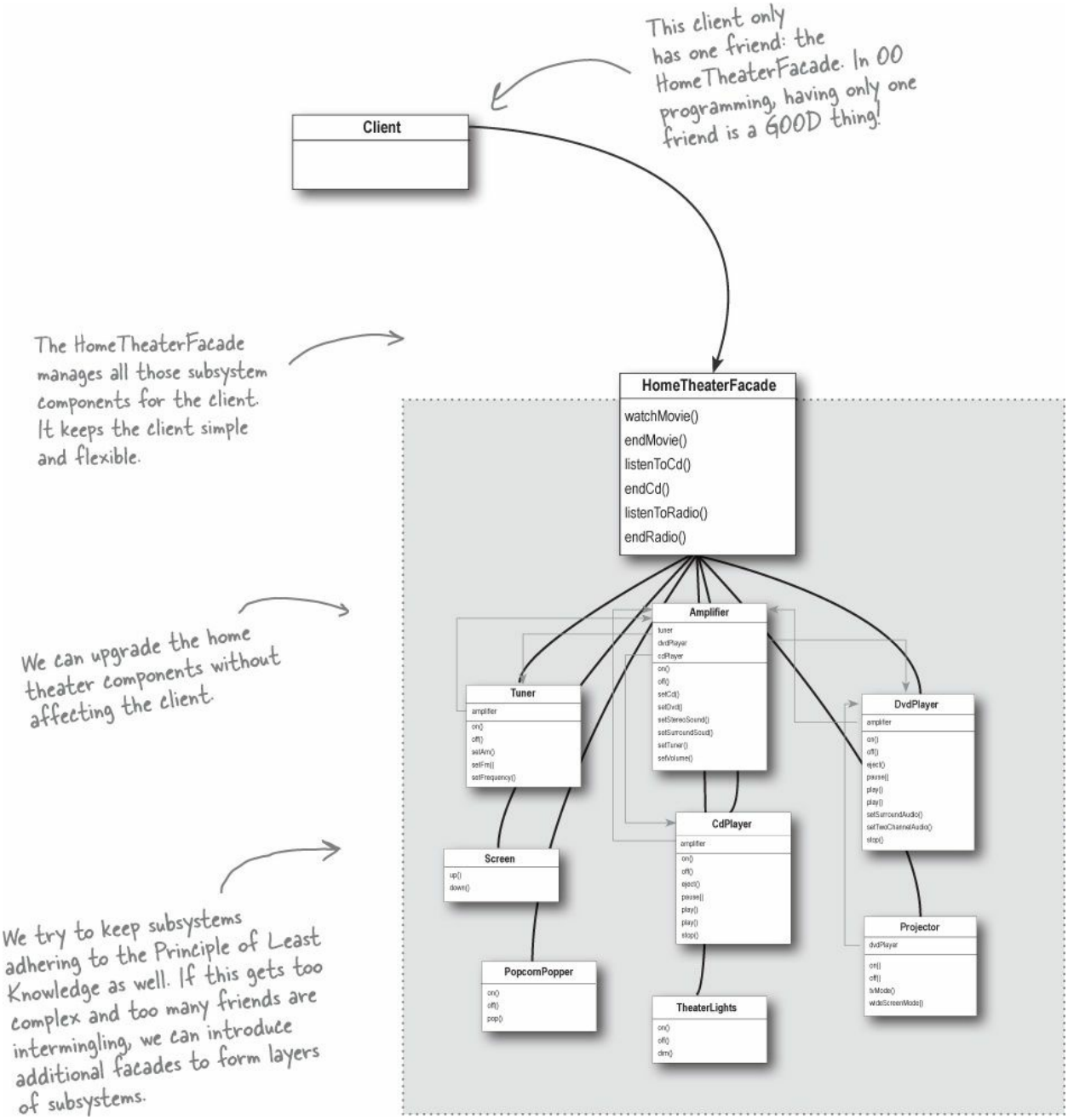
HARD HAT AREA.
WATCH OUT FOR
FALLING ASSUMPTIONS

BRAIN POWER

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

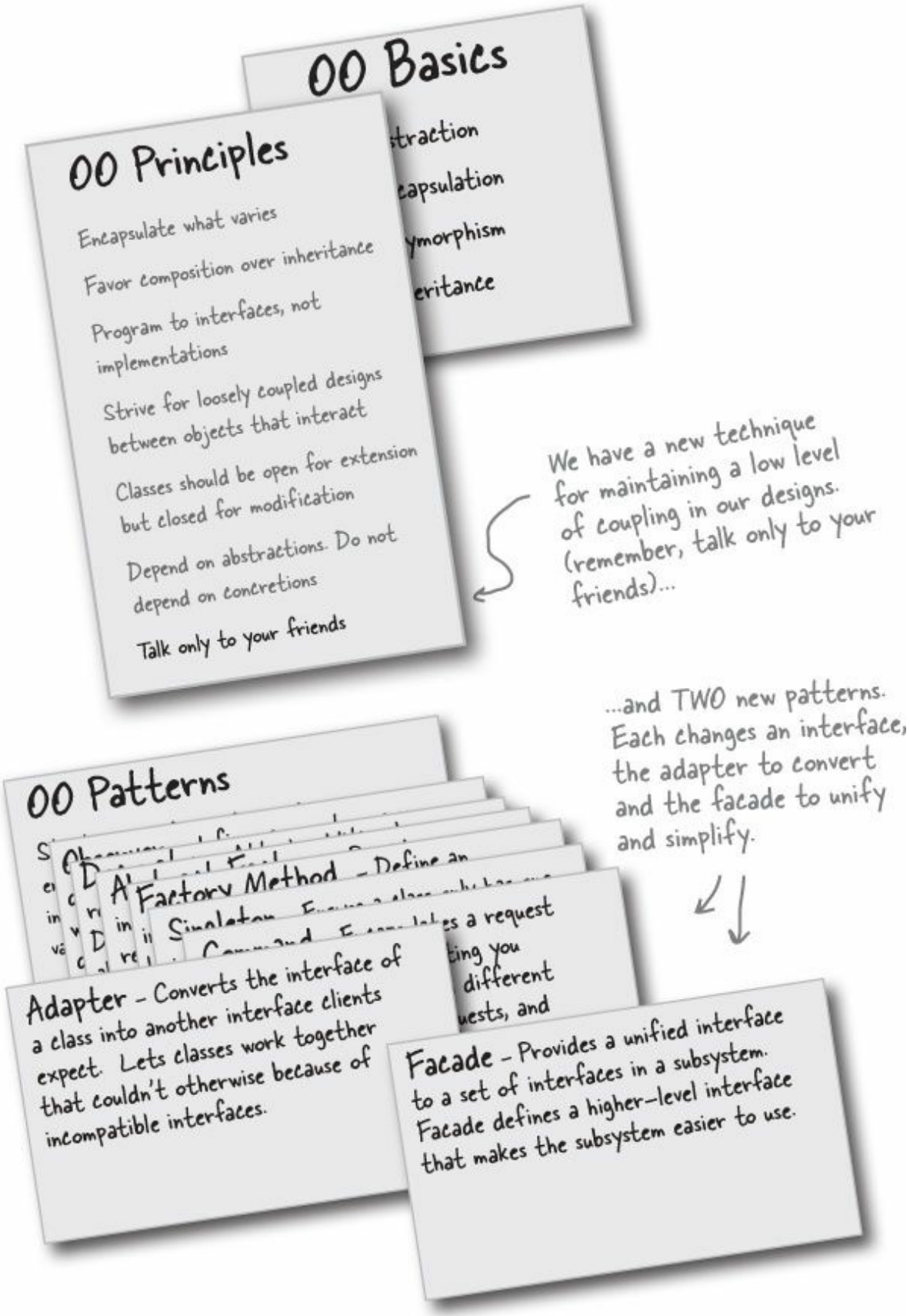
A: Answer: How about System.out.println()?

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.

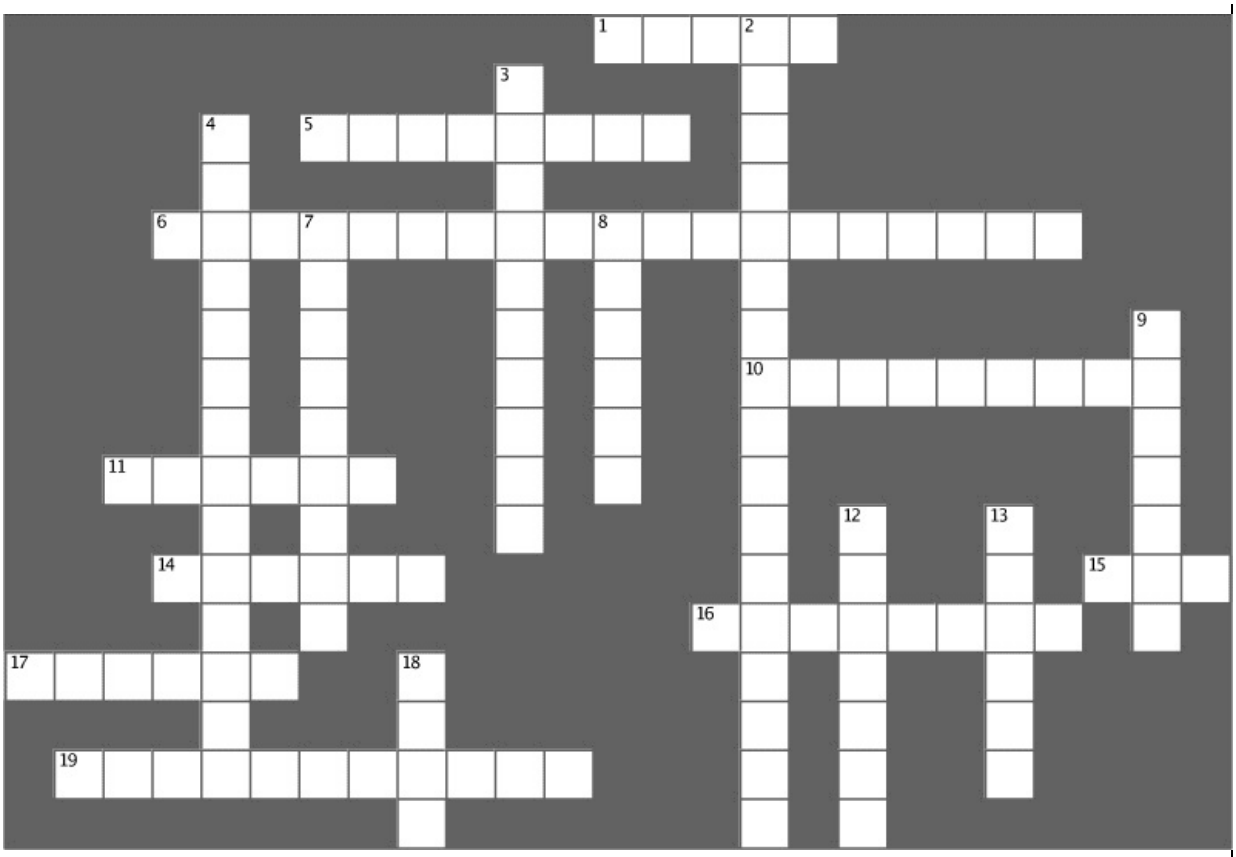

OO Basics
traction
apsulation
ymorphism
eritance
OO Principles
Encapsulate what varies
Favor composition over inheritance
Program to interfaces, not implementations
Strive for loosely coupled designs between objects that interact
Classes should be open for extension but closed for modification
Depend on abstractions. Do not depend on concretions
Talk only to your friends
We have a new technique for maintaining a low level of coupling in our designs. (remember, talk only to your friends)...
...and TWO new patterns. Each changes an interface, the adapter to convert and the facade to unify and simplify.
OO Patterns
Factory Method – Define an
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.

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

SHARPEN YOUR PENCIL SOLUTION

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();
    }
}
```

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

```
public House {
    WeatherStation station;
    // other methods and constructor
    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

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

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

EXERCISE SOLUTION

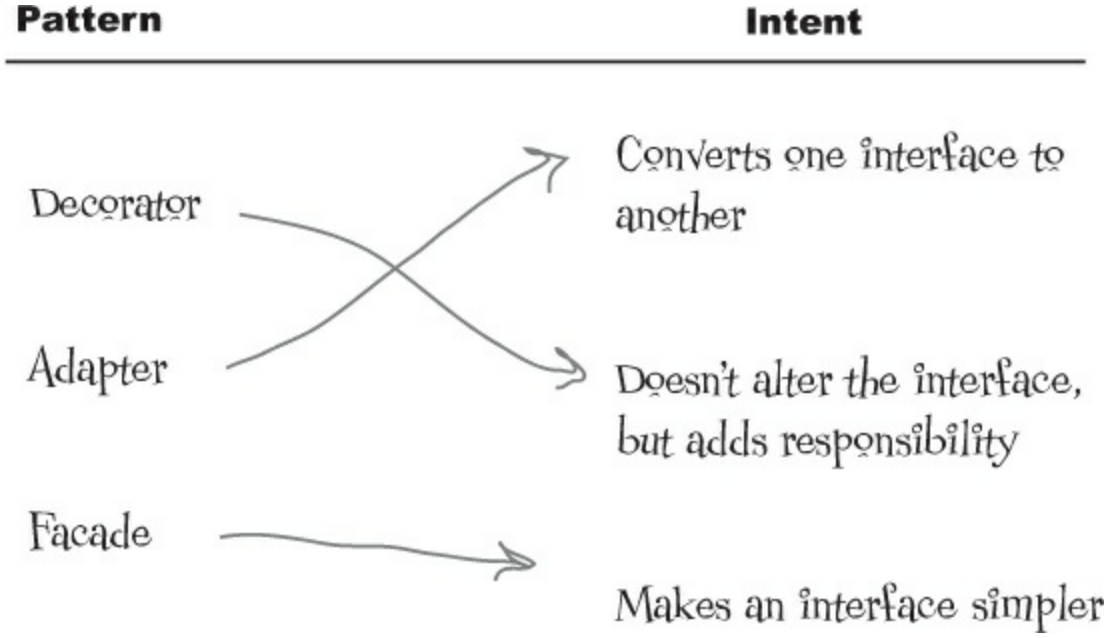
You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumeration.

```
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();
    }
}
```

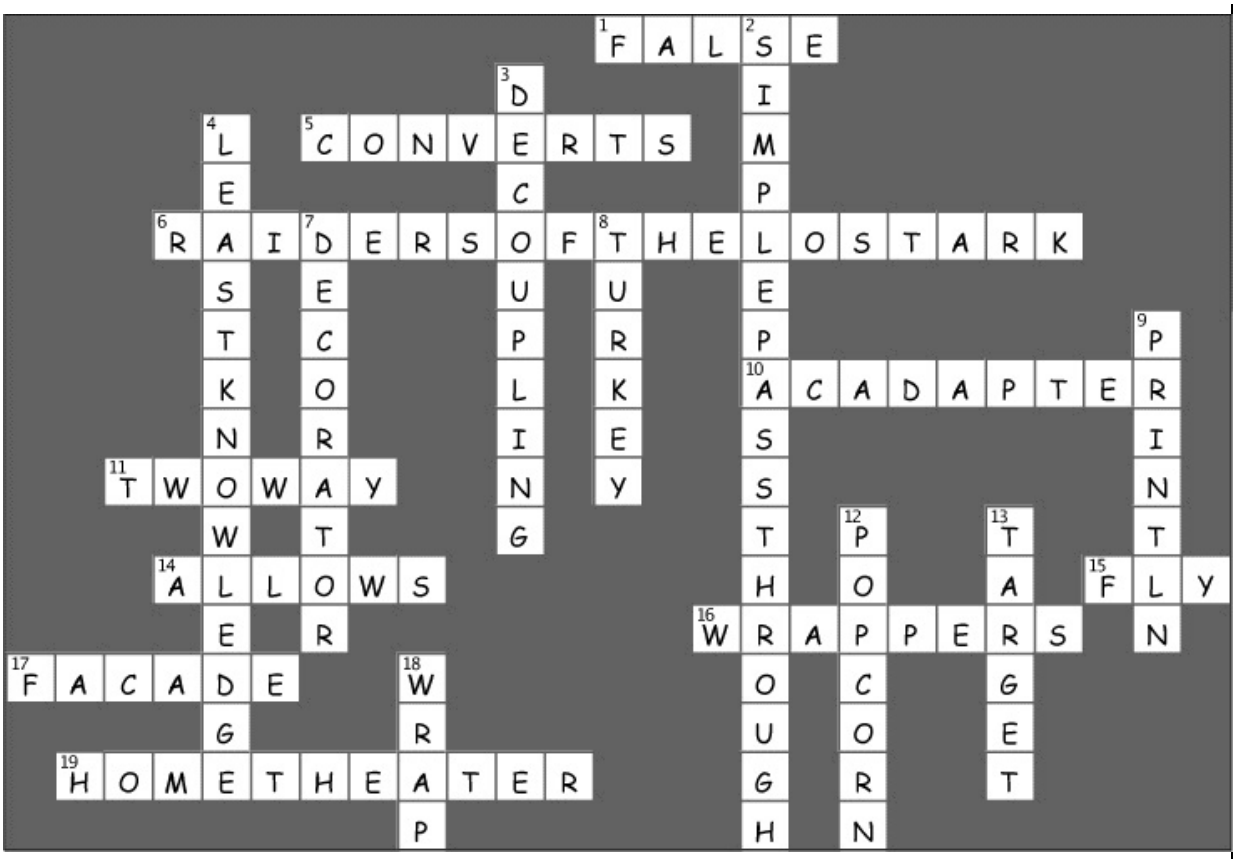
Notice we keep the type parameter generic so this will work for any type of object.

WHO DOES WHAT? SOLUTION

Match each pattern with its intent:



DESIGN PATTERNS CROSSWORD SOLUTION


1 FALSE
3 D I
4 L 5 CONVERTS M
E C P
6 RAIDERS 7 OF 8 THE LOSTARK
S E U U E
T C P R P 9 P
K O L K 10 ACADAPTER
N R I E S I
11 TWOWAY N Y S N
W T G T 12 P 13 T T
14 ALLOWS H O A 15 FLY
E R 16 WRAPPERS N
17 FACADE 18 W O C G
G R U O E
19 HOMETHEATER G R T
P H N

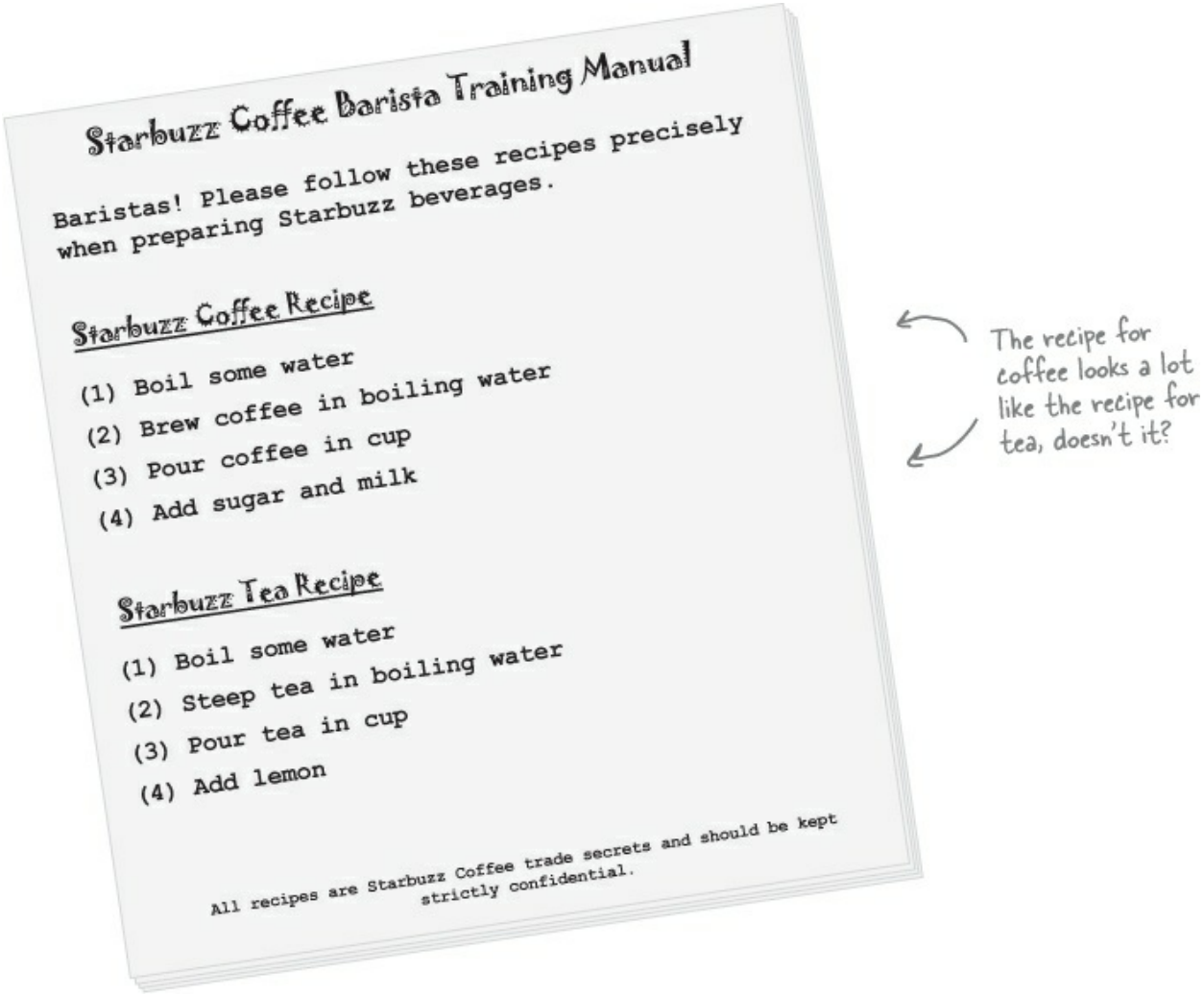
Chapter 8. The Template Method Pattern: Encapsulating Algorithms

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:



Whipping up some coffee and tea classes (in Java)

Let’s play “coding barista” and write some code for creating coffee and tea.

Here’s the coffee:

Here's our Coffee class for making coffee.

```
public class Coffee {

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

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

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

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

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

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

Each of the steps is implemented as a separate method.

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

And now the Tea...

```
public class Tea {

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

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

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

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

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

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

These two methods are specialized to Tea.

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

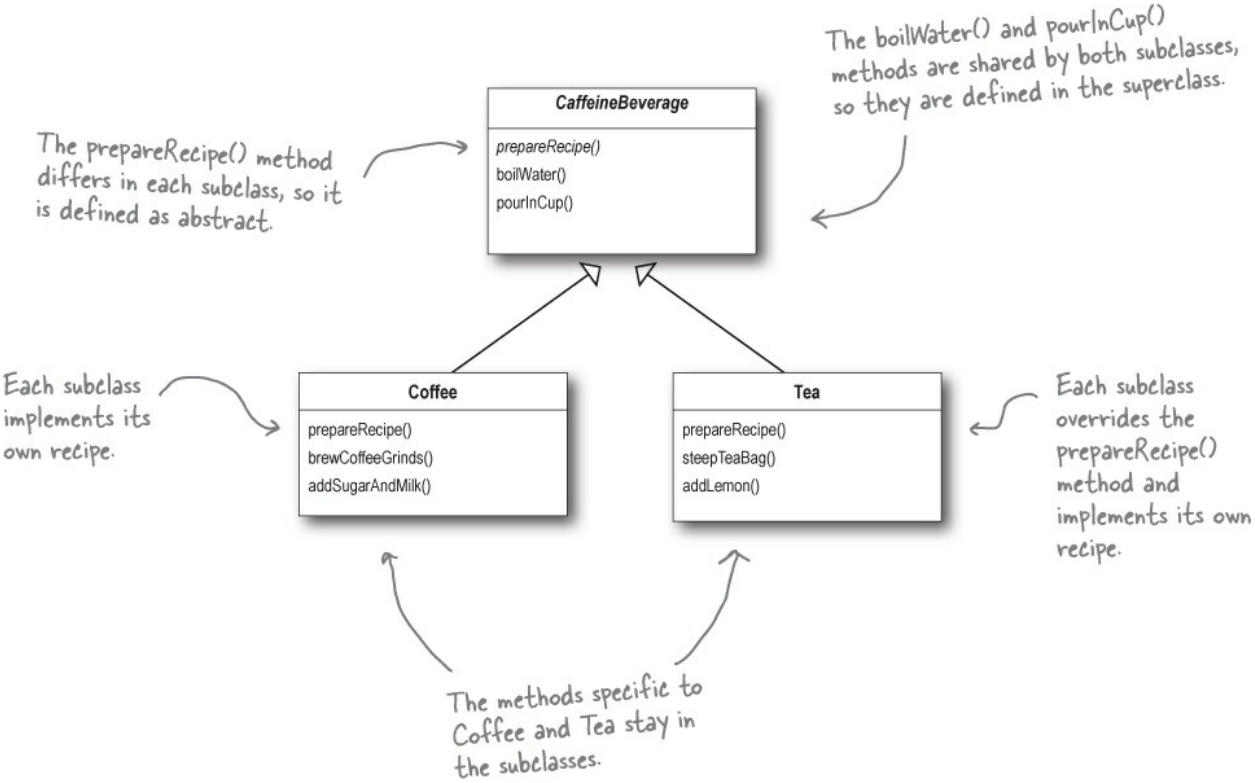
When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

DESIGN PUZZLE

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

Sir, may I abstract your Coffee, Tea?

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

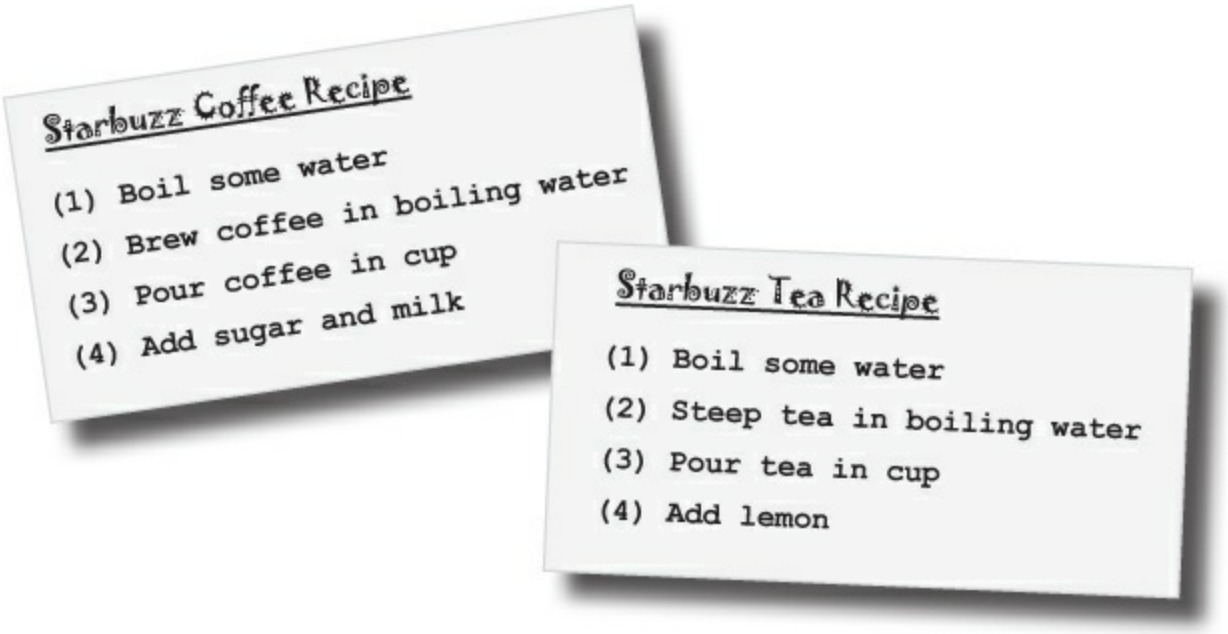


BRAIN POWER

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.**

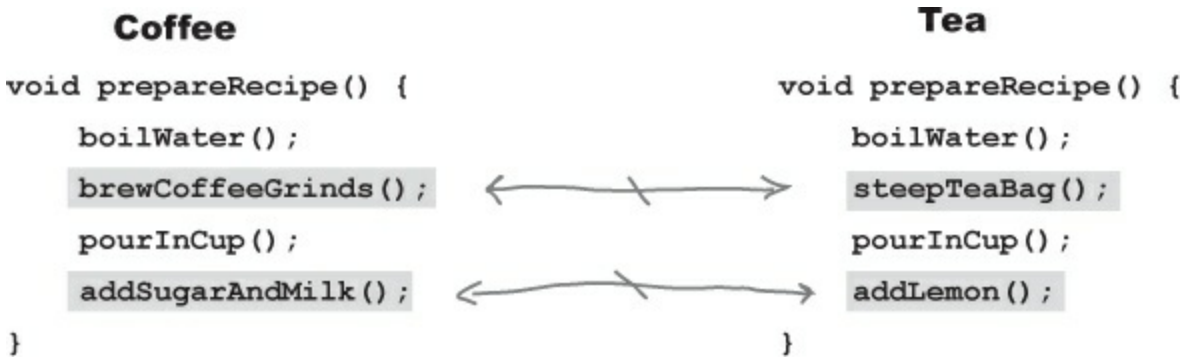
So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

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.



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.

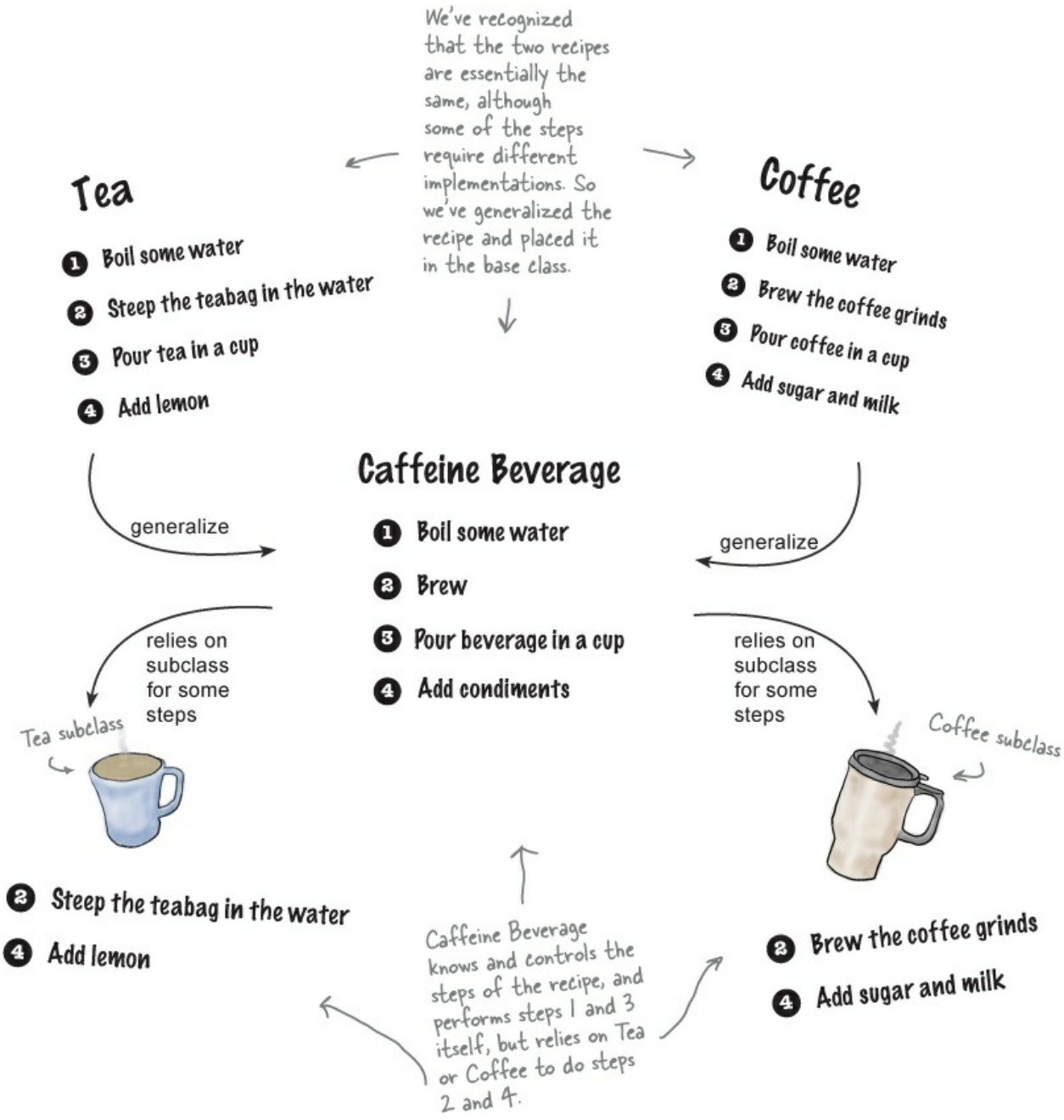
Tea needs to define brew() and addCondiments()—the two abstract methods from CaffeineBeverage.

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

SHARPEN YOUR PENCIL

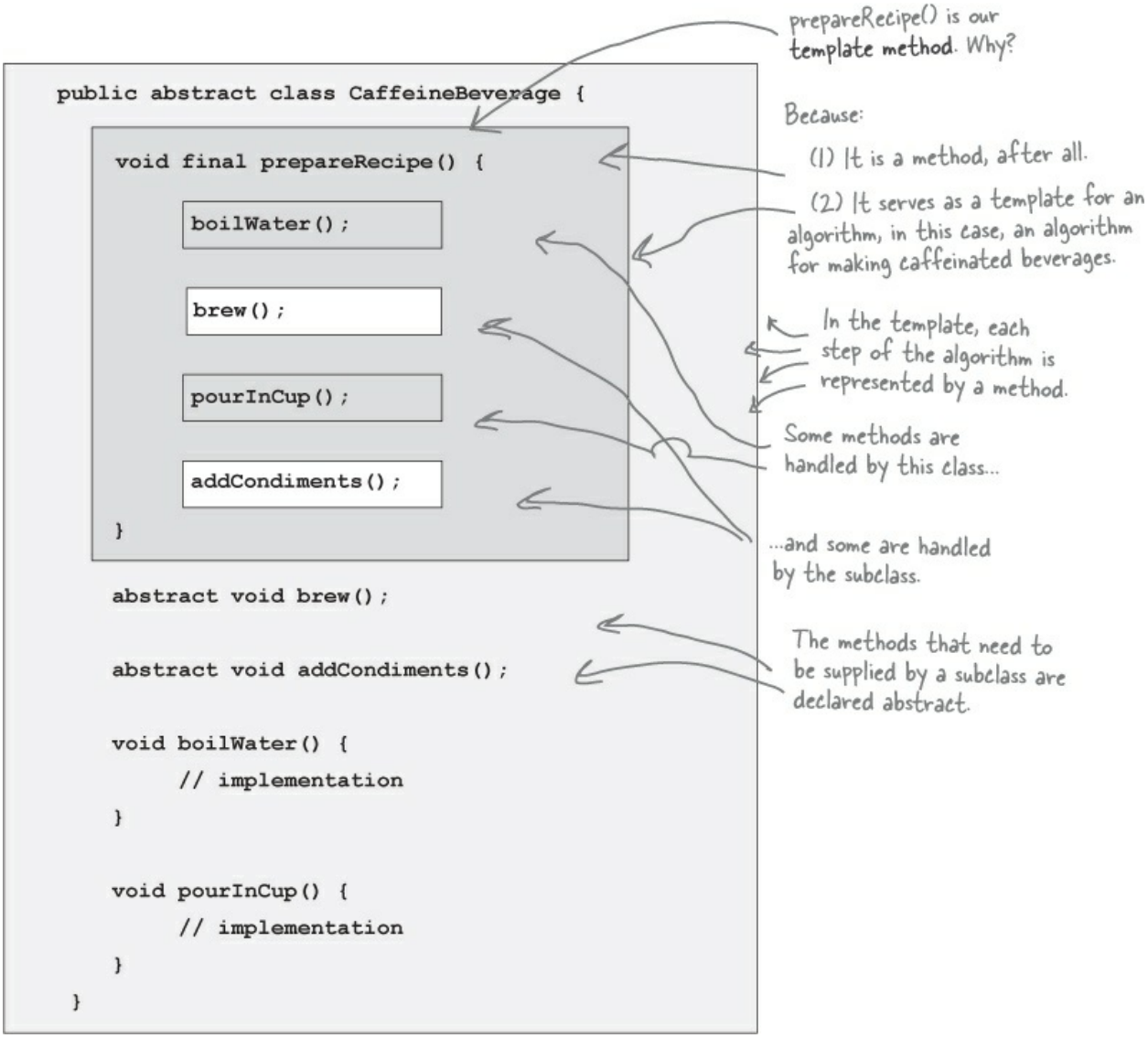
Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

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...

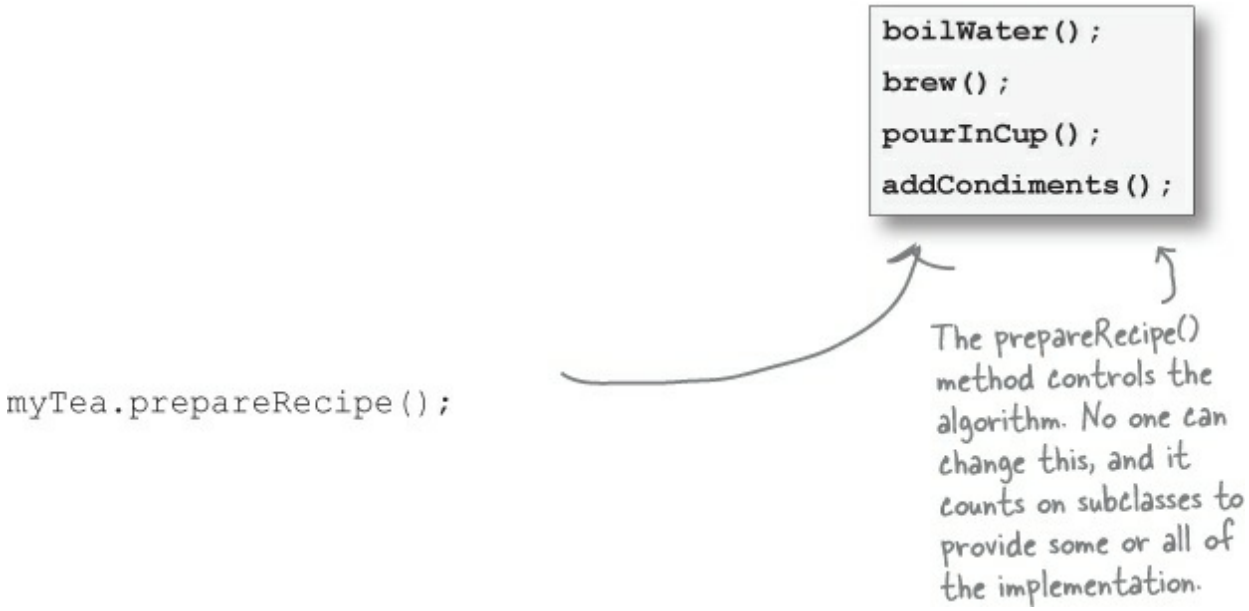
Behind the Scenes

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...

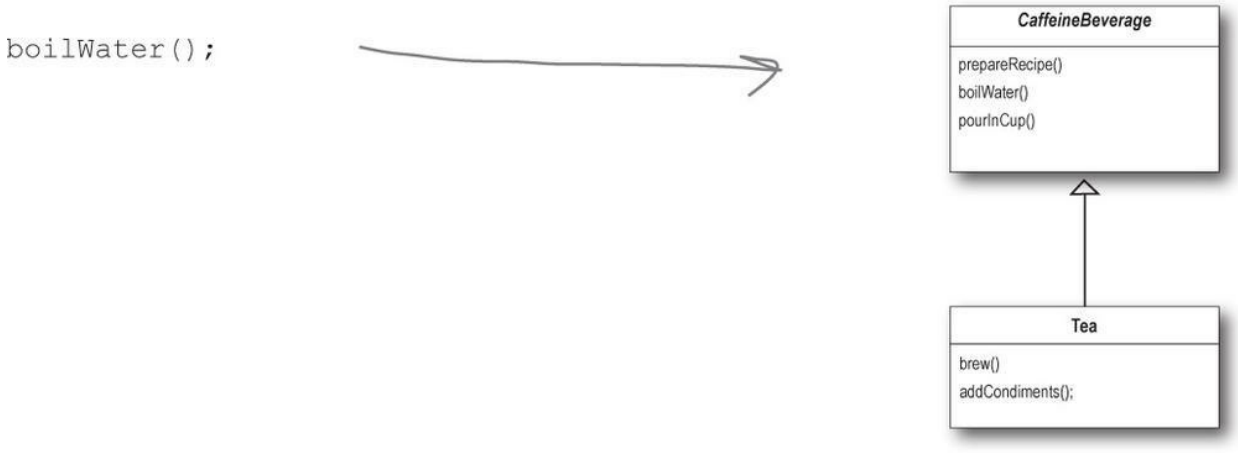
```
Tea myTea = new Tea();
```

② Then we call the template method:



which follows the algorithm for making caffeine beverages...

③ First we boil water:



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

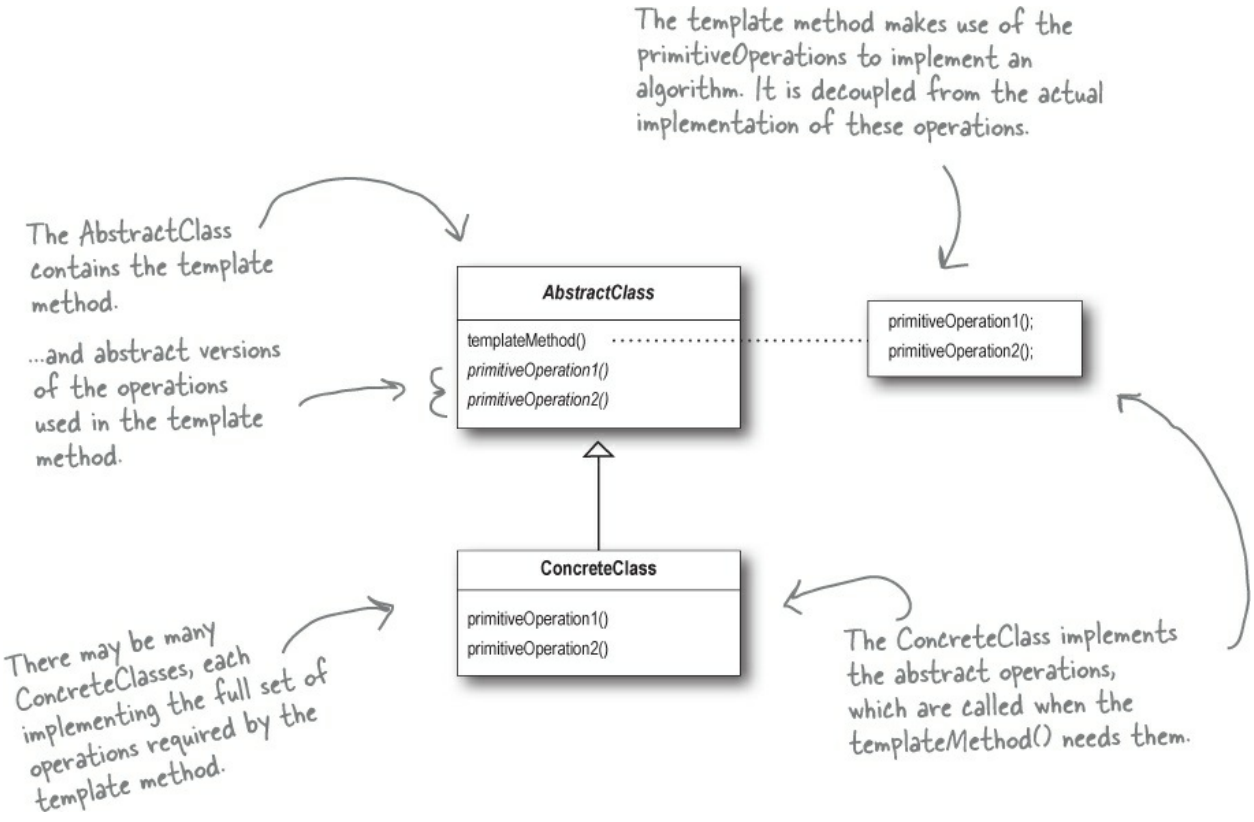
You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

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.

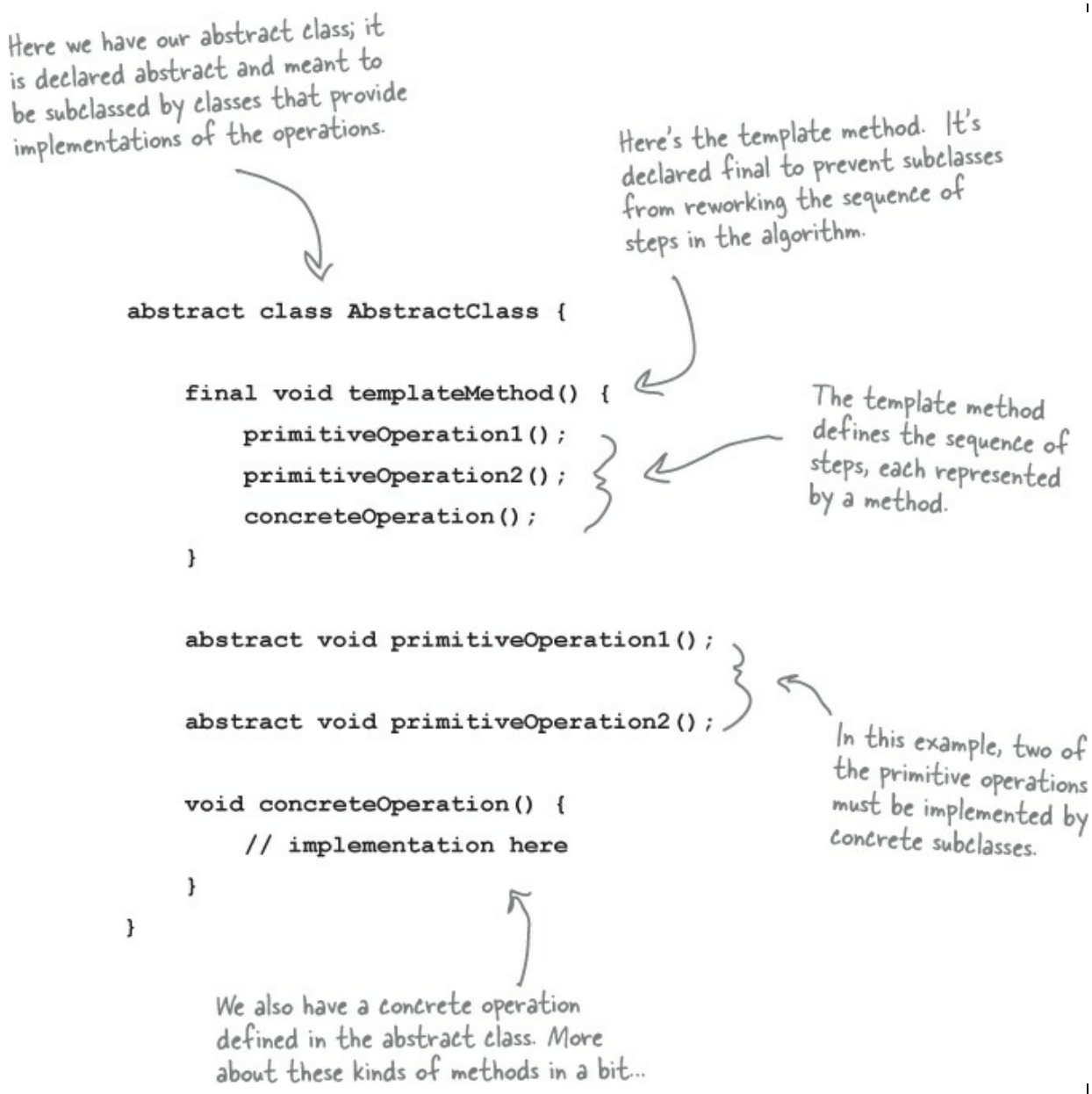
This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's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

Let's check out the class diagram:



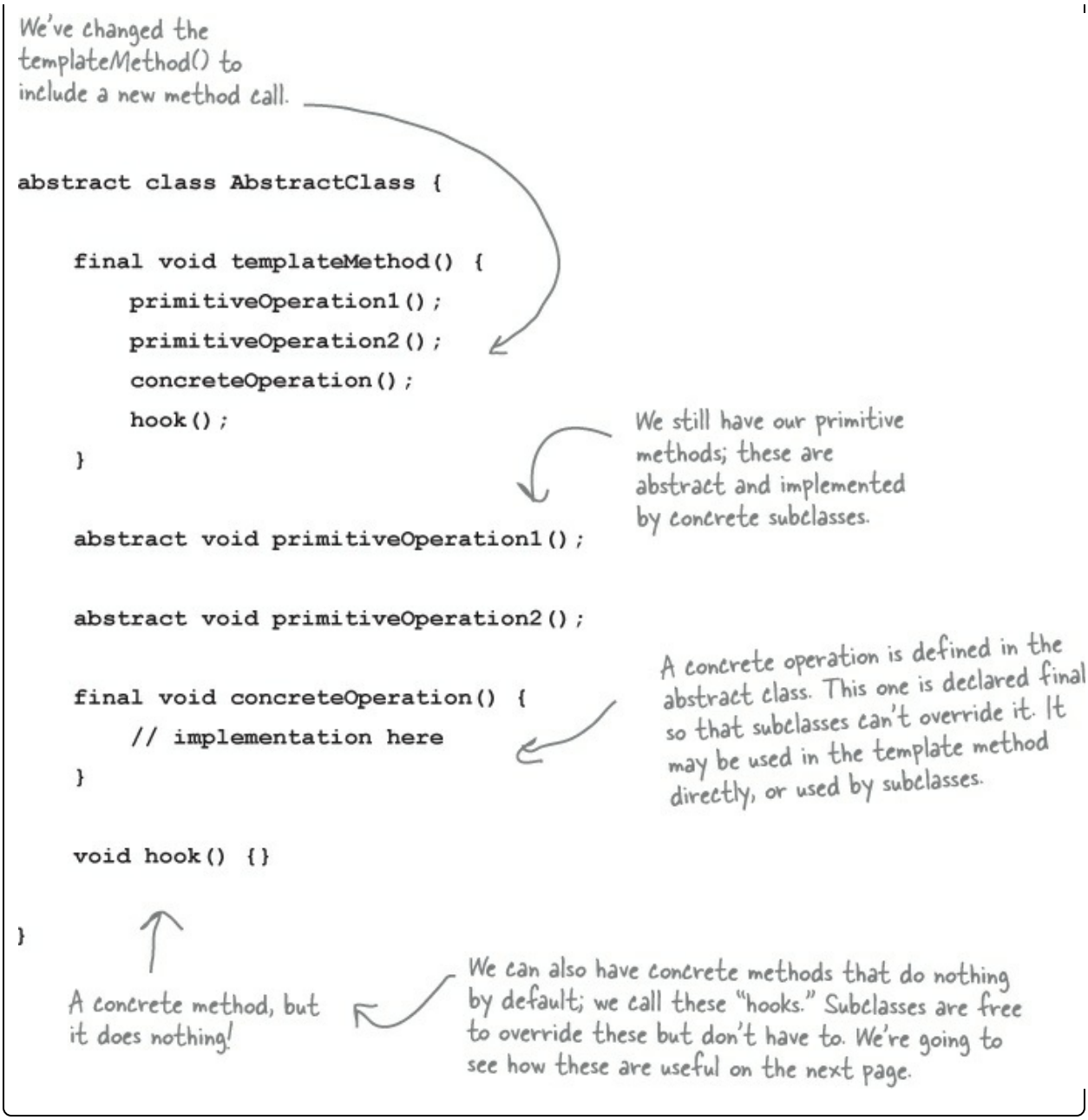
CODE UP CLOSE

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



CODE WAY UP CLOSE

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



Hooked on Template Method...

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

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

```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

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

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

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

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

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

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

Let's run the Test Drive

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

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

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

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

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

Create a tea.

A coffee.

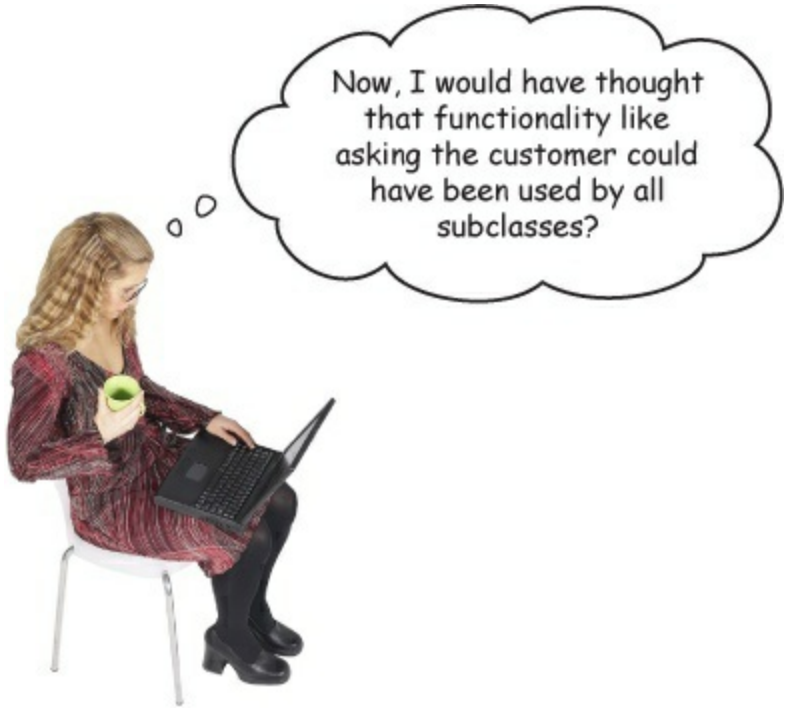
And call prepareRecipe() on both!

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

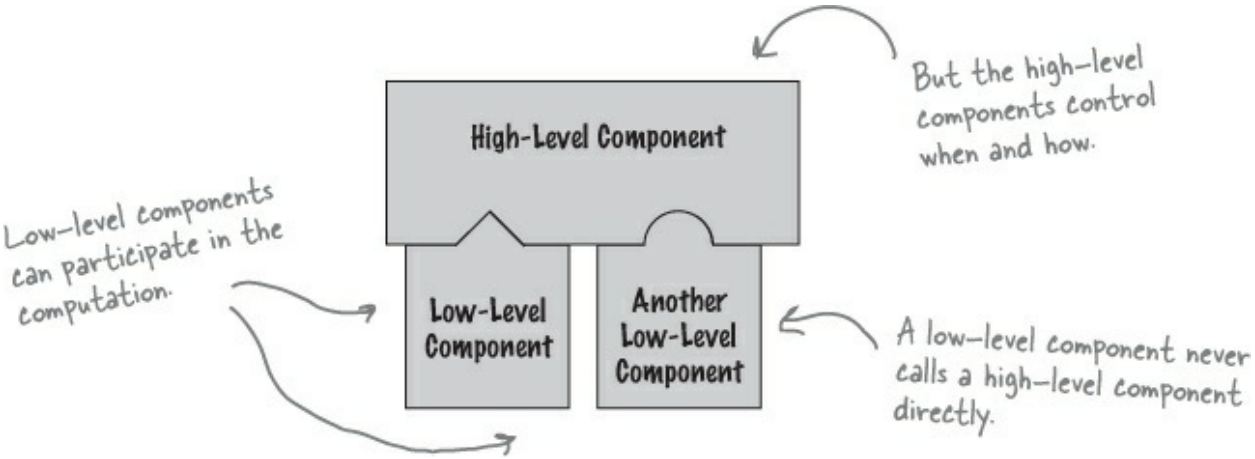
Don't call us, we'll call you.

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

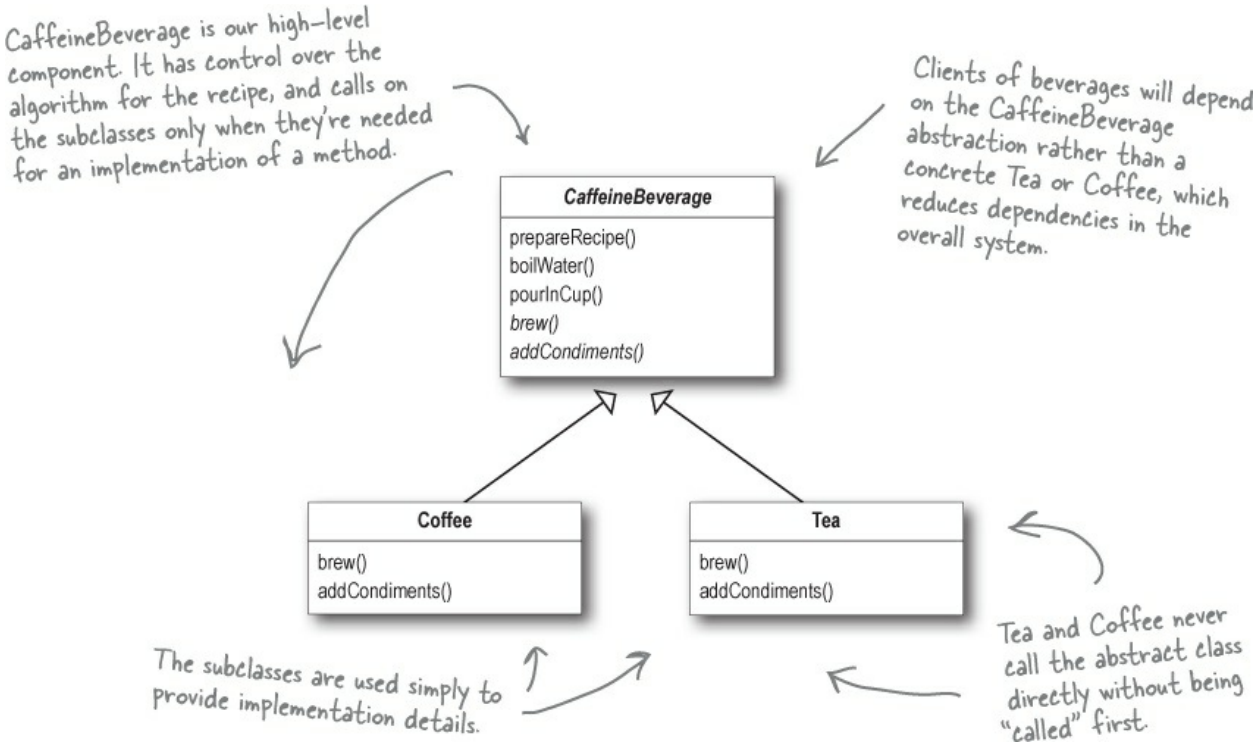
sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

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:



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:

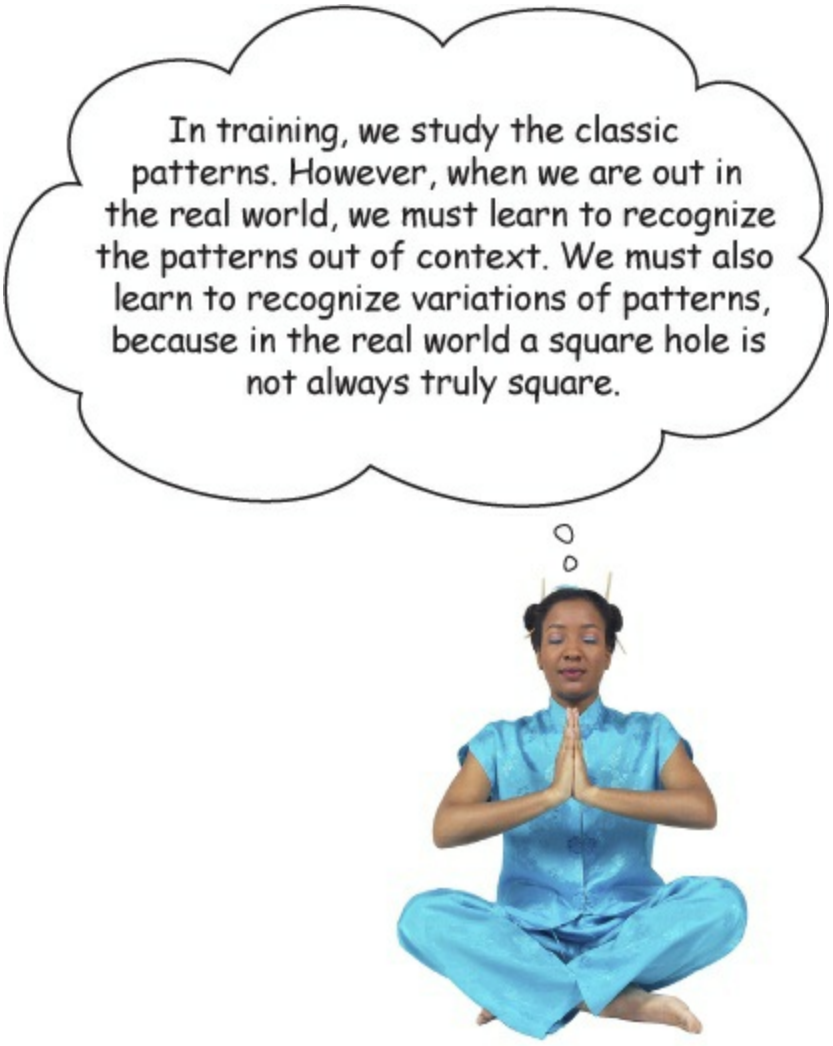
Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Factory Method	Subclasses decide which concrete classes to instantiate.

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)...



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?

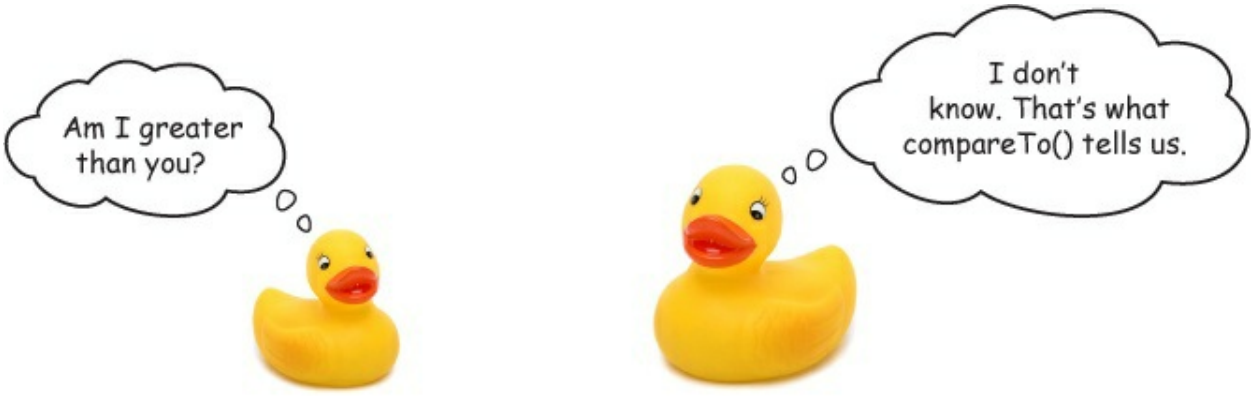
Good point. Here's the deal: the designers of sort() wanted it to be useful across all arrays, so they had to make sort() a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because sort() really isn't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

Remember, we need to implement the Comparable interface since we aren't really subclassing.

```
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;
        }
    }
}
```

Our Ducks have a name and a weight

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Let's sort some Ducks

Here's the test drive for sorting Ducks...

```
public class DuckSortTestDrive {

    public static void main(String[] args) {
        Duck[] ducks = {
                        new Duck("Daffy", 8),
                        new Duck("Dewey", 2),
                        new Duck("Howard", 7),
                        new Duck("Louie", 2),
                        new Duck("Donald", 10),
                        new Duck("Huey", 2)
         };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (Duck d : ducks) {
            System.out.println(d);
        }
    }
}
```

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

It's sort time!

Let's print them (again) to see their names and weights.

Notice that we call Arrays' static method sort, and pass it our Ducks.

Let the sorting commence!

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive
Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

The unsorted Ducks

The sorted Ducks

The making of the sorting duck machine

Behind the Scenes

① First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

② Then we call the sort() template method in the Array class and pass it our ducks:

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

```
Arrays.sort(ducks);
```

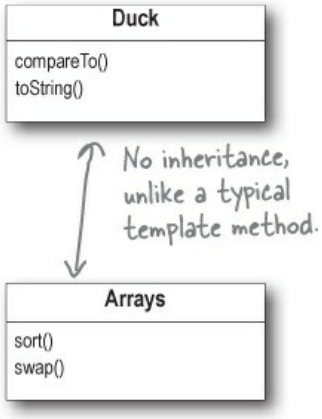
The sort() method controls the algorithm; no class can change this. sort() counts on a Comparable class to provide the implementation of compareTo().

The sort() method (and its helper mergeSort()) control the sort procedure.
③ To sort an array, you need to compare two items one by one until the entire list is in sorted order.
When it comes to comparing two ducks, the sort method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to:

```
ducks[0].compareTo(ducks[1]);
```

First Duck

Duck to compare it to



④ If the Ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays:

```
swap()
```

⑤ The sort() method continues comparing and swapping Ducks until the array is in the correct order!

THERE ARE NO DUMB QUESTIONS

Q:	**Q: Is this really the Template Method Pattern, or are you trying too hard?**
A:	**A:** The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps — and the Arrays sort is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints. The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted. So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.
Q:	**Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?**
A:	**A:** You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way — we're using the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the entire algorithm. The algorithm that Arrays implements for sort is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.
Q:	**Q: Are there other examples of template methods in the Java API?**
A:	**A:** Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the template method read(byte b[], int off, int len).

BRAIN POWER

We know that we should favor composition over inheritance, right? Well, the implementers of the sort() template method decided not to use inheritance and instead to implement sort() as a static method that is composed with a Comparable at runtime. How is this better? How is it worse? How would you approach this problem? Do Java arrays make this particularly tricky?

BRAIN^2 POWER

Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

Swingin' with Frames

Up next on our Template Method safari... keep your eye out for swinging JFrames!

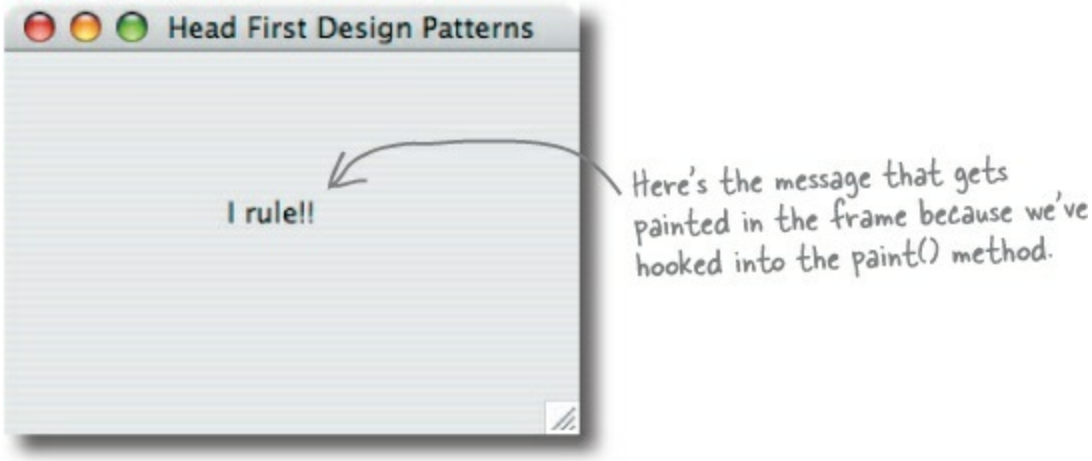
If you haven't encountered JFrame, it's the most basic Swing container and inherits a paint() method. By default, paint() does nothing because it's a hook! By overriding paint(), you can insert yourself into JFrame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the JFrame. Here's an embarrassingly simple example of using a JFrame to override the paint() hook method:

We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```

Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls paint(). By default, paint() does nothing... it's a hook. We're overriding paint(), and telling the JFrame to draw a message in the window.



Applets

Our final stop on the safari: the applet.

You probably know an applet is a small program that runs in a web page. Any applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them:

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet is going away...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint() is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

If the user goes to another page, the stop hook is used, and the applet can do whatever it needs to do to stop its actions.

And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

Well, looky here! Our old friend the paint() method! Applet also makes use of this method as a hook.

Concrete applets make extensive use of hooks to supply their own behaviors. Because these methods are implemented as hooks, the applet isn't required to implement them.

FIRESIDE CHATS

Tonight's talk: **Template Method and Strategy compare methods.**

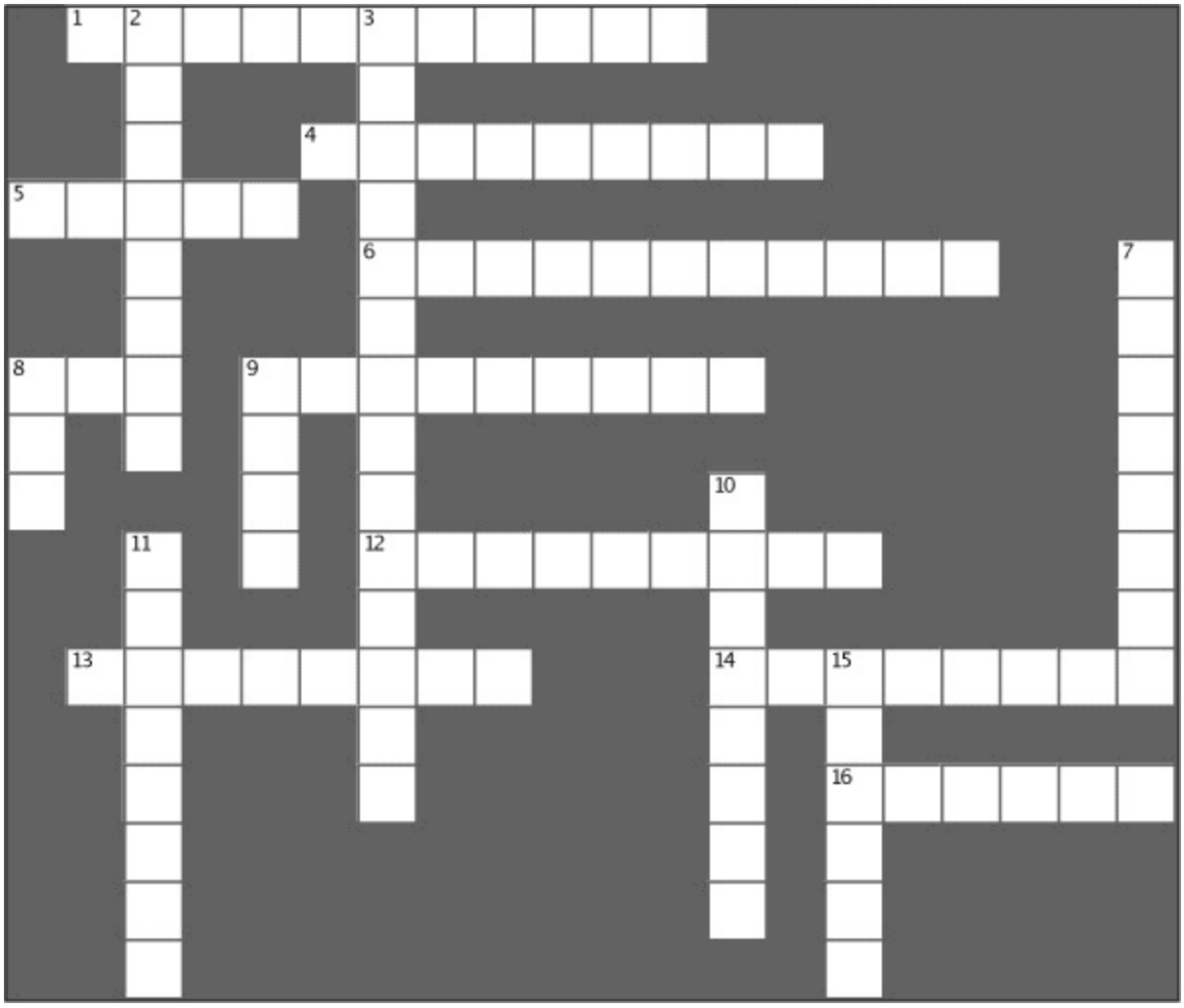
Template Method:	Strategy:
Hey Strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method.	
	Nope, it's me, although be careful — you and Factory Method are related, aren't you?
I was just kidding! But seriously, what are you	

doing here? We haven't heard from you in eight chapters!	
	I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...
You might want to remind the reader what you're all about, since it's been so long.	
	I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern...?" So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.
Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.	
	I'm not sure I'd put it quite like *that*... and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.
I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.	
	You might be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose me for Chapter 1 for nothing!

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.	
	Yeah, I guess... but, what about dependency? You're way more dependent than me.
How's that? My superclass is abstract.	
	But you have to depend on methods implemented in your subclasses, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!
Like I said, Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.	
	Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway; I'm always glad to help.
Got it. Don't call us, we'll call you...	

DESIGN PATTERNS CROSSWORD

It's that time again....



Across	Down
1. Strategy uses _____ rather than inheritance. 4. Type of sort used in Arrays. 5. The JFrame hook method that we overrode to print “I Rule”. 6. The Template Method Pattern uses _____ to defer implementation to other classes. 8. Coffee and _____. 9. “Don’t call us, we’ll call you” is known as the _____ Principle. 12. A template method defines the steps of an _____.	2. _____ algorithm steps are implemented by hook methods. 3. Factory Method is a _____ of Template Method. 7. The steps in the algorithm that must be supplied by the subclasses are usually declared _____. 8. Huey, Louie, and Dewey all weigh _____ pounds. 9. A method in the abstract superclass that does nothing or provides default behavior is called a _____ method. 10. Big-headed pattern. 11. Our favorite coffee shop in Objectville.

13. In this chapter, we give you more _____. 14. The template method is usually defined in an _____ class. 16. Class that likes web pages.	15. The Arrays class implements its template method as a _____ method.

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method you can reuse code like a pro while keeping control of your algorithms.

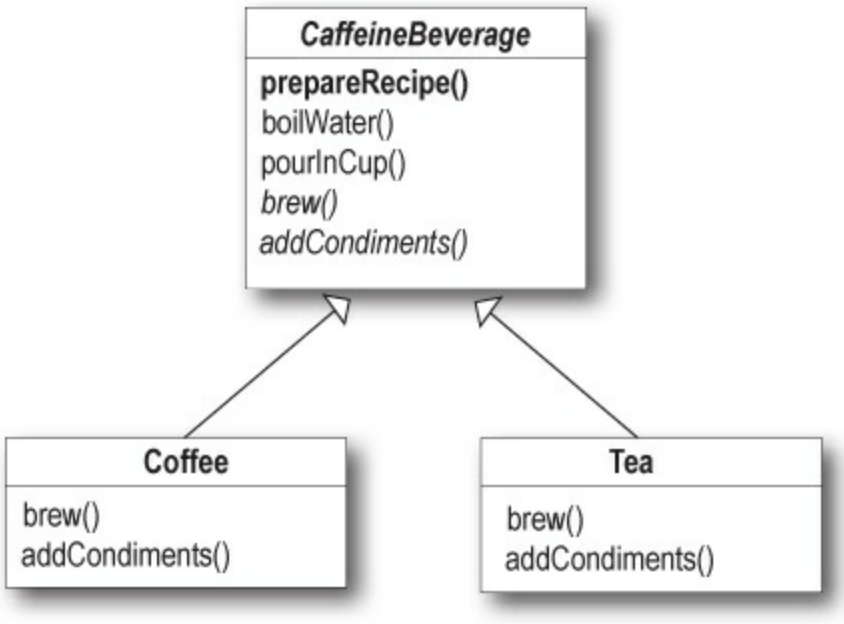

OO Basics
bstraction
ncapsulation
olymorphism
heritance
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
Our newest principle reminds you that your superclasses are running the show, so let them call your subclasses when they're needed, just like they do in Hollywood.
OO Patterns
Factory Method – Define an
Adapter – Encapsulates a request
Facade – Encapsulates a request
Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

BULLET POINTS

- A “template method” defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method’s abstract class may define concrete methods, abstract methods, and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- You’ll see lots of uses of the Template Method Pattern in real-world code, but don’t expect it all (like any pattern) to be designed “by the book.”
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is a specialization of Template Method.

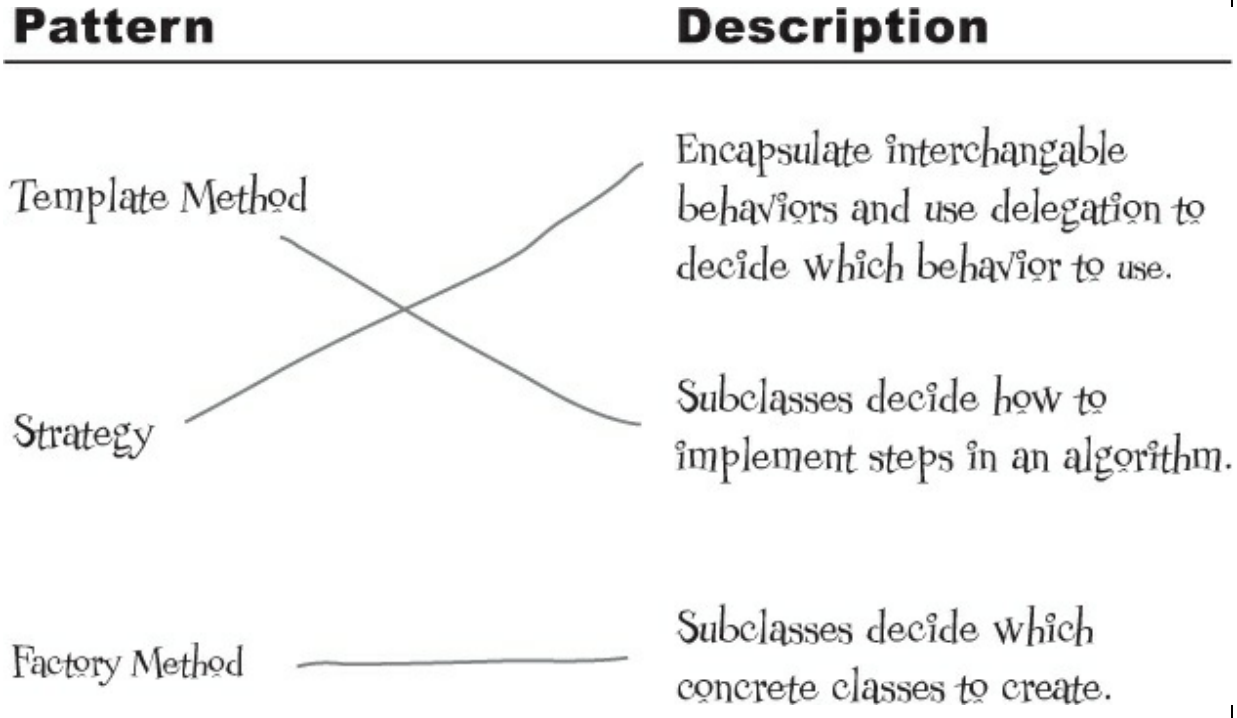
SHARPEN YOUR PENCIL SOLUTION

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



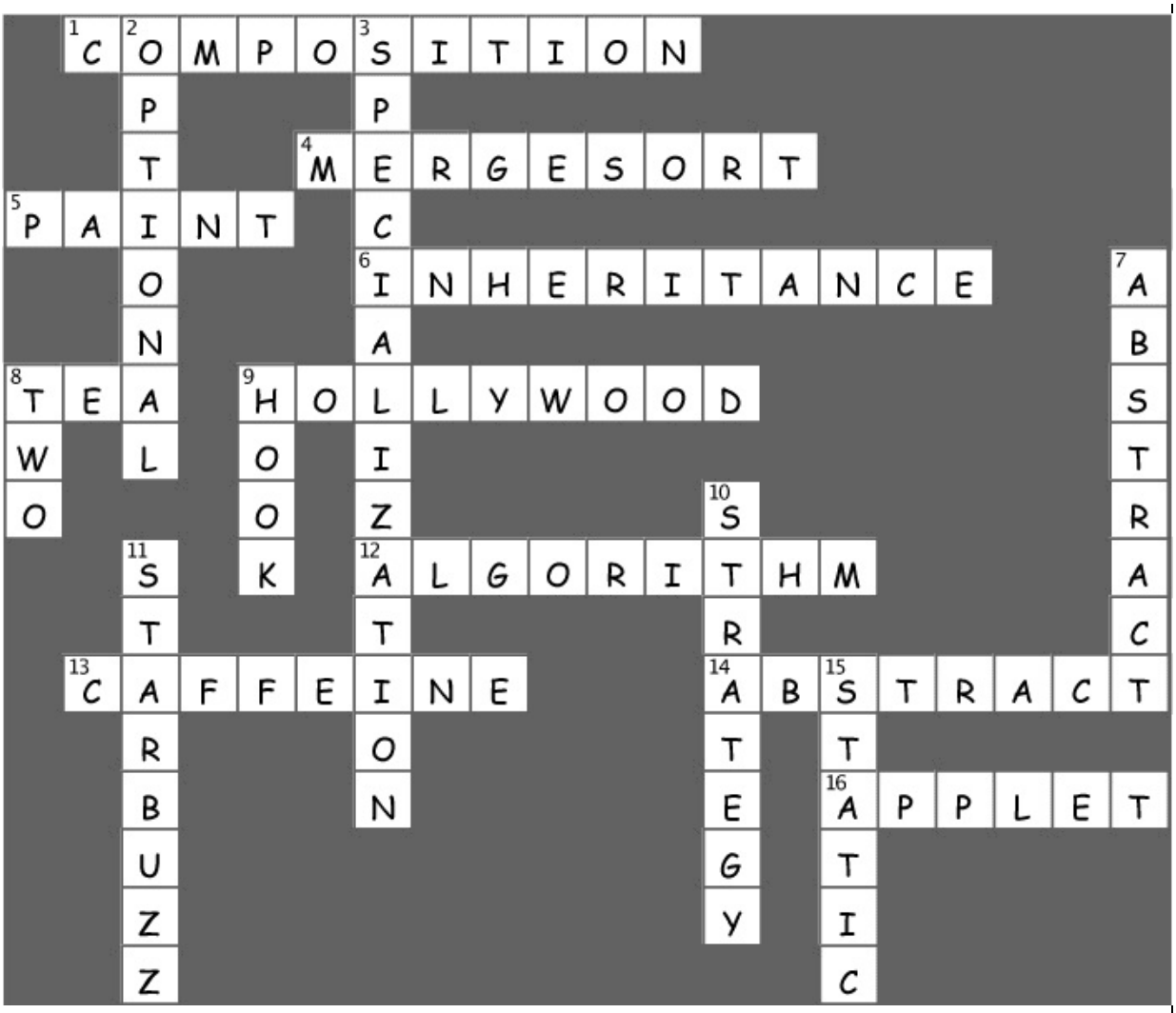
WHO DOES WHAT? SOLUTION

Match each pattern with its description:



DESIGN PATTERNS CROSSWORD SOLUTION

It’s that time again...


1 C O M P O S I T I O N
P P
T 4 M E R G E S O R T
5 P A I N T C
O 6 I N H E R I T A N C E 7 A
N A B
8 T E A 9 H O L L Y W O O D S
W L O I T
O O Z 10 S R
11 S K 12 A L G O R I T H M A
T T R C
13 C A F F E I N E 14 A B 15 S T R A C T
R O T T
B N E 16 A P P L E T
U G T
Z Y I
Z C

Chapter 9. The Iterator and Composite Patterns: Well-Managed Collections

There are lots of ways to stuff objects into a collection. Put them into an Array, a Stack, a List, a Hashmap, take your pick. Each has its own advantages and tradeoffs. But at some point your client is going to want to iterate over those objects, and when he does, are you going to show him your implementation? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; you're going to see how you can allow your clients to iterate through your objects without ever getting a peek at how you store your objects. You're also going to learn how to create some super collections of objects that can leap over some impressive data structures in a single bound. And if that's not enough, you're also going to learn a thing or two about object responsibility.

Breaking News: Objectville Diner and Objectville Pancake House Merge

That's great news! Now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But, there seems to be a slight problem...

Check out the Menu Items

At least Lou and Mel agree on the implementation of the MenuItems. Let's check out the items on each menu, and also take a look at the implementation.

The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price.

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

Lou and Mel's Menu implementations

Now let's take a look at what Lou and Mel are arguing about. They both have lots of time and code invested in the way they store their menu items in a menu, and lots of other code that depends on it.


I used an ArrayList
so I can easily expand
my menu.

Here's Lou's implementation of the Pancake House menu.

```
public class PancakeHouseMenu {
    ArrayList<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Lou's using an ArrayList to store his menu items.

Each menu item is added to the ArrayList here, in the constructor.

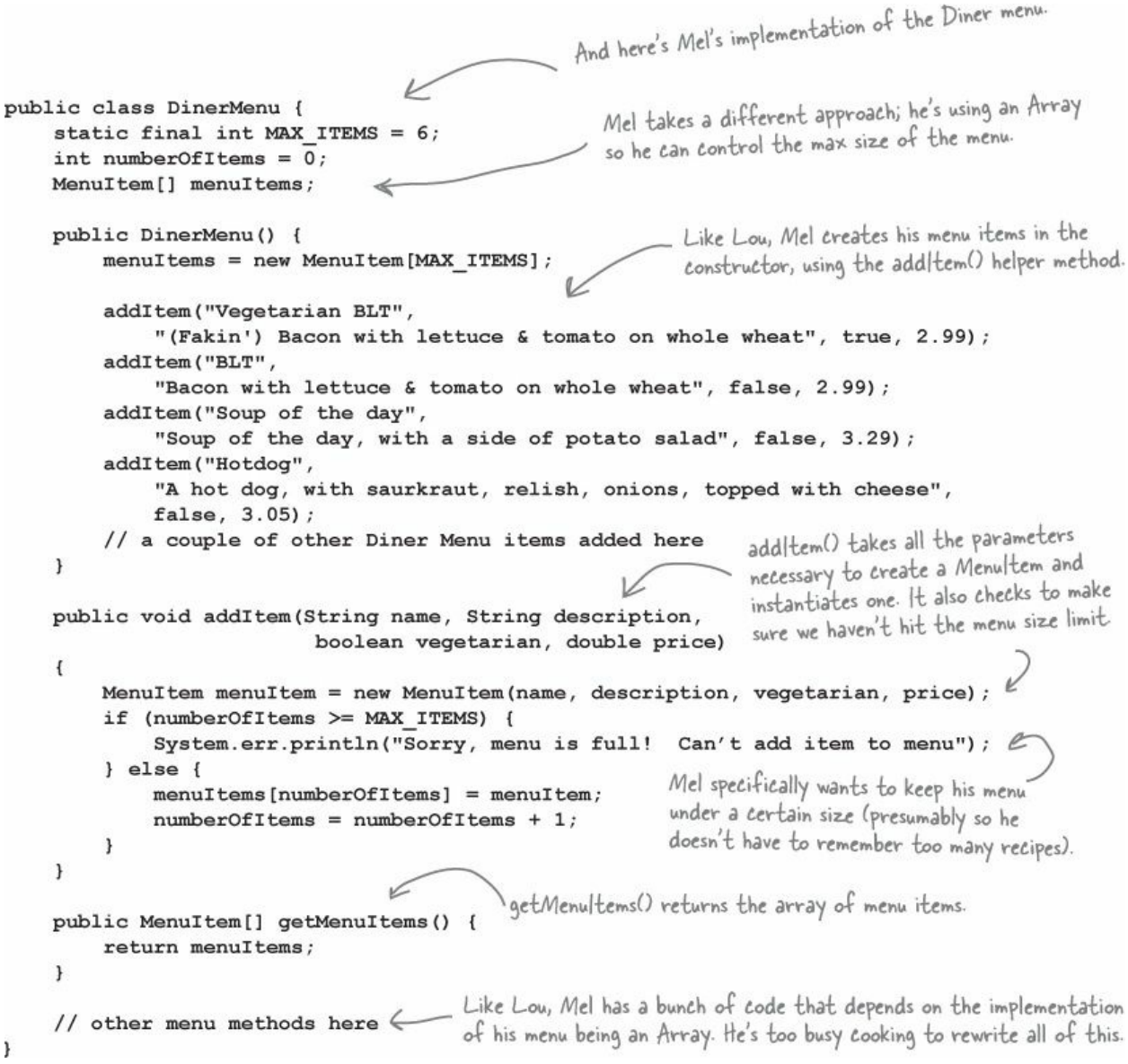
Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price.

To add a menu item, Lou creates a new MenuItem object, passing in each argument, and then adds it to the ArrayList.

The getMenuItems() method returns the list of menu items.

Lou has a bunch of other menu code that depends on the ArrayList implementation. He doesn't want to have to rewrite all that code!


Haah! An ArrayList... I used a REAL Array so I can control the maximum size of my menu.




```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

What's the problem with having two different menu representations?

To see why having two different menu representations complicates things, let's try implementing a client that uses the two menus. Imagine you have been hired by the new company formed by the merger of the Diner and the Pancake House to create a Java-enabled waitress (this is Objectville, after all). The spec for the Java-enabled waitress specifies that she can print a custom menu for customers on demand, and even tell you if a menu item is vegetarian without having to ask the cook — now that's an innovation!

Let’s check out the spec, and then step through what it might take to implement her...

The Java-Enabled Waitress Specification


```
Java-Enabled Waitress: code-name "Alice"

printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is vegetarian, otherwise,
     returns false
```

The spec for the Waitress

Let’s start by stepping through how we’d implement the printMenu() method:

① To print all the items on each menu, you’ll need to call the getMenuItems() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The method looks the same, but the calls are returning different types.

The implementation is showing through: breakfast items are in an ArrayList, and lunch items are in an Array.

② Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items we'll loop through the Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Now, we have to implement two different loops to step through the two implementations of the menu items...

...one loop for the ArrayList...

...and another for the Array.

③ Implementing every other method in the Waitress is going to be a variation of this theme. We're always going to need to get both menus and use two loops to iterate through their items. If another restaurant with a different implementation is acquired then we'll have *three* loops.

SHARPEN YOUR PENCIL

Based on our implementation of printMenu(), which of the following apply?

❑	A.	We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
❑	B.	The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a

		standard.
❑	C.	If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.
❑	D.	The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
❑	E.	We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
❑	F.	The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

What now?

Mel and Lou are putting us in a difficult position. They don't want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class. But if one of them doesn't give in, then we're going to have the job of implementing a Waitress that is going to be hard to maintain and extend.

It would really be nice if we could find a way to allow them to implement the same interface for their menus (they're already close, except for the return type of the getMenuItems() method). That way we can minimize the concrete references in the Waitress code and also hopefully get rid of the multiple loops required to iterate over both menus.

Sound good? Well, how are we going to do that?

Yes, using for each would allow us to hide the complexity of the different kinds of iteration. But that doesn't solve the real problem here: that we've got two different implementations of the menus, and the Waitress has to know how each kind of menu is implemented. That's not really the Waitress's job. We want her to focus on being a waitress, and not have to think about the type of the menus *at all*.

Even if we use for each loops to iterate through the menus, the Waitress still has to know about the type of each menu.

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
for (MenuItem menuItem : lunchItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
```

Our goal is to decouple the Waitress from the concrete implementations of

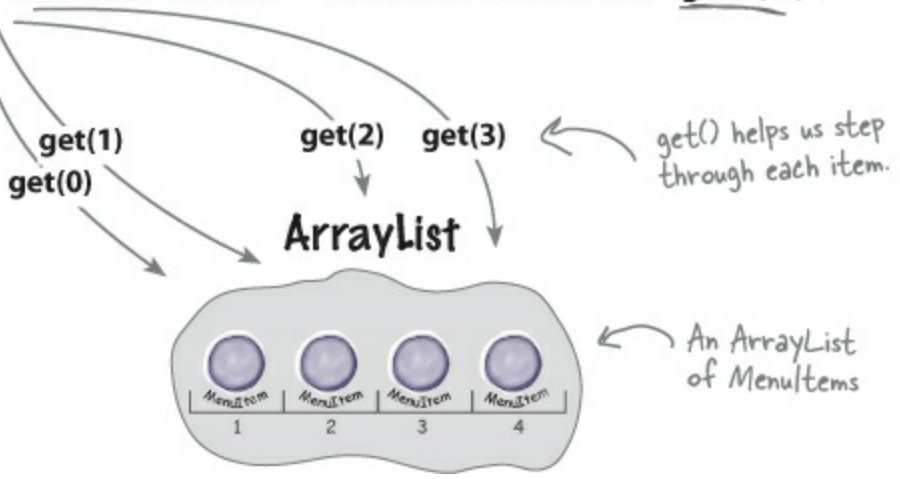
the menus completely. So hang in there, and you'll see there's a better way to do this.

Can we encapsulate the iteration?

If we've learned one thing in this book, it's encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this? Let's work through the idea...

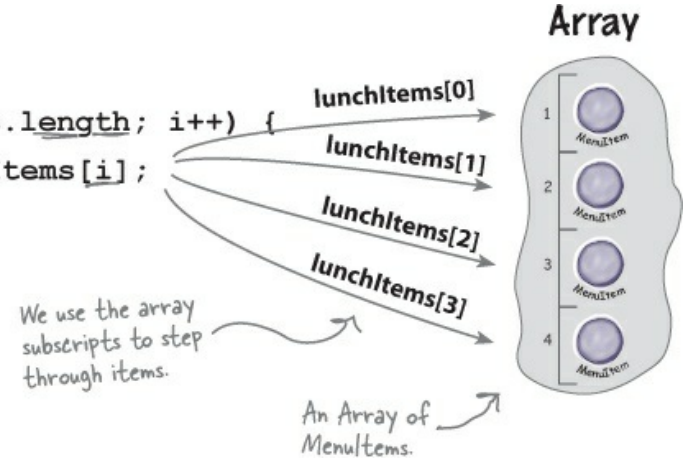
① To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



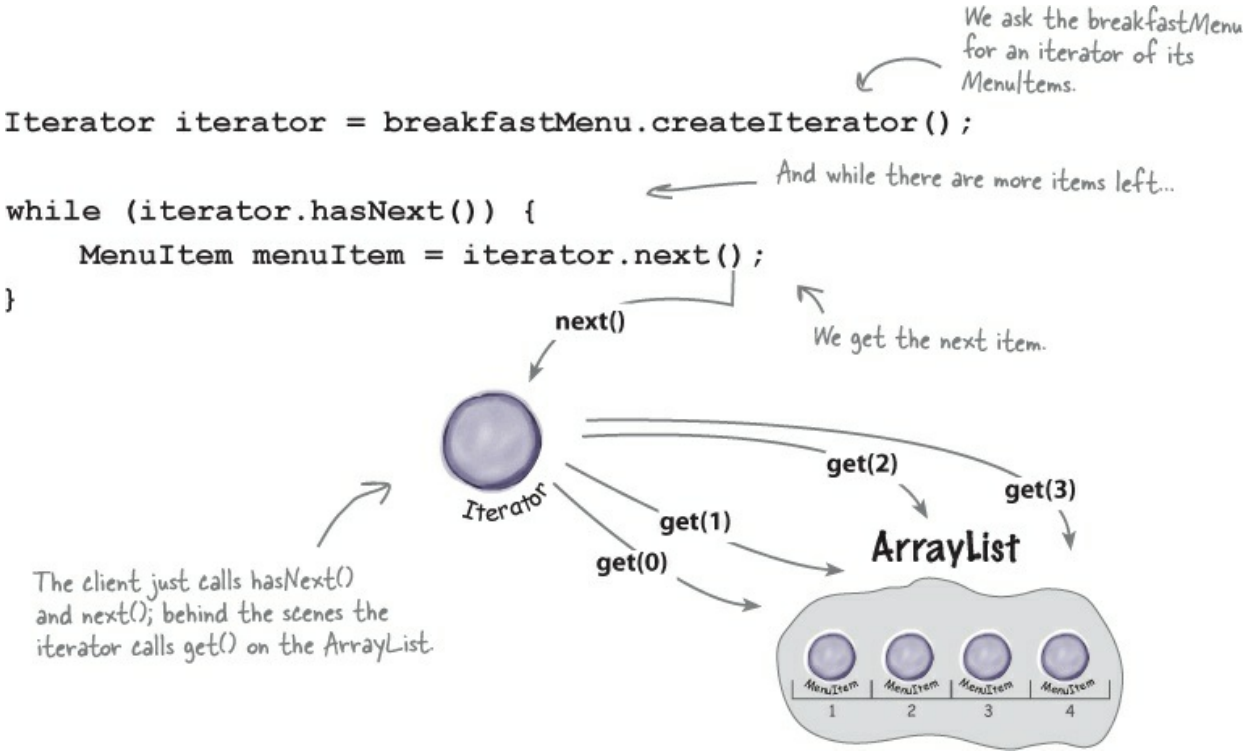
② And to iterate through the lunch items we use the Array length field and the array subscript notation on the MenuItem Array.

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

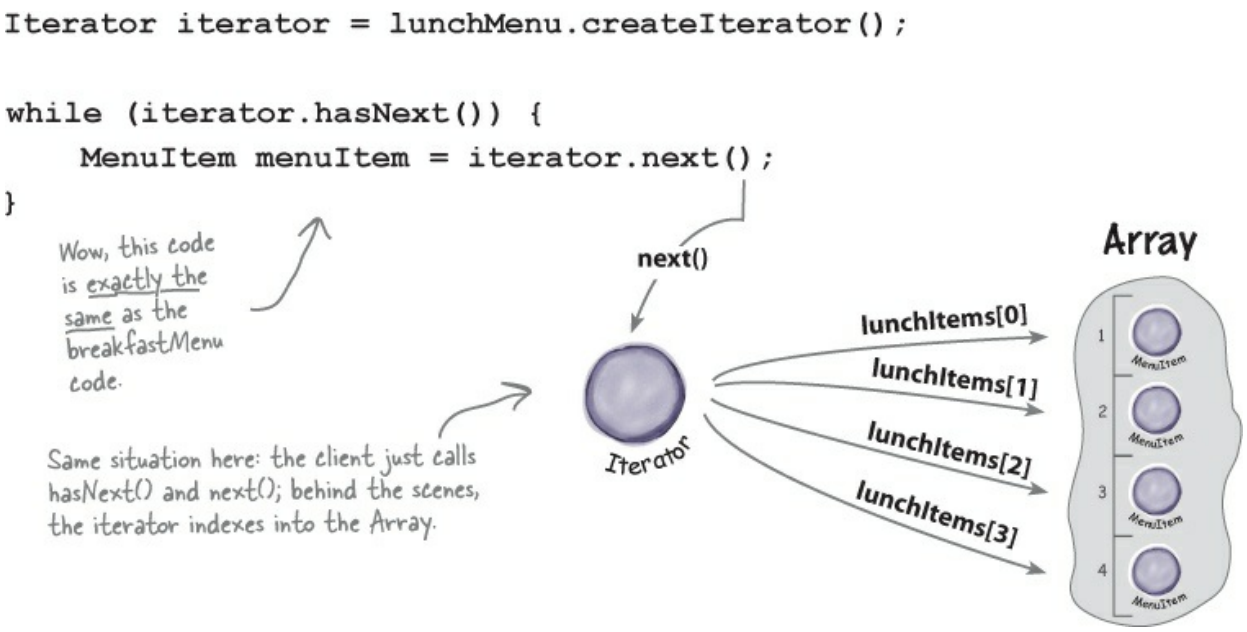


③ Now what if we create an object, let's call it an Iterator, that

encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList



④ Let's try that on the Array too:

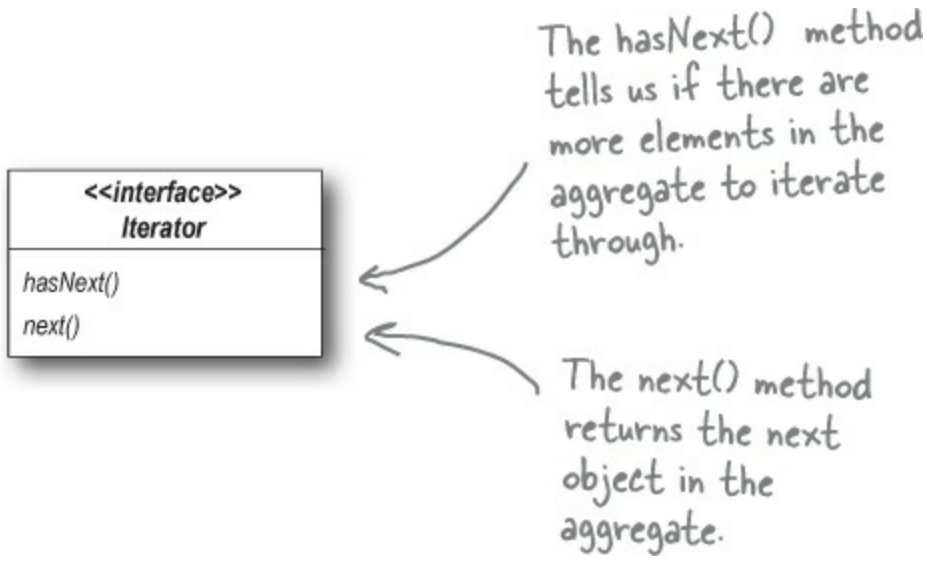


Meet the Iterator Pattern

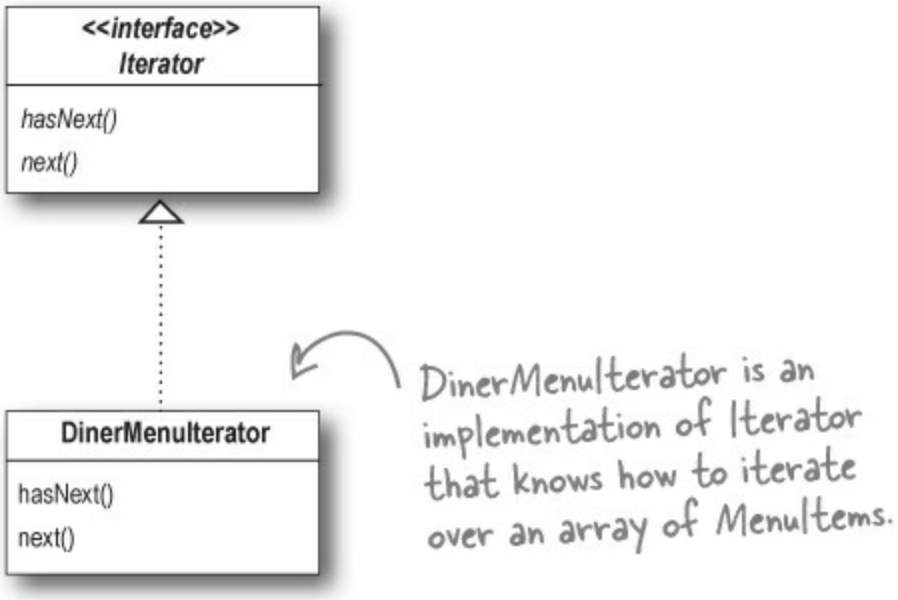
Well, it looks like our plan of encapsulating iteration just might actually

work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashmaps, ...pick your favorite collection of objects. Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:



Let's go ahead and implement this Iterator and hook it into the DinerMenu to see how this works...

Adding an Iterator to DinerMenu

To add an Iterator to the DinerMenu we first need to define the Iterator Interface:

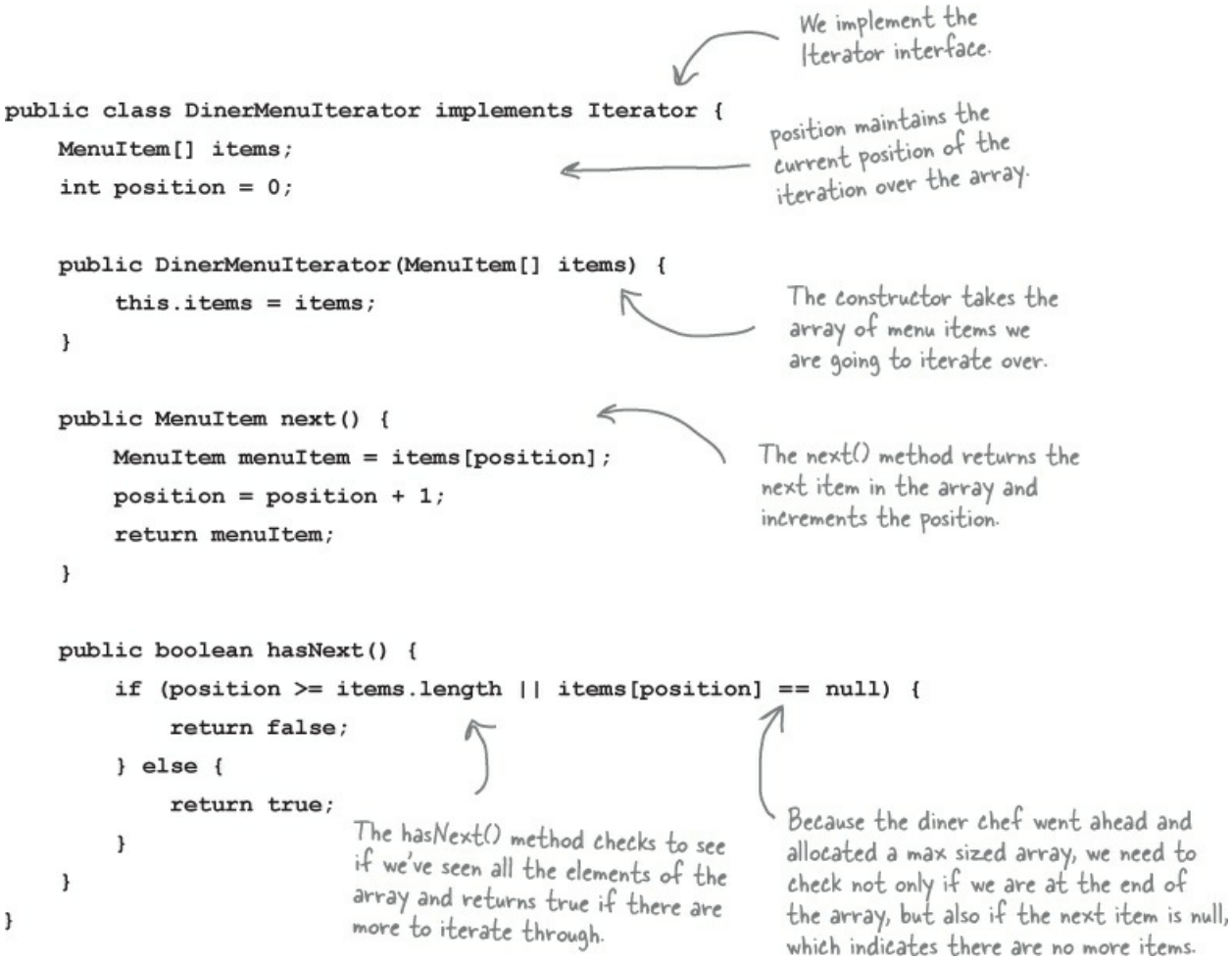
```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Here are our two methods:

The hasNext() method returns a boolean indicating whether or not there are more elements to iterate over...

...and the next() method returns the next element.

And now we need to implement a concrete Iterator that works for the Diner menu:




```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

Reworking the Diner Menu with Iterator

Okay, we've got the iterator. Time to work it into the DinerMenu; all we need to do is add one method to create a DinerMenuIterator and return it to the client:

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // constructor here

    // addItem here

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // other menu methods here
}
```

We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

EXERCISE

Go ahead and implement the PancakeHouseIterator yourself and make the changes needed to incorporate it into the PancakeHouseMenu.

Fixing up the Waitress code

Now we need to integrate the iterator code into the Waitress. We should be able to get rid of some of the redundancy in the process. Integration is pretty straightforward: first we create a printMenu() method that takes an Iterator; then we use the createIterator() method on each menu to retrieve the Iterator and pass it to the new method.

New and improved with Iterator.

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

And then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Use the item to get name, price, and description and print them.

Note that we're down to one loop.

Testing our code

It's time to put everything to a test. Let's write some test drive code and see how the Waitress works...

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu();
    }
}
```

First we create the new menus.

Then we create a Waitress and pass her the menus.

Then we print them.

Here's the test run...

```
File Edit Window Help GreenEggs&Ham
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

%
```

First we iterate through the pancake menu.

And then the lunch menu, all with the same iteration code.

What have we done so far?

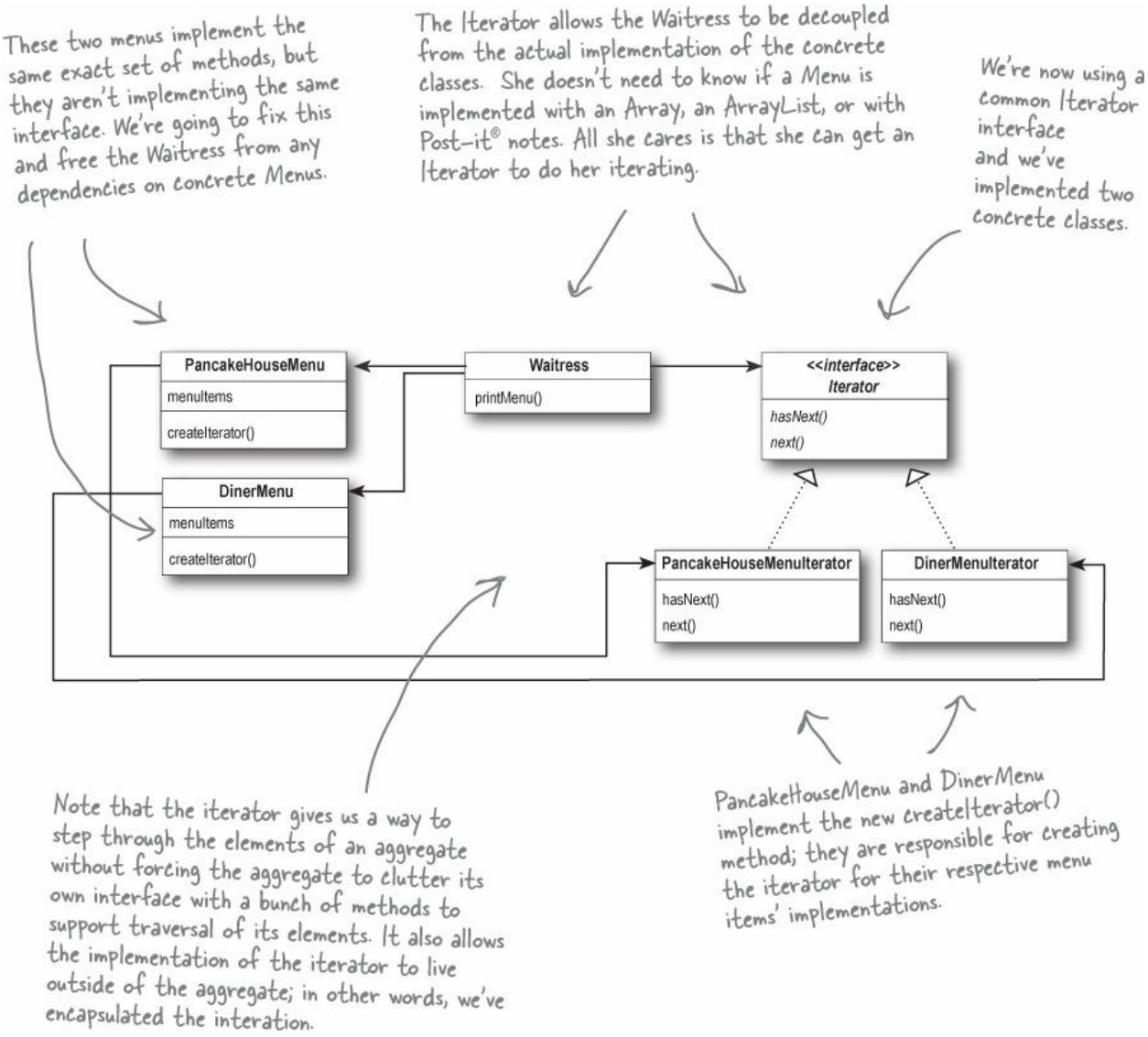
For starters, we've made our Objectville cooks very happy. They settled their differences and kept their own implementations. Once we gave them a PancakeHouseMenuIterator and a DinerMenuIterator, all they had to do was add a createIterator() method and they were finished.

We've also helped ourselves in the process. The Waitress will be much easier to maintain and extend down the road. Let's go through exactly what we did and think about the consequences:

Hard to Maintain Waitress Implementation	New, Hip Waitress Powered by Iterator
The Menus are not well encapsulated; we can see the Diner is using an ArrayList and the Pancake House an Array.	The Menu implementations are now encapsulated. The Waitress has no idea how the Menus hold their collection of menu items.
We need two loops to iterate through the MenuItems.	All we need is a loop that polymorphically handles any collection of items as long as it implements Iterator.
The Waitress is bound to concrete classes (MenuItem[] and ArrayList).	The Waitress now uses an interface (Iterator).
The Waitress is bound to two different concrete Menu classes, despite their interfaces being almost identical.	The Menu interfaces are now exactly the same and, uh oh, we still don't have a common interface, which means the Waitress is still bound to two concrete Menu classes. We'd better fix that.

What we have so far...

Before we clean things up, let's get a bird's-eye view of our current design.

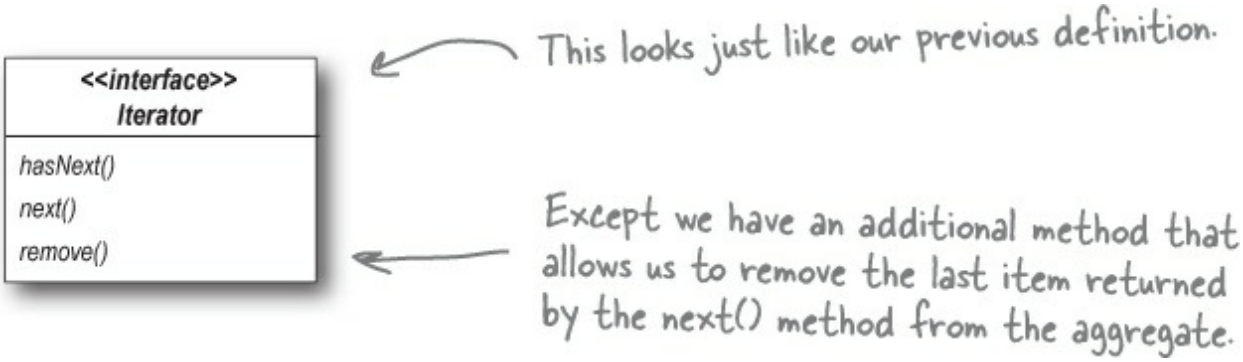


Making some improvements...

Okay, we know the interfaces of PancakeHouseMenu and DinerMenu are exactly the same and yet we haven't defined a common interface for them. So, we're going to do that and clean up the Waitress a little more.

You may be wondering why we're not using the Java Iterator interface — we did that so you could see how to build an iterator from scratch. Now that we've done that, we're going to switch to using the Java Iterator interface, because we'll get a lot of leverage by implementing that instead of our home-grown Iterator interface. What kind of leverage? You'll soon see.

First, let's check out the java.util.Iterator interface:



This is going to be a piece of cake: we just need to change the interface that both PancakeHouseMenuIterator and DinerMenuIterator extend, right? Almost... actually, it's even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator. In other words, we never needed to implement our own iterator for ArrayList. However, we'll still need our implementation for the DinerMenu because it relies on an Array, which doesn't support the iterator() method (or any other way to create an array iterator).

THERE ARE NO DUMB QUESTIONS

Q: **Q: What if I don't want to provide the ability to remove something from the underlying collection of objects?**

A: **A:** The remove() method is considered optional. You don't have to provide remove functionality. But, you should provide the method because it's part of the Iterator interface. If you're not going to allow remove() in your iterator you'll want to throw the runtime exception java.lang.UnsupportedOperationException. The Iterator API documentation specifies that this exception may be thrown from remove() and any client that is a good citizen will check for this exception when calling the remove() method.

Q: **Q: How does remove() behave under multiple threads that may be using different iterators over the same collection of objects?**

A: **A:** The behavior of the remove() is unspecified if the collection changes while you are iterating over it. So you should be careful in designing your own multithreaded code when accessing a collection concurrently.

Cleaning things up with java.util.Iterator

Let's start with the PancakeHouseMenu. Changing it over to java.util.Iterator is going to be easy. We just delete the PancakeHouseMenuIterator class, add an import java.util.Iterator to the top of PancakeHouseMenu and change one line of the PancakeHouseMenu:

```
public Iterator<MenuItem> createIterator() {
    return menuItems.iterator();
}
```

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList.

And that's it, PancakeHouseMenu is done.

Now we need to make the changes to allow the DinerMenu to work with java.util.Iterator.

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public MenuItem next() {
        //implementation here
    }

    public boolean hasNext() {
        //implementation here
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

First we import java.util.Iterator, the interface we're going to implement.

None of our current implementation changes...

...but we do need to implement remove(). Here, because the chef is using a fixed-size Array, we just shift all the elements up one when remove() is called.

We are almost there...

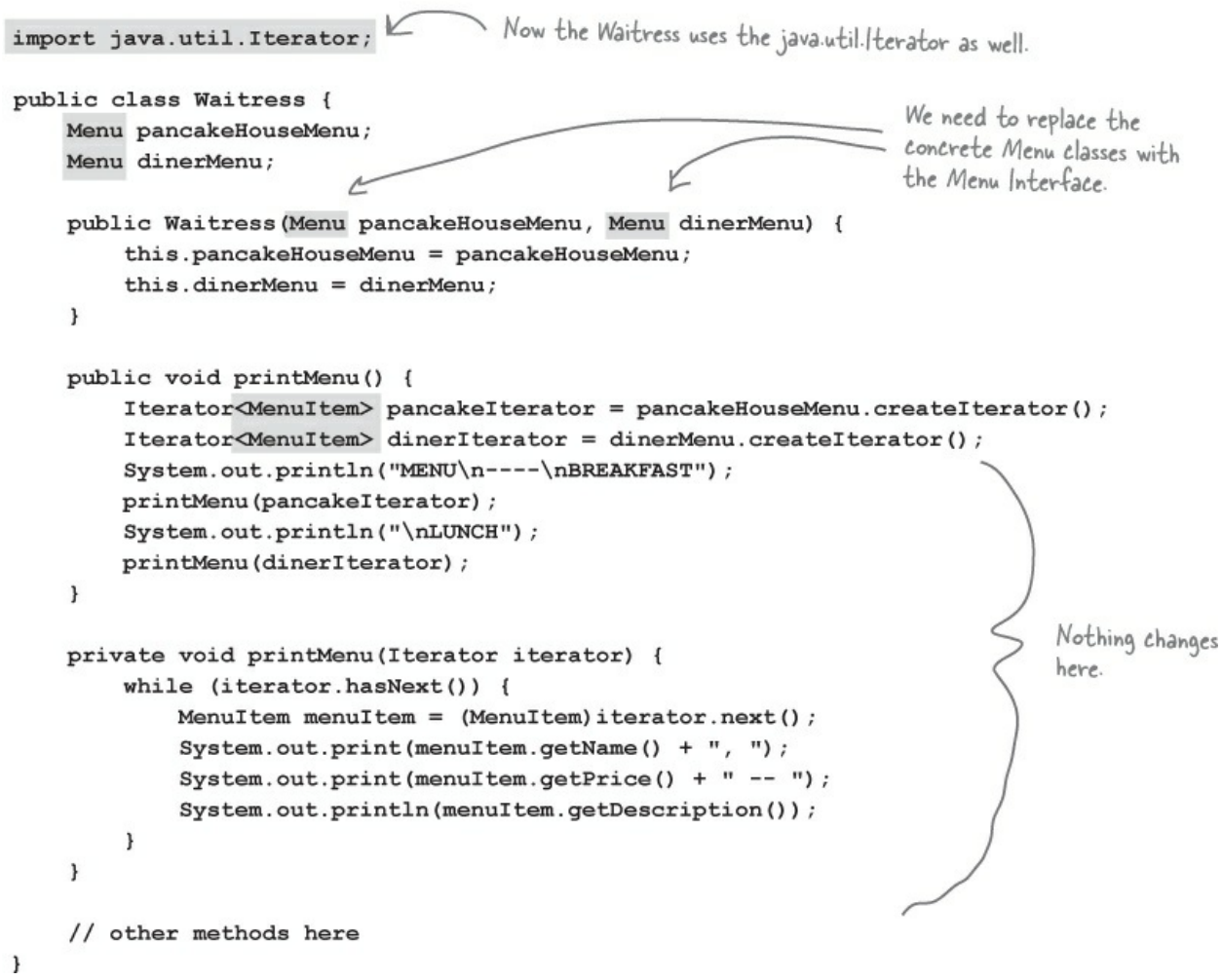
We just need to give the Menus a common interface and rework the Waitress a little. The Menu interface is quite simple: we might want to add a few more methods to it eventually, like addItem(), but for now we will let the chefs control their menus by keeping that method out of the public interface:

```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}
```

This is a simple interface that just lets clients get an iterator for the items in the menu.

Now we need to add an `implements Menu` to both the PancakeHouseMenu

and the DinerMenu class definitions and update the Waitress:




```
import java.util.Iterator;

public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

What does this get us?

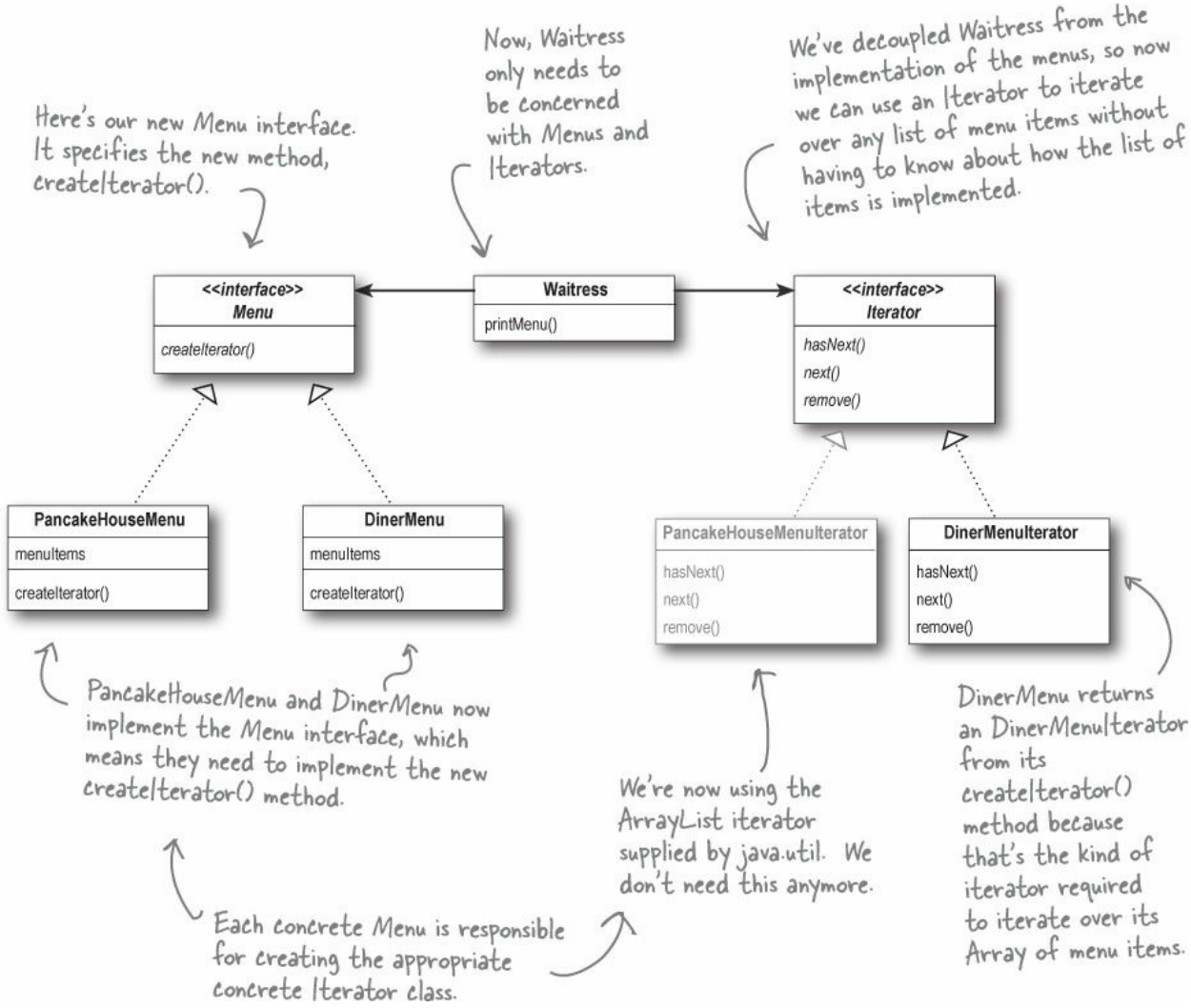
The PancakeHouseMenu and DinerMenu classes implement an interface, Menu. Waitress can refer to each menu object using the interface rather than the concrete class. So, we're reducing the dependency between the Waitress and the concrete classes by "programming to an interface, not an implementation."

> **NOTE**
>
> This solves the problem of the Waitress depending on the concrete Menus.

The new Menu interface has one method, createIterator(), that is implemented by PancakeHouseMenu and DinerMenu. Each menu class assumes the

responsibility of creating a concrete Iterator that is appropriate for its internal implementation of the menu items.

> **NOTE**
>
> This solves the problem of the Waitress depending on the implementation of the MenuItems.



Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection oriented classes (the ArrayList). Now it's time to check out the official definition of the pattern:

NOTE

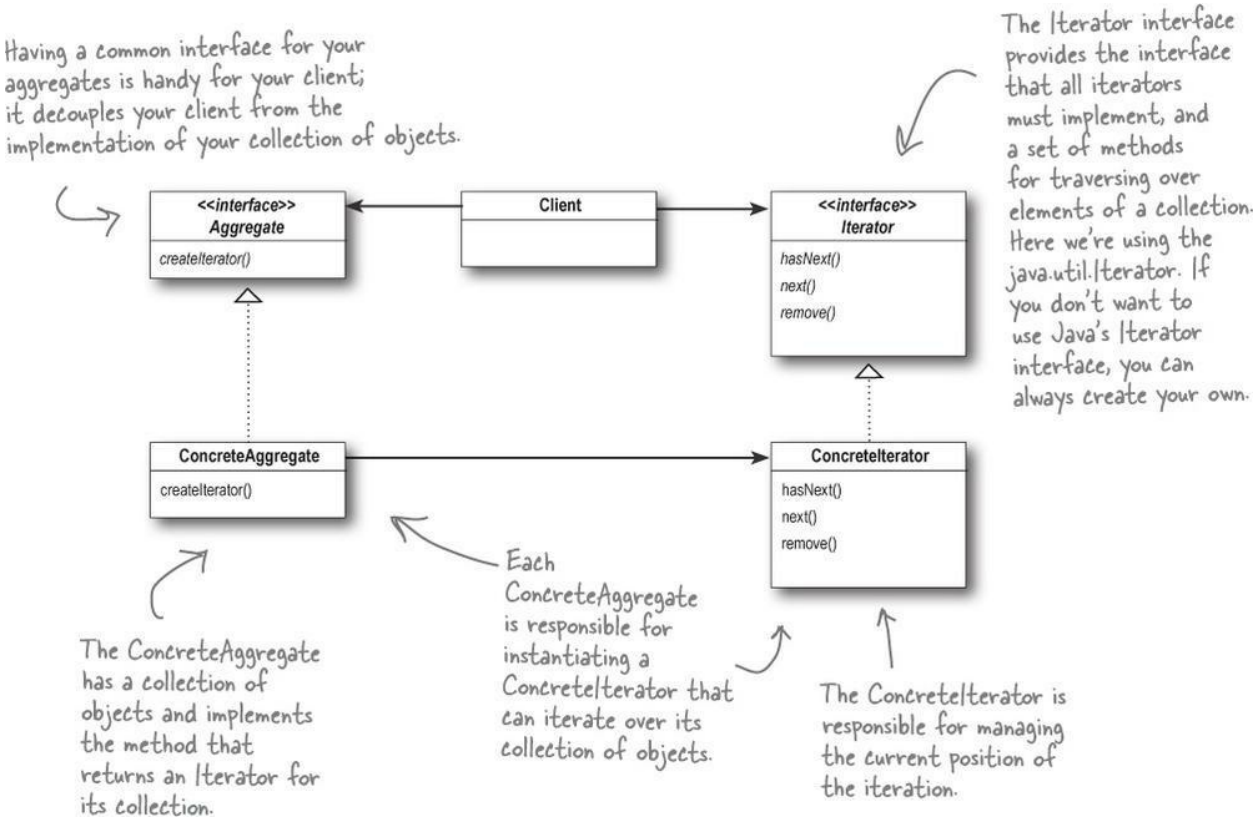
The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates — just like the printMenu() method, which doesn't care if the menu items are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.

> **The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.**
> **It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.**

The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Let's check out the class diagram to put all the pieces in context...



BRAIN POWER

The class diagram for the Iterator Pattern looks very similar to another pattern you've studied; can you think of what it is? Hint: a subclass decides which object to create.

THERE ARE NO DUMB QUESTIONS

Q: **Q: I've seen other books show the Iterator class diagram with the methods first(), next(), isDone() and currentItem(). Why are these methods different?**

A: **A:** Those are the "classic" method names that have been used. These names have changed over time and we now have next(), hasNext() and even remove() in java.util.Iterator.
Let's look at the classic methods. The next() and currentItem() have been merged into one method in java.util. The isDone() method has obviously become hasNext(); but we have no method corresponding to first(). That's because in Java we tend to just get a new iterator whenever we need to start the traversal over. Nevertheless, you can see there is very little difference in these interfaces. In fact, there is a whole range of behaviors you can give your iterators. The remove() method is an example of an extension in java.util.Iterator.

Q: **Q: I've heard about "internal" iterators and "external" iterators. What are they? Which kind did we implement in the example?**

A: **A:** We implemented an external iterator, which means that the client controls the iteration by calling next() to get the next element. An internal iterator is controlled by the iterator itself. In that case, because it's the iterator that's stepping through the elements, you have to tell the iterator what to do with those elements as it goes through them. That means you need a way to pass an operation to an iterator. Internal iterators are less flexible than external iterators because the client doesn't have control of the iteration. However, some might argue that they are easier to use because you just hand them an operation and tell them to iterate, and they do all the work for you.

Q:	**Q: Could I implement an Iterator that can go backwards as well as forwards?**
A:	**A:** Definitely. In that case, you'd probably want to add two methods, one to get to the previous element, and one to tell you when you're at the beginning of the collection of elements. Java's Collection Framework provides another type of iterator interface called ListIterator. This iterator adds previous() and a few other methods to the standard Iterator interface. It is supported by any Collection that implements the List interface.
Q:	**Q: Who defines the ordering of the iteration in a collection like Hashtable, which are inherently unordered?**
A:	**A:** Iterators imply no ordering. The underlying collections may be unordered as in a hashtable or in a bag; they may even contain duplicates. So ordering is related to both the properties of the underlying collection and to the implementation. In general, you should make no assumptions about ordering unless the Collection documentation indicates otherwise.
Q:	**Q: You said we can write "polymorphic code" using an iterator; can you explain that more?**
A:	**A:** When we write methods that take Iterators as parameters, we are using polymorphic iteration. That means we are creating code that can iterate over any collection as long as it supports Iterator. We don't care about how the collection is implemented, we can still write code to iterate over it.
Q:	**Q: If I'm using Java, won't I always want to use the java.util.Iterator interface so I can use my own iterator implementations with classes that are already using the Java iterators?**
A:	**A:** Probably. If you have a common Iterator interface, it will certainly make it easier for you to mix and match your own aggregates with Java aggregates like ArrayList and Vector. But remember, if you need to add functionality to your Iterator interface for your aggregates, you can always extend the Iterator interface.
Q:	**Q: I've seen an Enumeration interface in Java; does that implement the Iterator Pattern?**
A:	**A:** We talked about this in the Adapter Pattern chapter (Chapter 7). Remember? The java.util.Enumeration is an older implementation of Iterator that has since been replaced by java.util. Iterator. Enumeration has two methods, hasMoreElements(), corresponding to hasNext(), and nextElement(), corresponding to next(). However, you'll probably want to use Iterator over Enumeration as more Java classes support it. If you need to convert from one to another, review Chapter 7 again where you implemented the adapter for Enumeration and Iterator.

Single Responsibility

What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Two? Yup, two: it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CHANGE is at the center of another design principle:

DESIGN PRINCIPLE

A class should have only one reason to change.

Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.
This principle guides us to keep each class to a single responsibility.

We know we want to avoid change in a class like the plague — modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

The solution? The principle guides us to assign each responsibility to one class, and only one class.

That's right, it's as easy as that, and then again it's not: separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

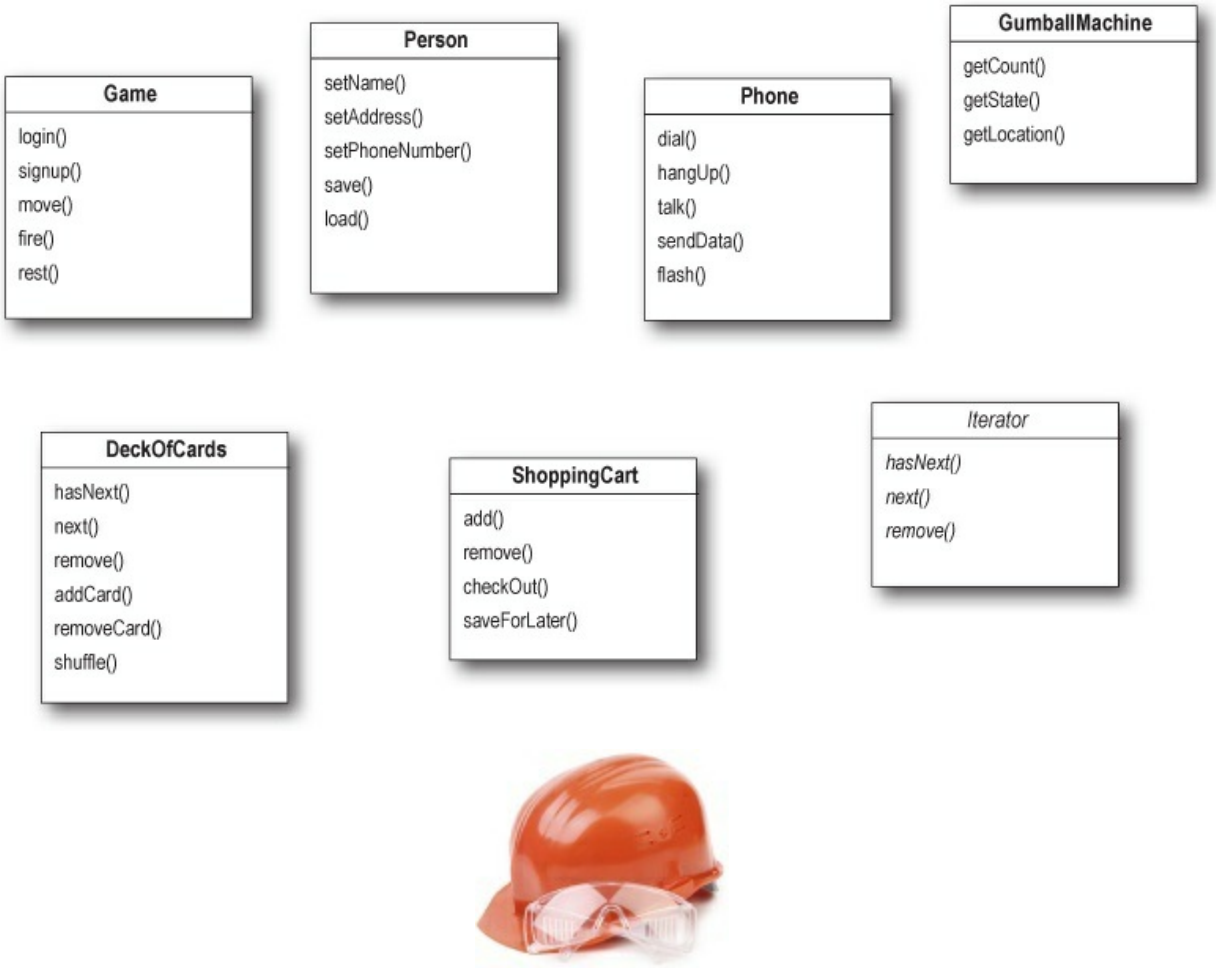
Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

BRAIN POWER

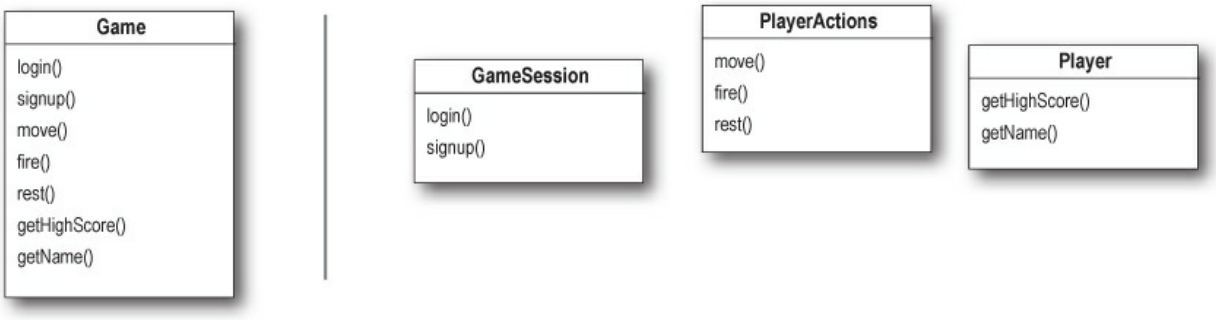
Examine these classes and determine which ones have multiple responsibilities.



HARD HAT AREA. WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN^2 POWER

Determine if these classes have low or high cohesion.



Taking a look at the Café Menu

Here's the café menu. It doesn't look like too much trouble to integrate the CafeMenu class into our framework... let's check it out.

```
public class CafeMenu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Map<String, MenuItem> getItems() {
        return menuItems;
    }
}
```

CafeMenu doesn't implement our new Menu interface, but this is easily fixed.

The café is storing their menu items in a HashMap. Does that support Iterator? We'll see shortly...

Like the other Menus, the menu items are initialized in the constructor.

Here's where we create a new MenuItem and add it to the menuItems hashtable.

The key is the item name.

The value is the menuItem object.

We're not going to need this anymore.

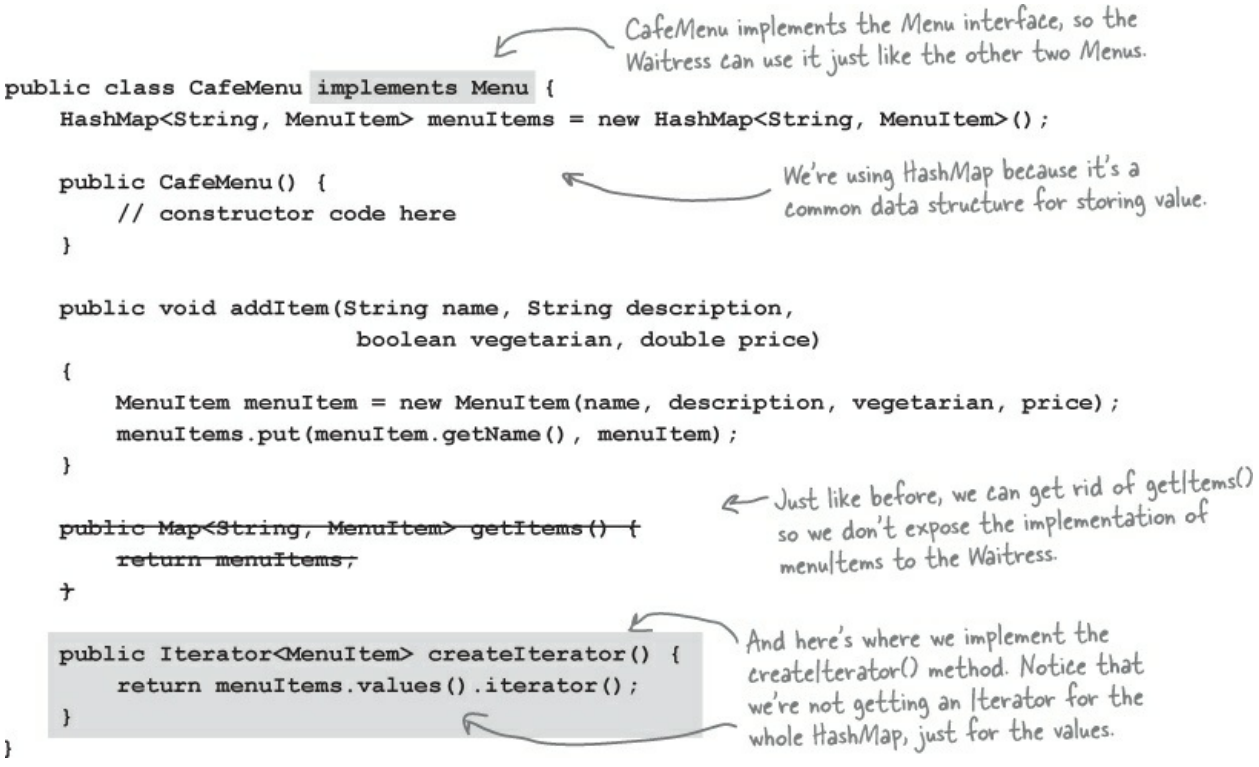
SHARPEN YOUR PENCIL

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. _____
2. _____
3. _____

Reworking the Café Menu code

Integrating the CafeMenu into our framework is easy. Why? Because HashMap is one of those Java collections that supports Iterator. But it's not quite the same as ArrayList...



CODE UP CLOSE

HashMap is a little more complex than the ArrayList because it supports both keys and values, but we can still get an Iterator for the values (which are the MenuItems).

```
public Iterator<MenuItem> createIterator() {
    return menuItems.values().iterator();
}
```

First we get the values of the Hashtable, which is just a collection of all the objects in the hashtable.

Luckily that collection supports the iterator() method, which returns a object of type java.util.Iterator.

Adding the Café Menu to the Waitress

That was easy; how about modifying the Waitress to support our new Menu? Now that the Waitress expects Iterators, that should be easy too.

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
        this.cafeMenu = cafeMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
        System.out.println("\nDINNER");
        printMenu(cafeIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

The café menu is passed into the Waitress in the constructor with the other menus, and we stash it in an instance variable.

We're using the café's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

Nothing changes here.

Breakfast, lunch AND dinner

Let's update our test drive to make sure this all works.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

Create a CafeMenu...

... and pass it to the waitress.

Now, when we print we should see all three menus.

Here's the test run; check out the new dinner menu from the Café!

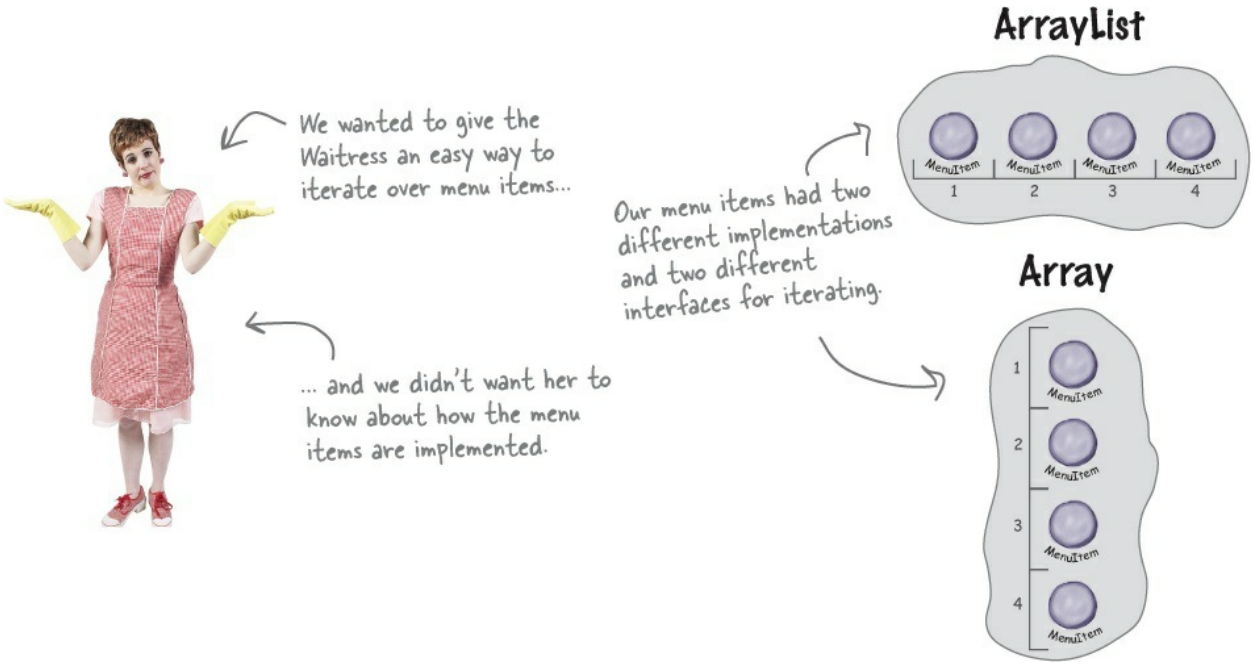
```
File Edit Window Help Kathy&BertLikePancakes
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

First we iterate through the pancake menu.

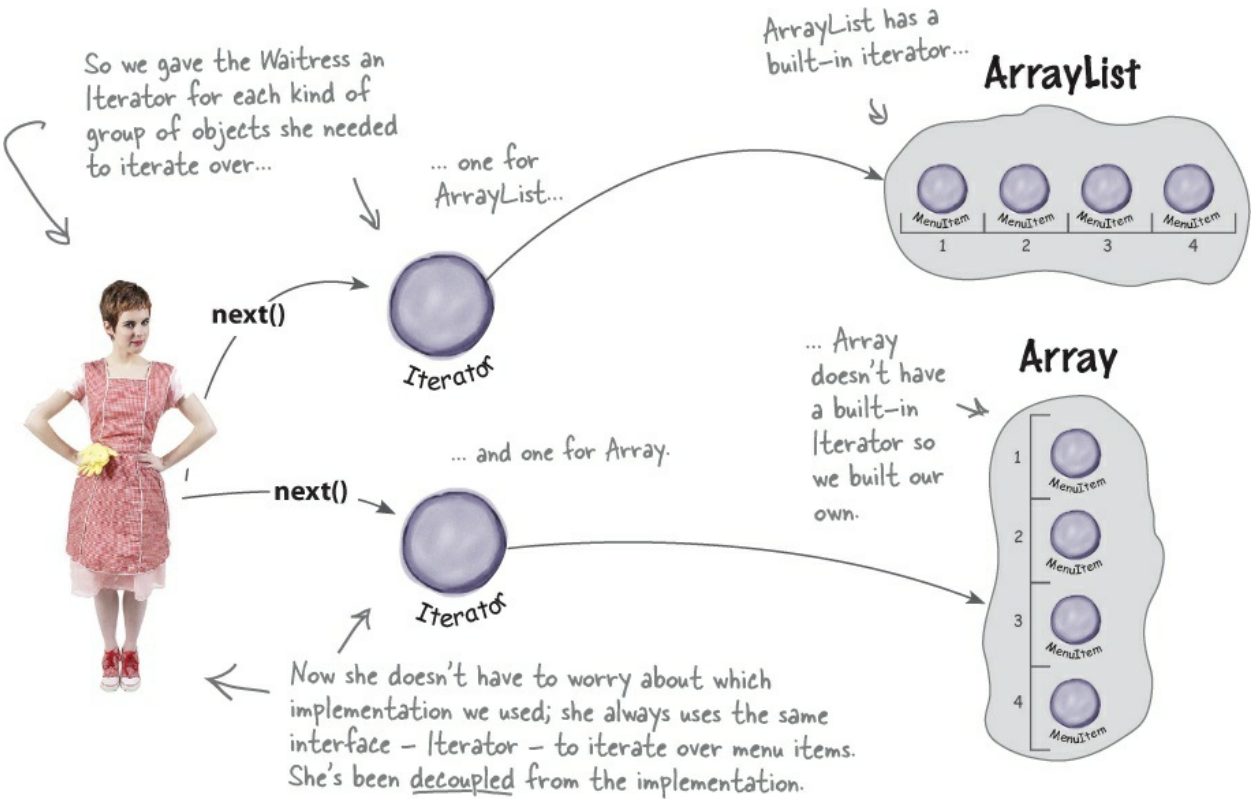
And then the diner menu.

And finally the new café menu, all with the same iteration code.

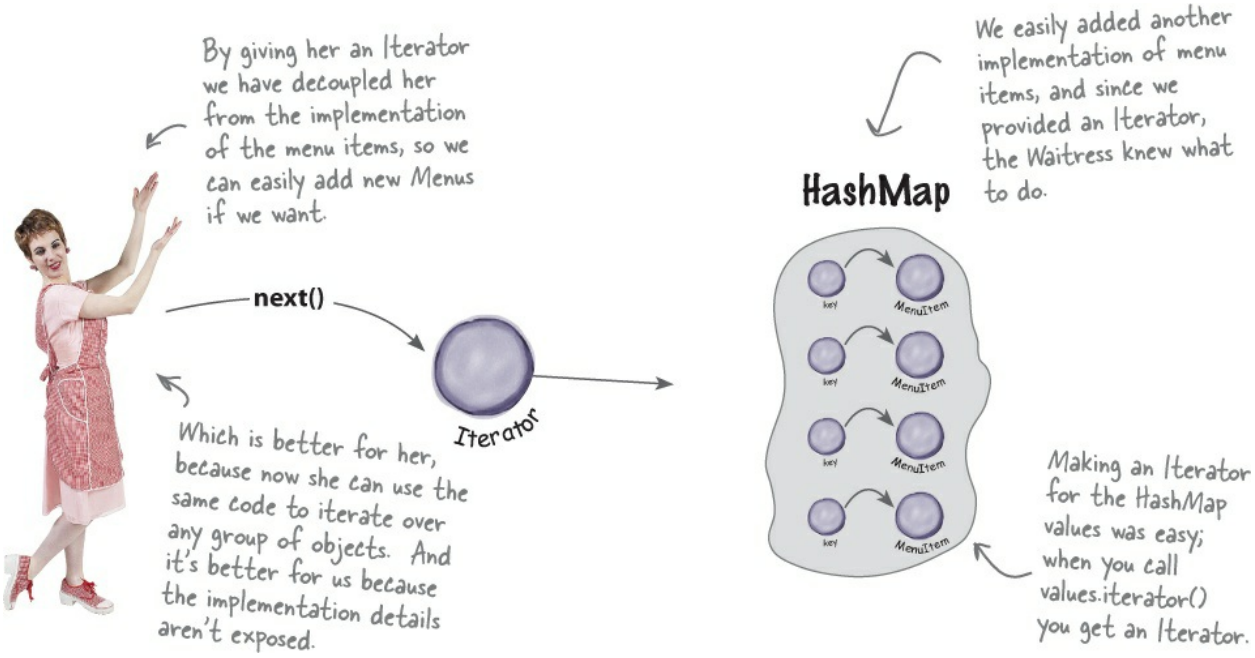
What did we do?



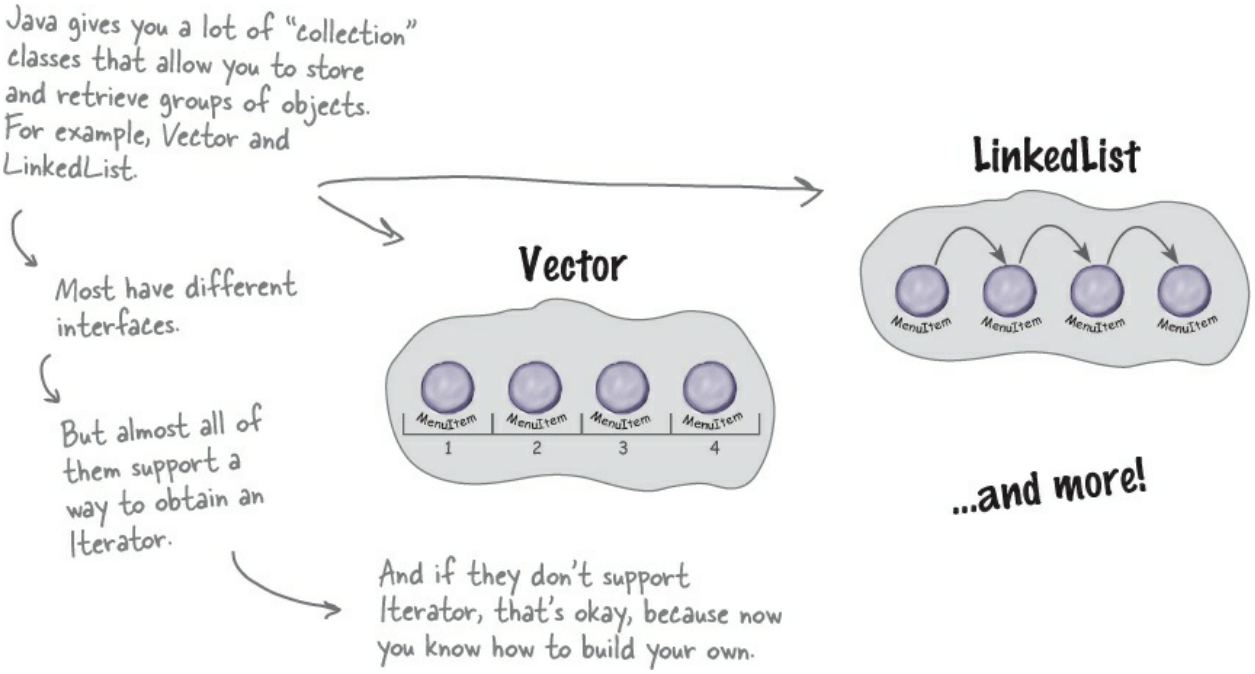
We decoupled the Waitress....



... and we made the Waitress more extensible



But there's more!

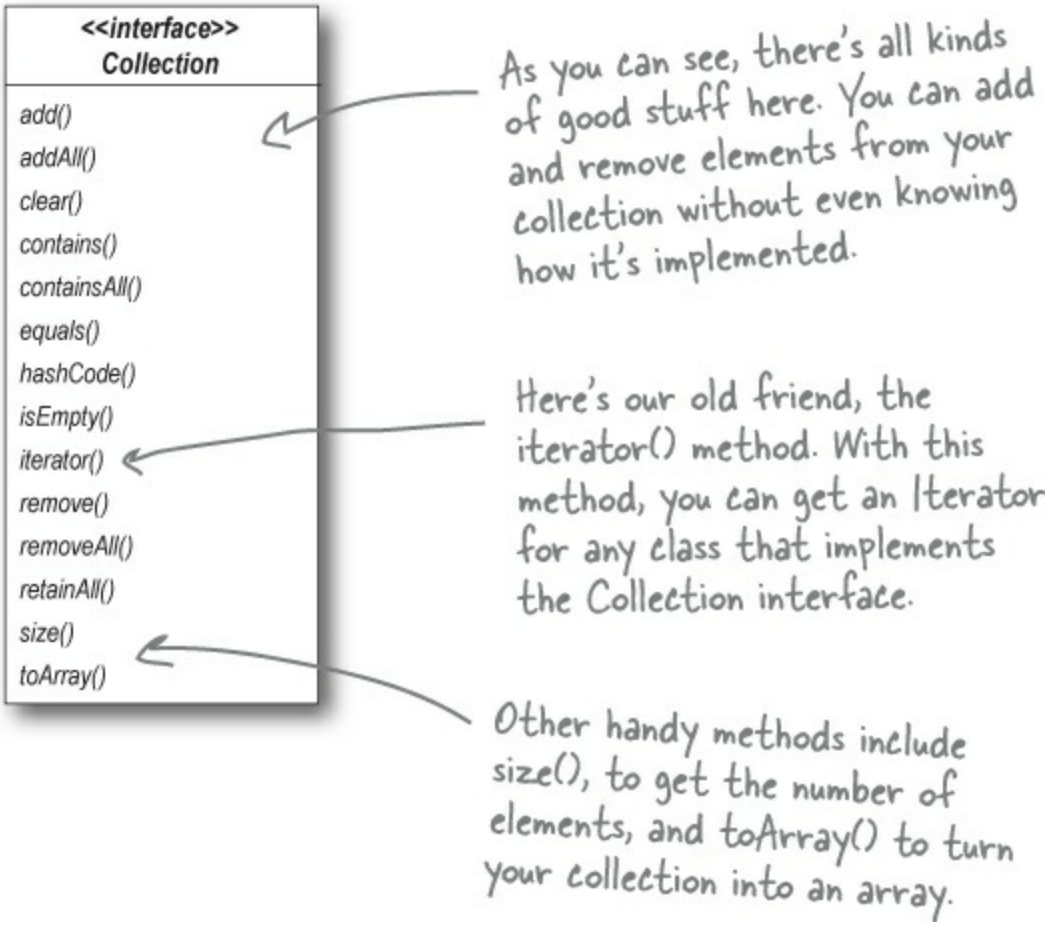


Iterators and Collections

We've been using a couple of classes that are part of the Java Collections Framework. This "framework" is just a set of classes and interfaces, including ArrayList, which we've been using, and many others like Vector,

LinkedList, Stack, and PriorityQueue. Each of these classes implements the java.util.Collection interface, which contains a bunch of useful methods for manipulating groups of objects.

Let's take a quick look at the interface:



WATCH IT!

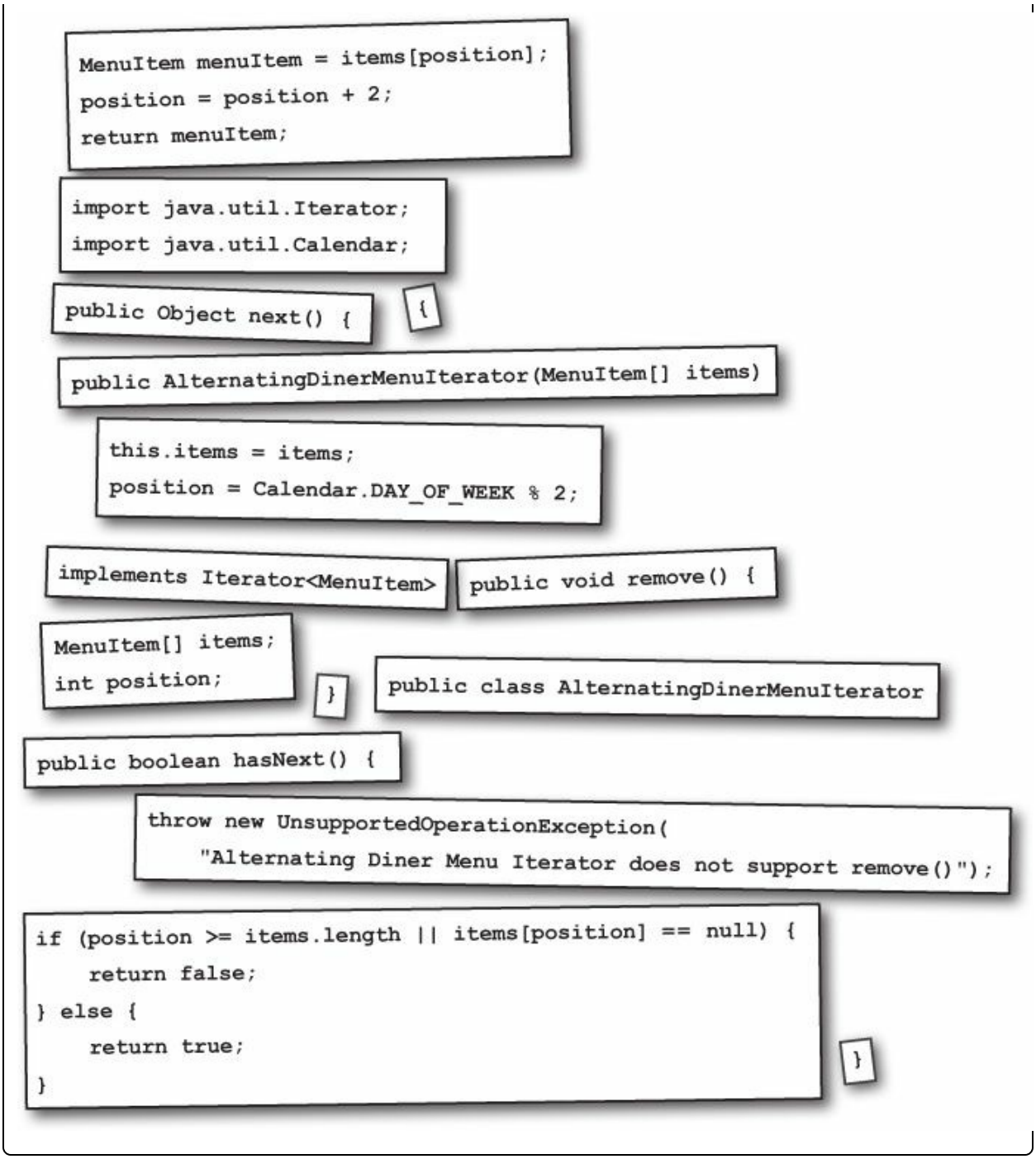
Hashtable is one of a few classes that *indirectly* supports Iterator.

As you saw when we implemented the CafeMenu, you could get an Iterator from it, but only by first retrieving its Collection called values. If you think about it, this makes sense: the HashMap holds two sets of objects: keys and values. If we want to iterate over

its values, we first need to retrieve them from the HashMap, and then obtain the iterator.

CODE MAGNETS

The Chefs have decided that they want to be able to alternate their lunch menu items; in other words, they will offer some items on Monday, Wednesday, Friday, and Sunday, and other items on Tuesday, Thursday, and Saturday. Someone already wrote the code for a new “Alternating” DinerMenu Iterator so that it alternates the menu items, but she scrambled it up and put it on the fridge in the Diner as a joke. Can you put it back together? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.



Is the Waitress ready for prime time?

The Waitress has come a long way, but you've gotta admit those three calls to printMenu() are looking kind of ugly.

Let's be real — every time we add a new menu we are going to have to open up the Waitress implementation and add more code. Can you say "violating

the Open Closed Principle”?

Three createIterator() calls.

```
public void printMenu() {
    Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
    Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

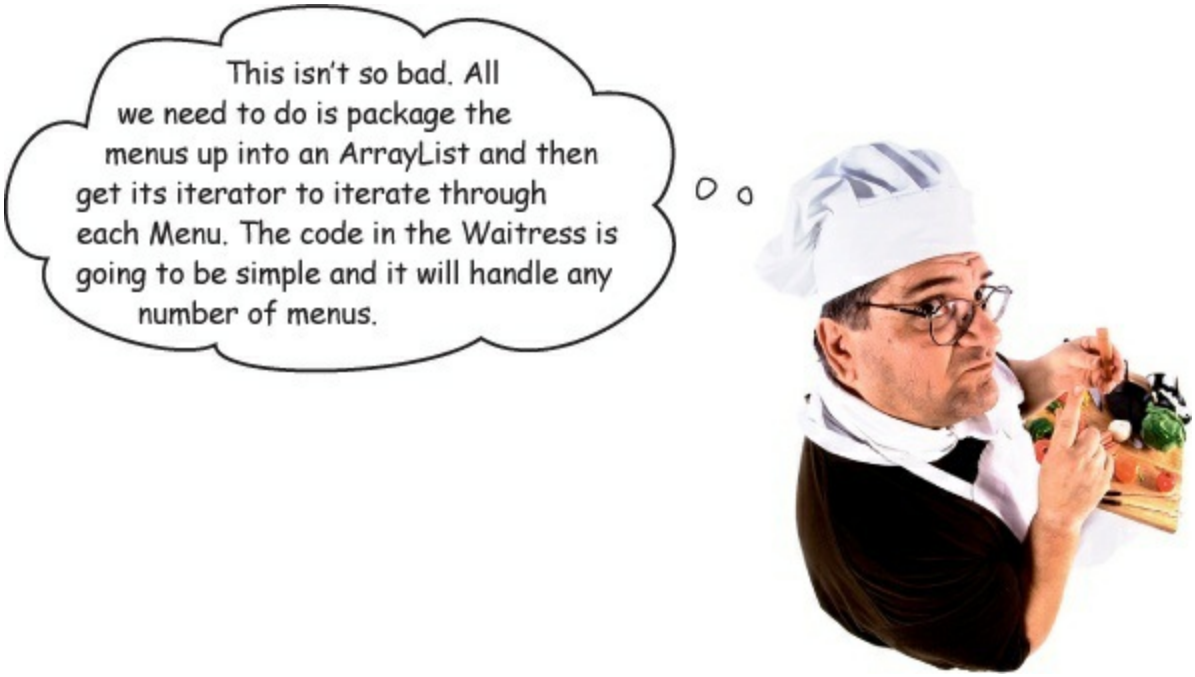
Three calls to printMenu.

Every time we add or remove a menu we're going to have to open this code up for changes.

It’s not the Waitress’ fault. We have done a great job of decoupling the menu implementation and extracting the iteration into an iterator. But we still are handling the menus with separate, independent objects — we need a way to manage them together.

BRAIN POWER

The Waitress still needs to make three calls to printMenu(), one for each menu. Can you think of a way to combine the menus so that only one call needs to be made? Or perhaps so that one Iterator is passed to the Waitress to iterate over all the menus?



Sounds like the chef is on to something. Let's give it a try:

```
public class Waitress {
    ArrayList<Menu> menus;

    public Waitress(ArrayList<Menu> menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator<Menu> menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator<Menu> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Now we just take an ArrayList of menus.

And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

No code changes here.

This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.

Just when we thought it was safe...

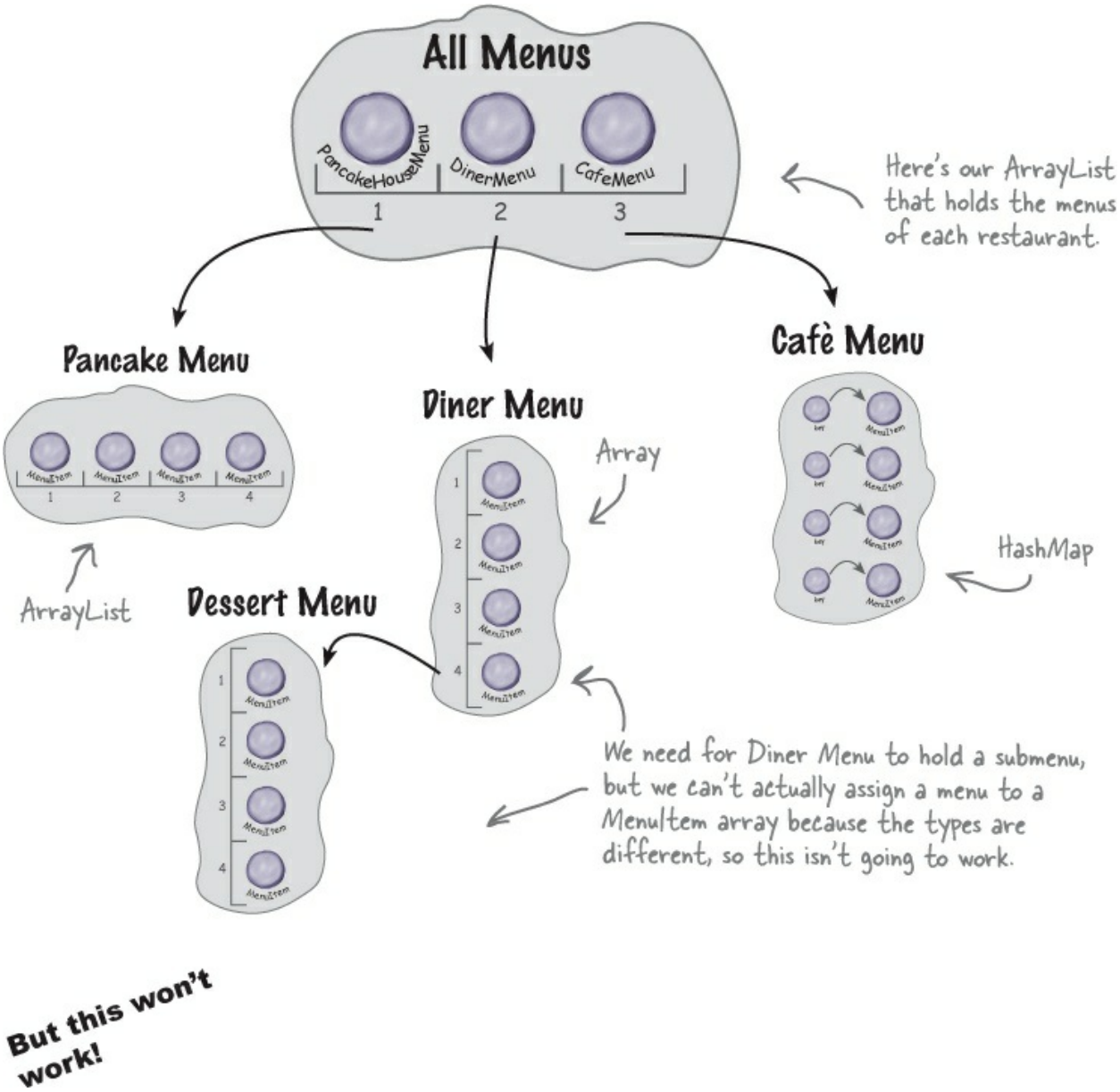
Now they want to add a dessert submenu.

Okay, now what? Now we have to support not only multiple menus, but menus within menus.

It would be nice if we could just make the dessert menu an element of the DinerMenu collection, but that won't work as it is now implemented.

What we want (something like this):


I just heard the Diner is going to be creating a dessert menu that is going to be an insert into their regular menu.



We can’t assign a dessert menu to a MenuItem array.

Time for a change!

What do we need?

The time has come to make an executive decision to rework the chef’s implementation into something that is general enough to work over all the menus (and now submenus). That’s right, we’re going to tell the chefs that the time has come for us to reimplement their menus.

The reality is that we’ve reached a level of complexity such that if we don’t

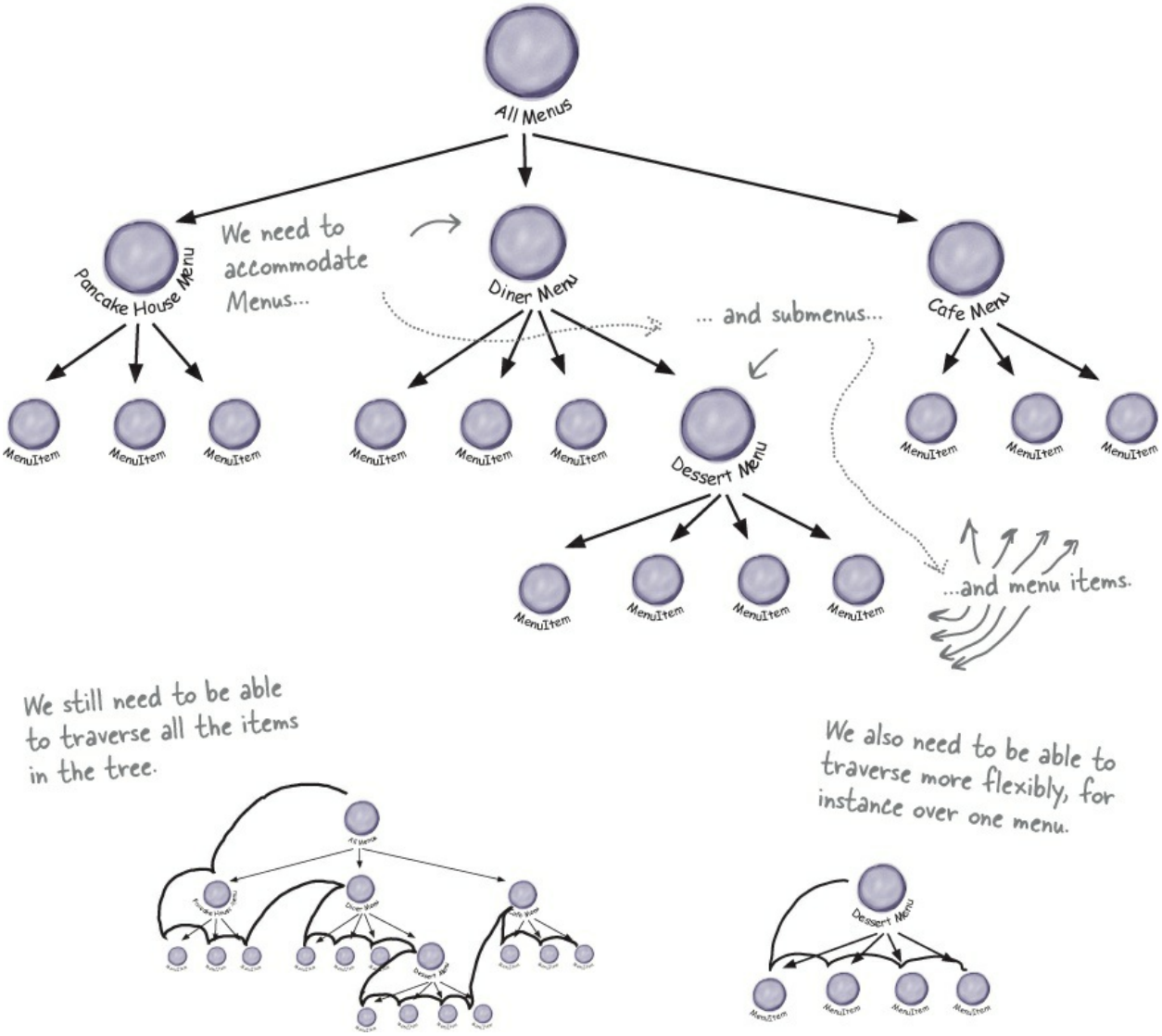
rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

So, what is it we really need out of our new design?

❑	We need some kind of a tree-shaped structure that will accommodate menus, submenus, and menu items.
❑	We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
❑	We may need to traverse the items in a more flexible manner. For instance, we might need to iterate over only the Diner's dessert menu, or we might need to iterate over the Diner's entire menu, including the dessert submenu.

NOTE

Because we need to represent menus, nested submenus and menu items, we can naturally fit them in a tree-like structure.



BRAIN POWER

How would you handle this new wrinkle to our design requirements? Think about it before turning the page.

The Composite Pattern defined

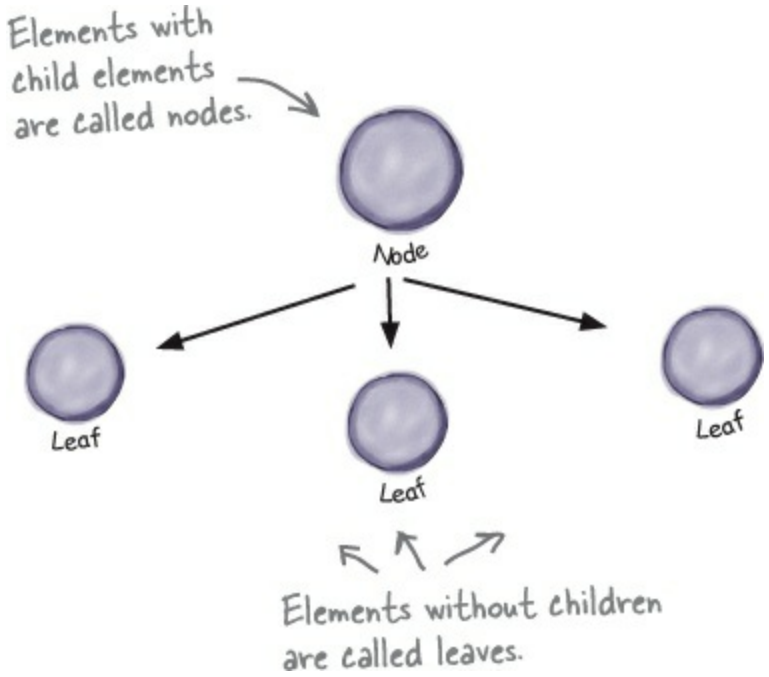
That’s right; we’re going to introduce another pattern to solve this problem. We didn’t give up on Iterator — it will still be part of our solution — however, the problem of managing menus has taken on a new dimension that Iterator doesn’t solve. So, we’re going to step back and solve it with the

Composite Pattern.

We’re not going to beat around the bush on this pattern; we’re going to go ahead and roll out the official definition now:

NOTE

Here’s a tree structure.



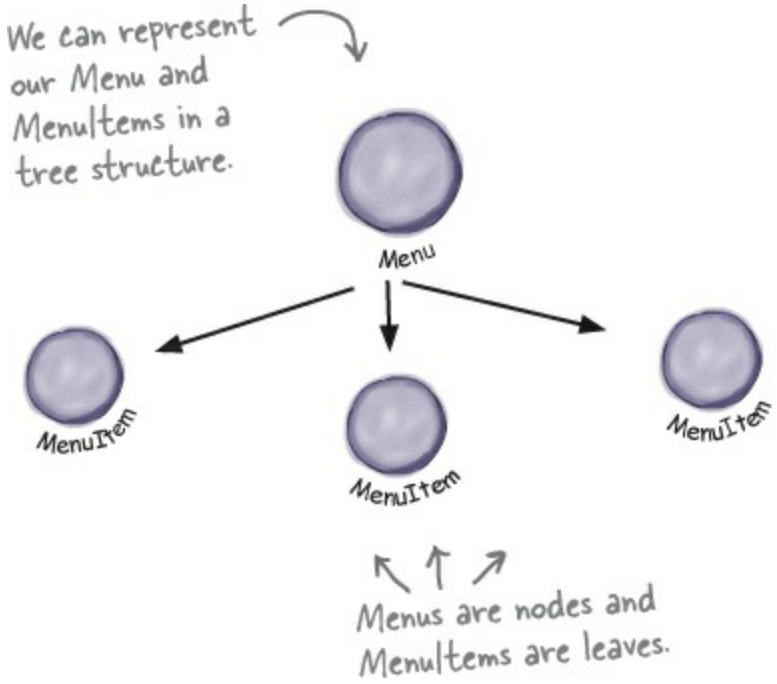
NOTE

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Let’s think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus *and* menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy; that is, a tree of objects that is made of parts (menus and menu items) but that can be treated as a whole, like one big über menu.

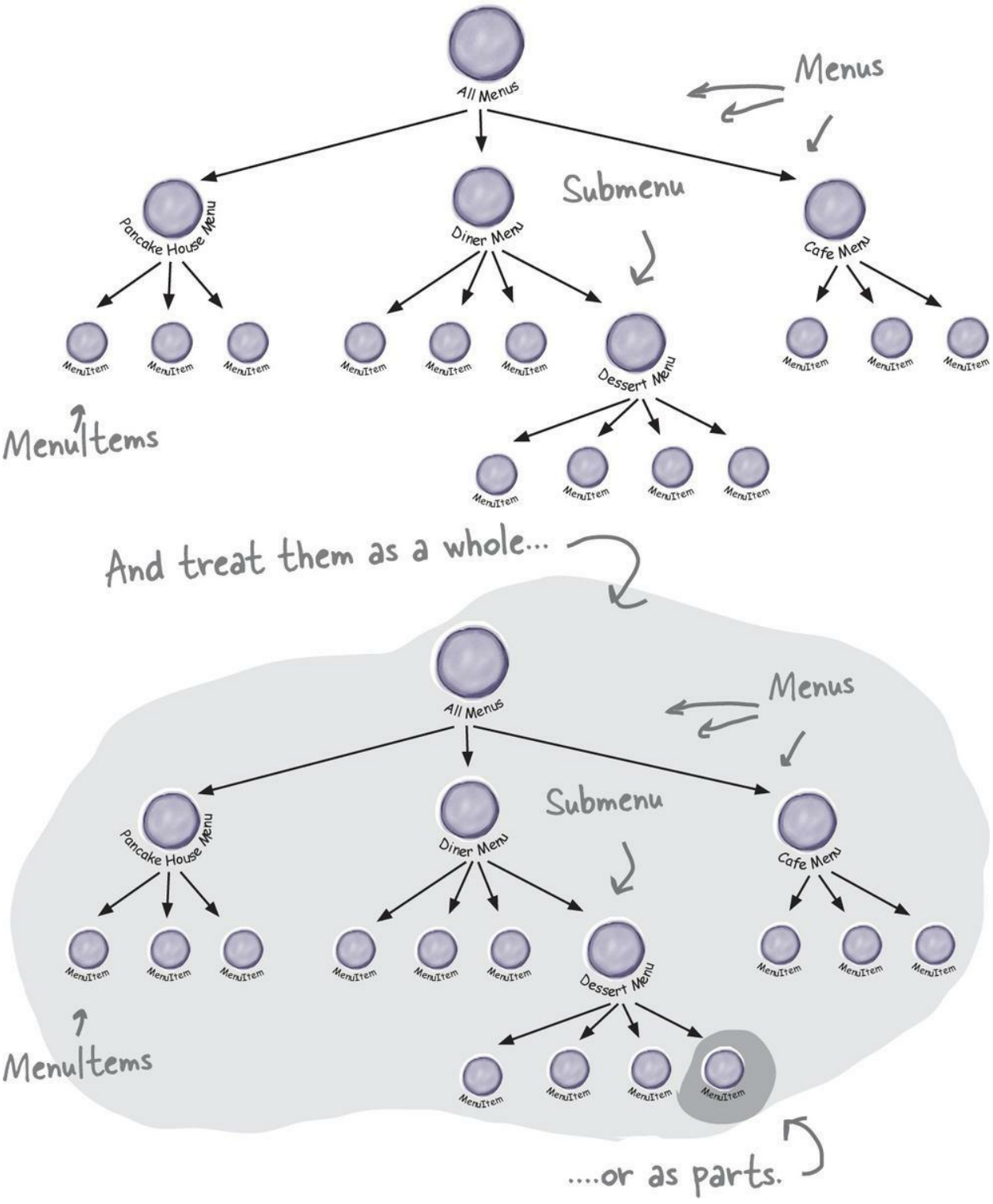
Once we have our über menu, we can use this pattern to treat “individual objects and compositions uniformly.” What does that mean? It means if we

have a tree structure of menus, submenus, and perhaps subsubmenus along with menu items, then any menu is a “composition” because it can contain both other menus and menu items. The individual objects are just the menu items — they don’t hold other objects. As you’ll see, using a design that follows the Composite Pattern is going to allow us to write some simple code that can apply the same operation (like printing!) over the entire menu structure.



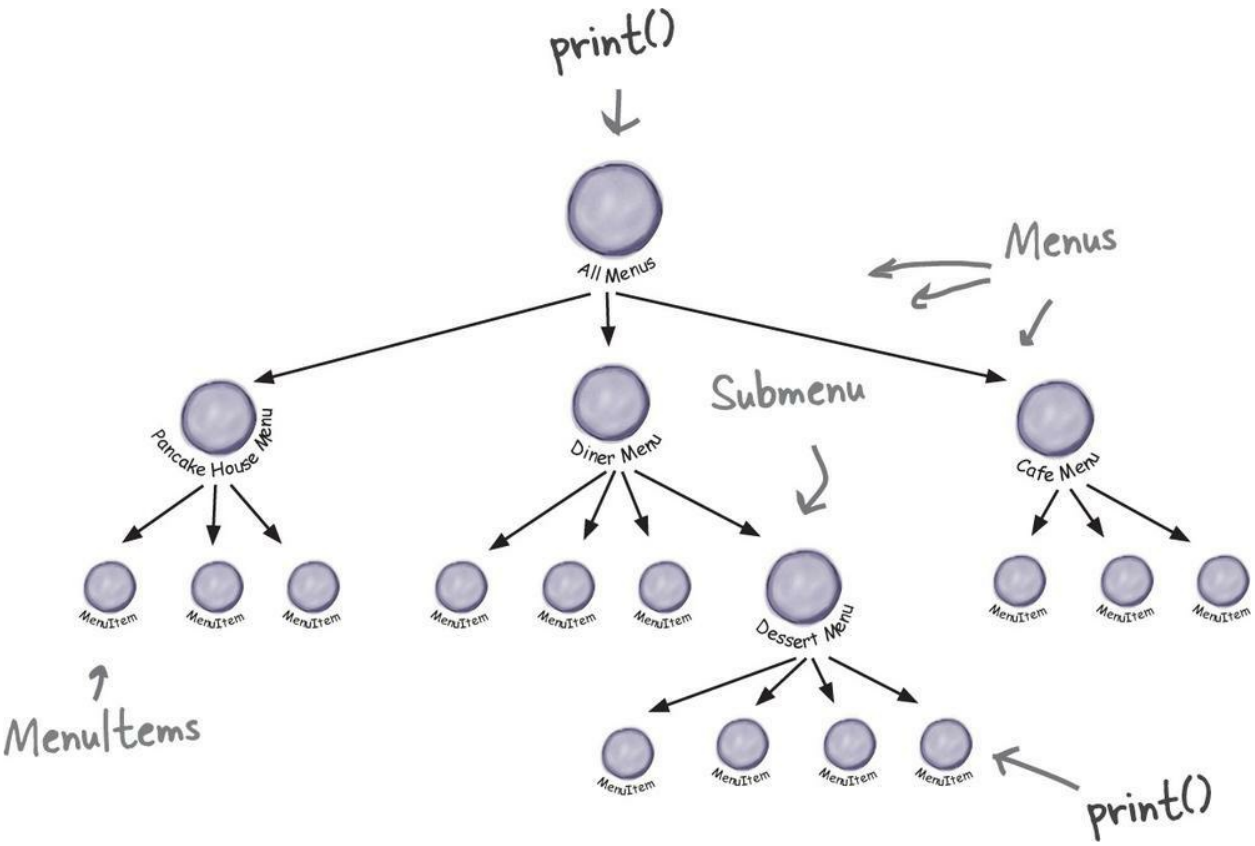
NOTE

We can create arbitrarily complex trees.



NOTE

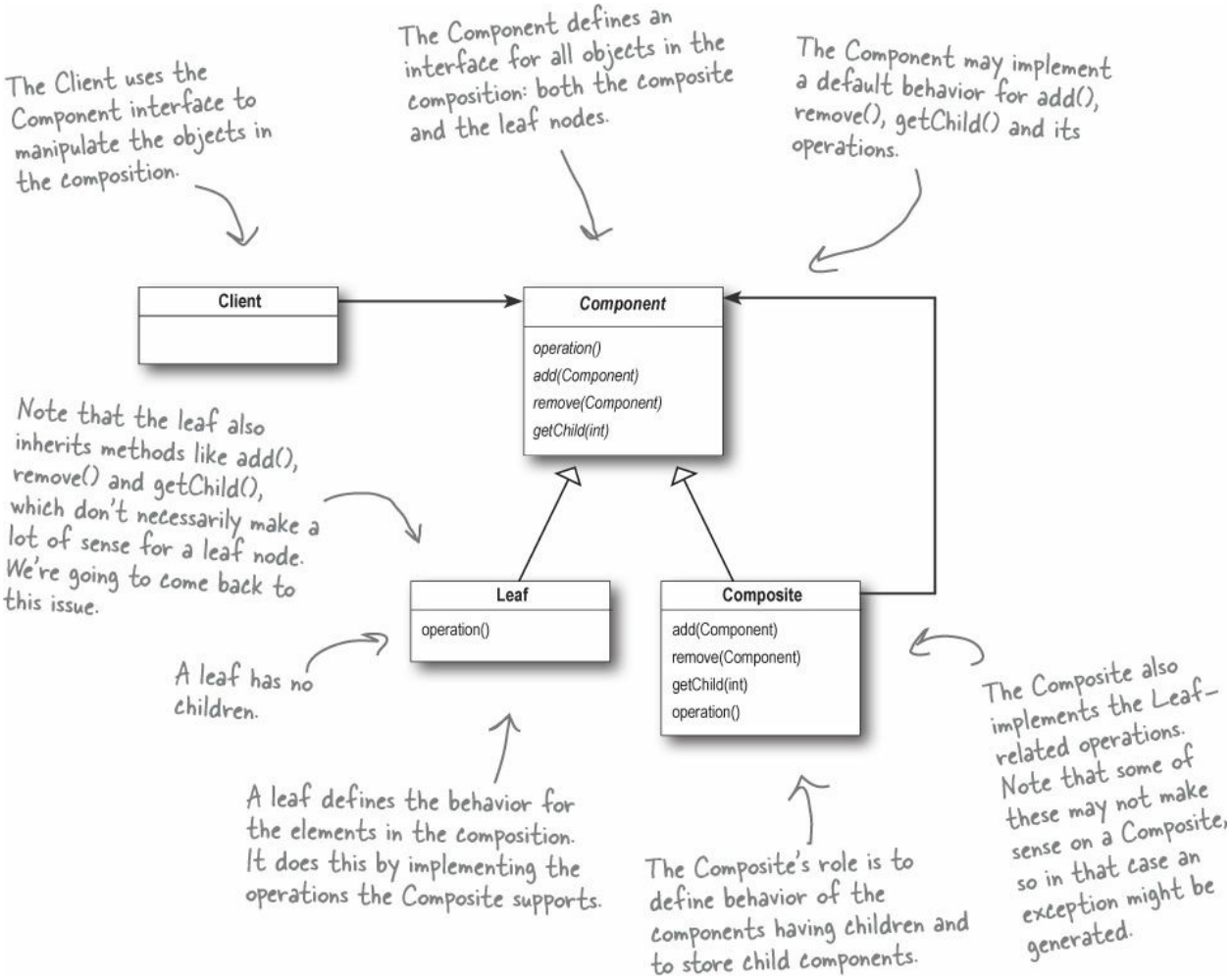
Operations can be applied to the whole.



NOTE

Or the parts.

The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes. Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can <u>ignore</u> the differences between compositions of objects and individual objects.



THERE ARE NO DUMB QUESTIONS

Q: **Q: Component, Composite, Trees? I'm confused.**

A: **A:** A composite contains components. Components come in two flavors: composites and leaf elements. Sound recursive? It is. A composite holds a set of children; those children may be other composites or leaf elements. When you organize data in this way you end up with a tree structure (actually an upside-down tree structure) with a composite at the root and branches of composites growing up to leaf nodes.

Q: **Q: How does this relate to iterators?**

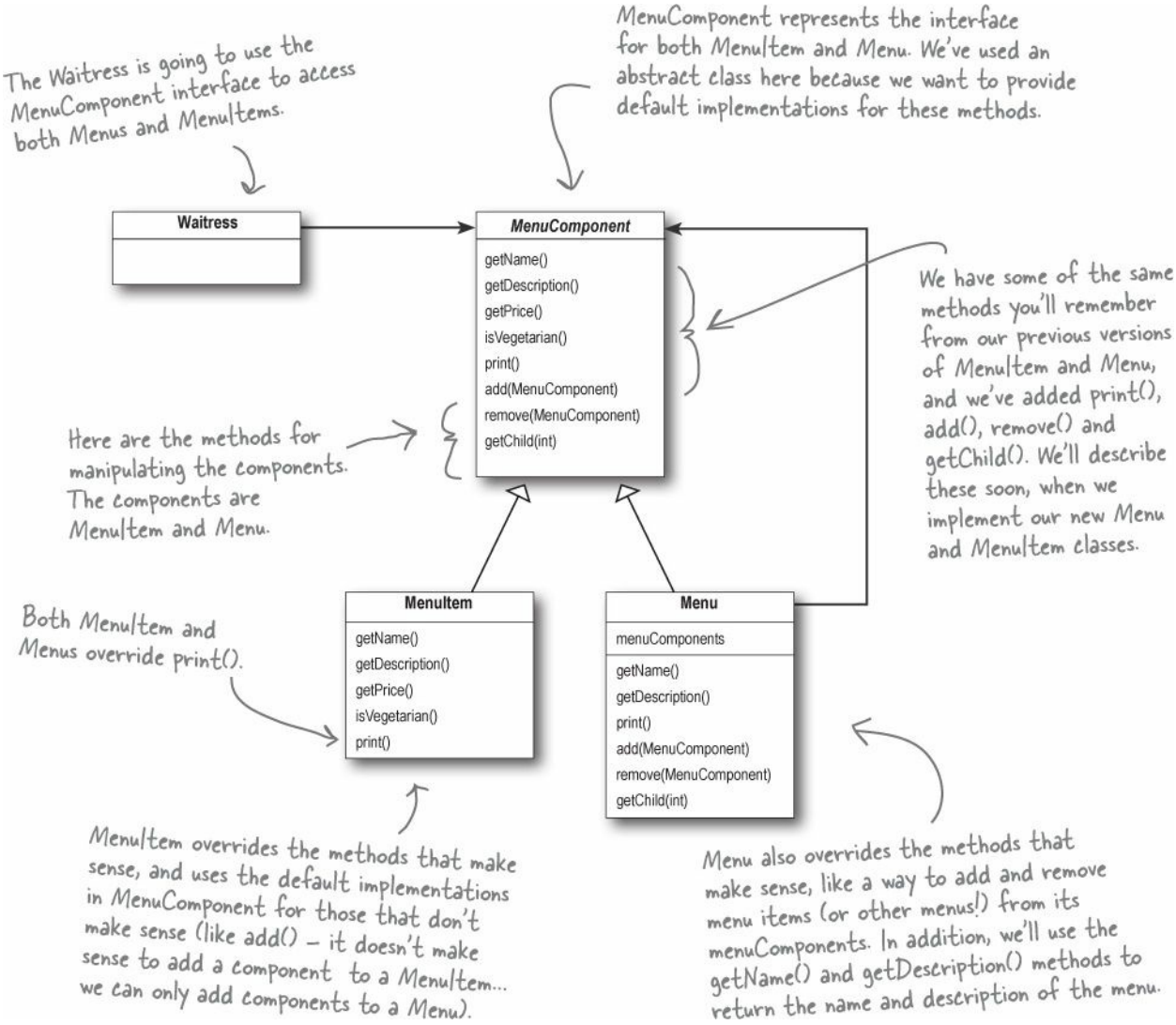
A: **A:** Remember, we're taking a new approach. We're going to re-implement the menus with a new solution: the Composite Pattern. So don't look for some magical transformation from an iterator to a composite. That said, the two work very nicely together. You'll soon see that we can use iterators in a couple of ways in the composite implementation.

Designing Menus with Composite

So, how do we apply the Composite Pattern to our menus? To start with, we need to create a component interface; this acts as the common interface for

both menus and menu items and allows us to treat them uniformly. In other words, we can call the *same* method on menus or menu items.

Now, it may not make *sense* to call some of the methods on a menu item or a menu, but we can deal with that, and we will in just a moment. But for now, let's take a look at a sketch of how the menus are going to fit into a Composite Pattern structure:



Implementing the Menu Component

Okay, we're going to start with the MenuComponent abstract class; remember, the role of the menu component is to provide an interface for the leaf nodes and the composite nodes. Now you might be asking, "Isn't the MenuComponent playing two roles?" It might well be and we'll come back

to that point. However, for now we're going to provide a default implementation of the methods so that if the MenuItem (the leaf) or the Menu (the composite) doesn't want to implement some of the methods (like getChild() for a leaf node) they can fall back on some basic behavior:

NOTE

All components must implement the MenuComponent interface; however, because leaves and nodes have different roles we can't always define a default implementation for each method that makes sense. Sometimes the best you can do is throw a runtime exception.

NOTE

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, they don't have to do anything; they can just inherit the default implementation.

MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

We've grouped together the "composite" methods – that is, methods to add, remove and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItems. It turns out we can also use a couple of them in Menu too, as you'll see in a couple of pages when we show the Menu code.

print() is an "operation" method that both our Menus and MenuItems will implement, but we provide a default operation here.

Implementing the Menu Item

Okay, let's give the MenuItem class a shot. Remember, this is the leaf class in the Composite diagram and it implements the behavior of the elements of the composite.


I'm glad we're going in this direction. I'm thinking this is going to give me the flexibility I need to implement that crêpe menu I've always wanted.

```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

First we need to extend the MenuComponent interface.

The constructor just takes the name, description, etc. and keeps a reference to them all. This is pretty much like our old menu item implementation.

Here's our getter methods – just like our previous implementation.

This is different from the previous implementation. Here we're overriding the print() method in the MenuComponent class. For MenuItem this method prints the complete menu entry: name, description, price and whether or not it's veggie.

Implementing the Composite Menu

Now that we have the MenuItem, we just need the composite class, which we're calling Menu. Remember, the composite class can hold MenuItems *or* other Menus. There's a couple of methods from MenuComponent this class doesn't implement: getPrice() and isVegetarian(), because those don't make a lot of sense for a Menu.

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        returnmenuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");
    }
}
```

Menu is also a MenuComponent, just like MenuItem.

Menu can have any number of children of type MenuComponent. We'll use an internal ArrayList to hold these.

This is different than our old implementation: we're going to give each Menu a name and a description. Before, we just relied on having different classes for each menu.

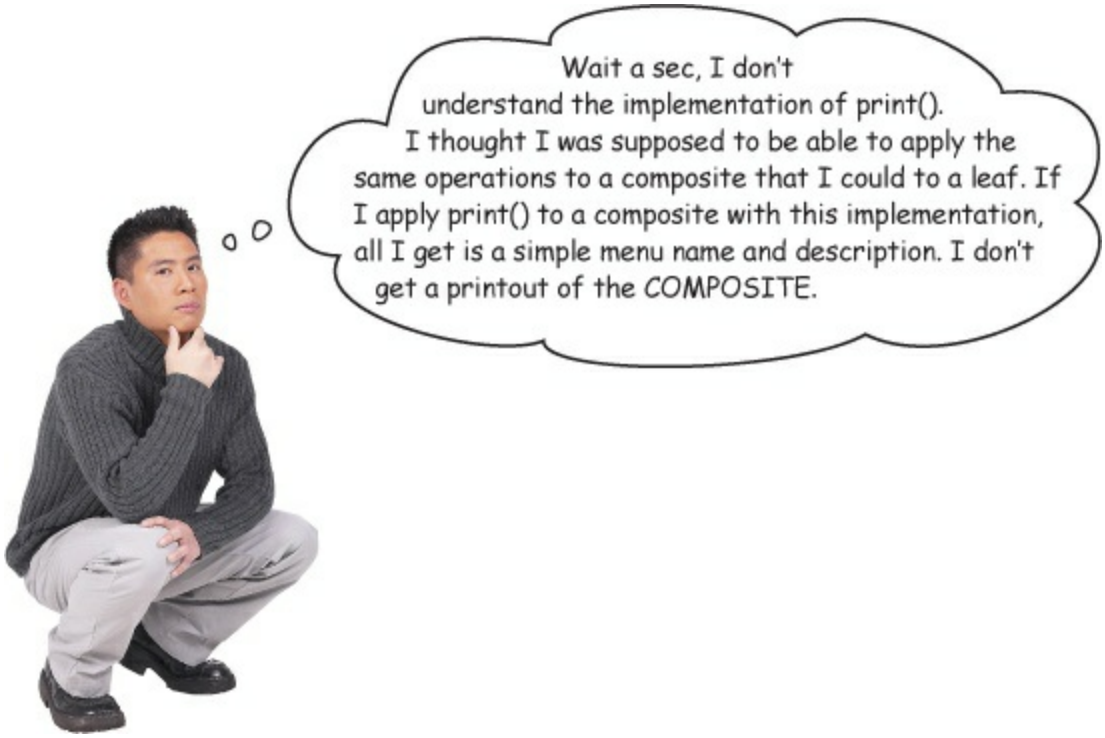
Here's how you add MenuItems or other Menus to a Menu. Because both MenuItems and Menus are MenuComponents, we just need one method to do both.

You can also remove a MenuComponent or get a MenuComponent.

Here are the getter methods for getting the name and description.

Notice, we aren't overriding getPrice() or isVegetarian() because those methods don't make sense for a Menu (although you could argue that isVegetarian() might make sense). If someone tries to call those methods on a Menu, they'll get an UnsupportedOperationException.

To print the Menu, we print the Menu's name and description.



Excellent catch. Because menu is a composite and contains both MenuItems and other Menus, its print() method should print everything it contains. If it didn't we'd have to iterate through the entire composite and print each item ourselves. That kind of defeats the purpose of having a composite structure.

As you're going to see, implementing print() correctly is easy because we can rely on each component to be able to print itself. It's all wonderfully recursive and groovy. Check it out:

Fixing the print() method

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator<MenuComponent> iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                                iterator.next();
            menuComponent.print();
        }
    }
}
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Look! We get to use an Iterator. We use it to iterate through all the Menu's components... those could be other Menus, or they could be MenuItems.

Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

Getting ready for a test drive...

It's about time we took this code for a test drive, but we need to update the Waitress code before we do — after all she's the main client of this code:

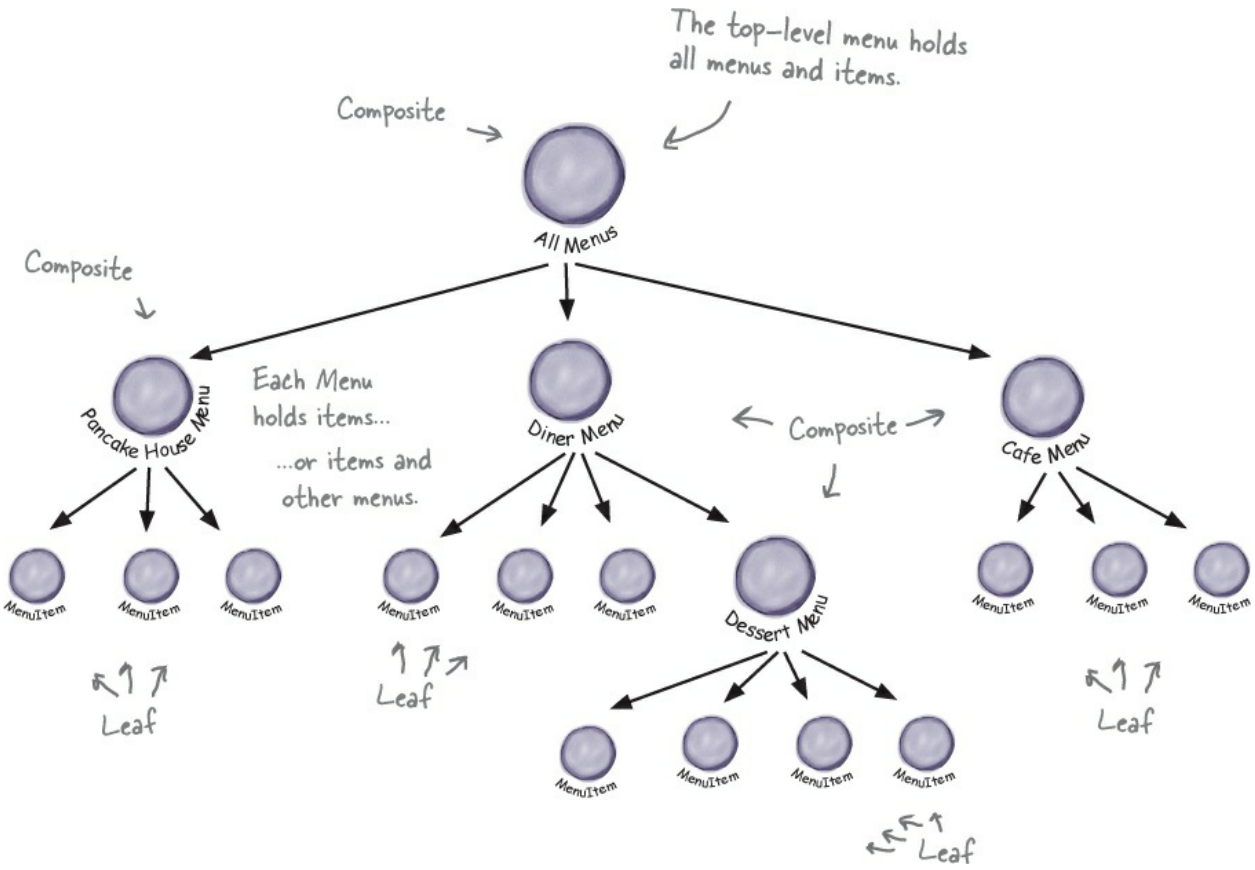
```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Yup! The Waitress code really is this simple. Now we just hand her the top-level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy – all the menus, and all the menu items – is call print() on the top level menu.

We're gonna have one happy Waitress.

Okay, one last thing before we write our test drive. Let's get an idea of what the menu composite is going to look like at runtime:



Now for the test drive...

Okay, now we just need a test drive. Unlike our previous version, we're going to handle all the menu creation in the test drive. We could ask each chef to give us his new menu, but let's get it all tested first. Here's the code:

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need a top-level menu that we'll name allMenus.

We're using the Composite add() method to add each menu to the top-level menu, allMenus.

Now we need to add all the menu items. Here's one example; for the rest, look at the complete source code.

And we're also adding a menu to a menu. All dinerMenu cares about is that everything it holds, whether it's a menu item or a menu, is a MenuComponent.

Add some apple pie to the dessert menu...

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's as easy as apple pie for her to print it out.

Getting ready for a test drive...

NOTE

NOTE: this output is based on the complete source.

```
File  Edit  Window  Help  GreenEggs&Spam
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Here's all our menus... we printed all this just by calling print() on the top level menu.

The new dessert menu is printed when we are printing all the Diner menu components.

There is some truth to that observation. We could say that the Composite Pattern takes the Single Responsibility design principle and trades it for *transparency*. What's transparency? Well, by allowing the Component interface to contain the child management operations *and* the leaf operations, a client can treat both composites and leaf nodes uniformly; so whether an element is a composite or leaf node becomes transparent to the client.

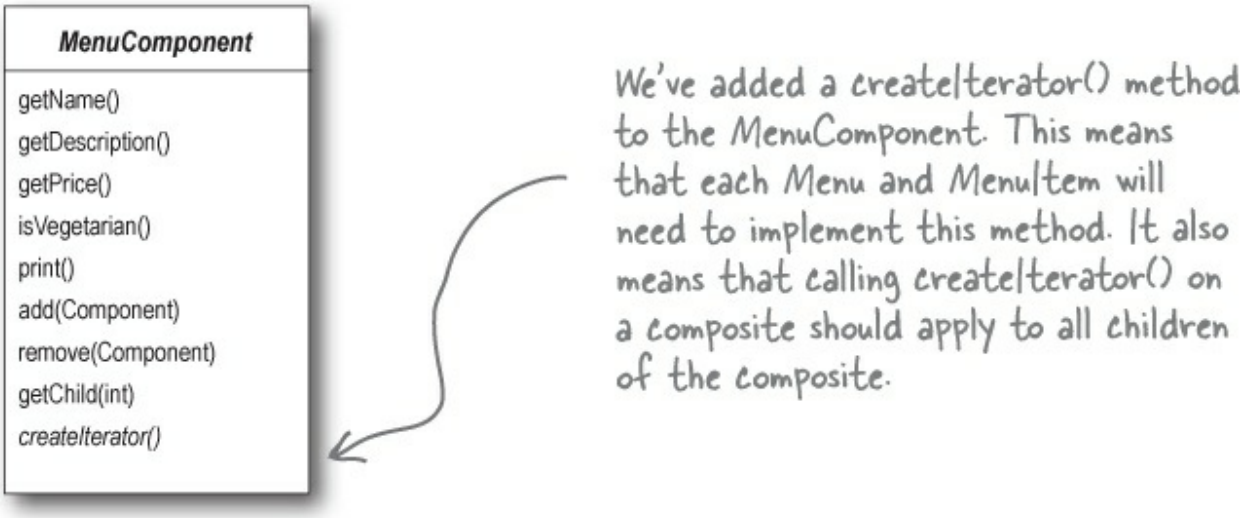
Now given we have both types of operations in the Component class, we lose a bit of *safety* because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate out the responsibilities into interfaces. This would make our design safe, in the sense that any inappropriate calls on elements would be caught at compile time or runtime, but we'd lose transparency and our code would have to use conditionals and the `instanceof` operator.

So, to return to your question, this is a classic case of tradeoff. We are guided by design principles, but we always need to observe the effect they have on our designs. Sometimes we purposely do things in a way that seems to violate the principle. In some cases, however, this is a matter of perspective; for instance, it might seem incorrect to have child management operations in the leaf nodes (like add(), remove() and getChild()), but then again you can always shift your perspective and see a leaf as a node with zero children.

Flashback to Iterator

We promised you a few pages back that we'd show you how to use Iterator with a Composite. You know that we are already using Iterator in our internal implementation of the print() method, but we can also allow the Waitress to iterate over an entire composite if she needs to — for instance, if she wants to go through the entire menu and pull out vegetarian items.

To implement a Composite iterator, let's add a createIterator() method in every component. We'll start with the abstract MenuComponent class:



Now we need to implement this method in the Menu and MenuItem classes:

```
public class Menu extends MenuComponent {
    Iterator<MenuComponent> iterator = null;
    // other code here doesn't change

    public Iterator<MenuComponent> createIterator() {
        if (iterator == null) {
            iterator = new CompositeIterator(menuComponents.iterator());
        }
        return iterator;
    }
}
public class MenuItem extends MenuComponent {
    // other code here doesn't change

    public Iterator<MenuComponent> createIterator() {
        return new NullIterator();
    }
}
```

Here we're using a new iterator called CompositeIterator. It knows how to iterate over any composite. We pass it the current composite's iterator.

Now for the MenuItem...

Whoa! What's this NullIterator? You'll see in two pages.

The Composite Iterator

The CompositeIterator is a SERIOUS iterator. It's got the job of iterating over the MenuItems in the component, and of making sure all the child Menus (and child child Menus, and so on) are included.

Here's the code. Watch out. This isn't a lot of code, but it can be a little mind bending. As you go through it just repeat to yourself "recursion is my friend, recursion is my friend."

WATCH OUT: RECURSION ZONE AHEAD

```
import java.util.*;

public class CompositeIterator implements Iterator {
    Stack<Iterator<MenuComponent>> stack = new Stack<Iterator<MenuComponent>>();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator<MenuComponent> iterator = stack.peek();
            MenuComponent component = iterator.next();

            stack.push(component.createIterator());

            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator<MenuComponent> iterator = stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }
}
```

Like all iterators, we're implementing the java.util.Iterator interface.

The iterator of the top-level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

If there is a next element, we get the current iterator off the stack and get its next element.

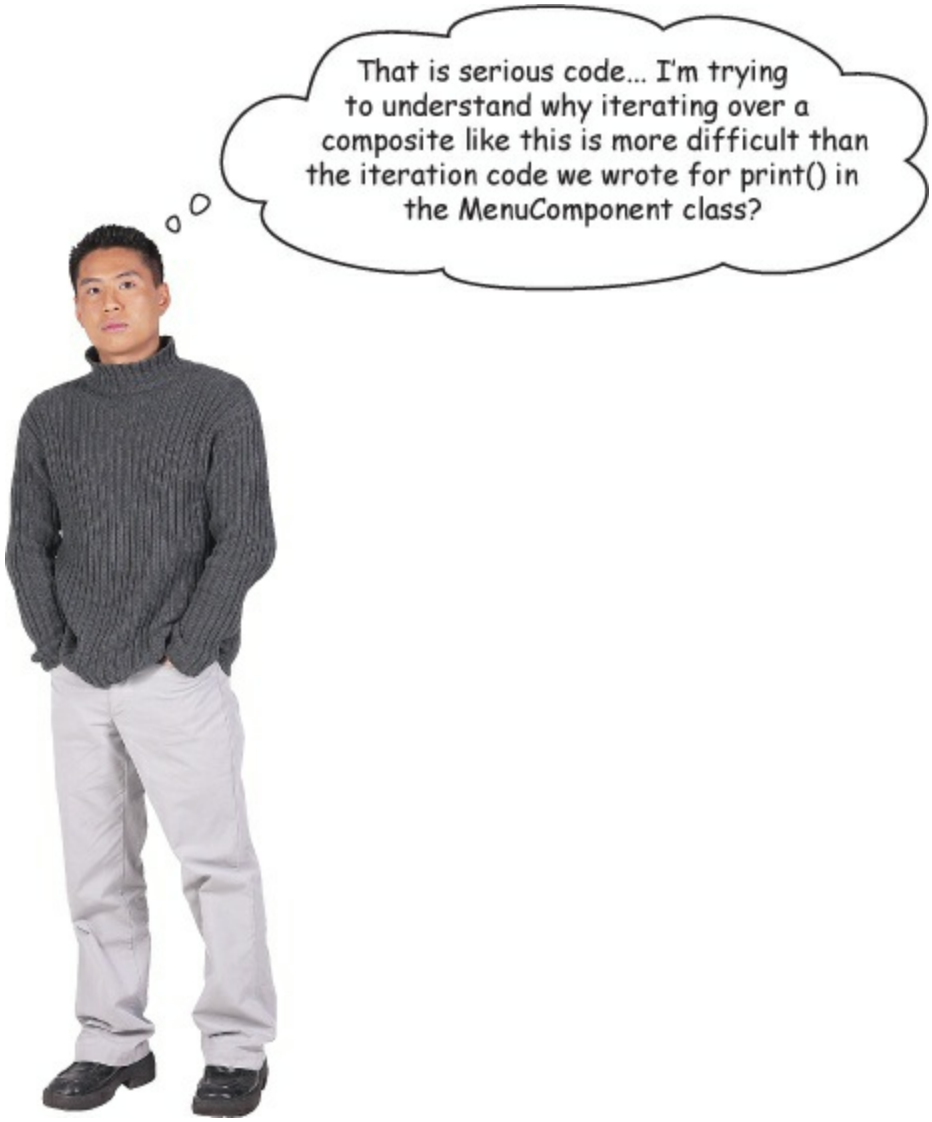
We then throw that component's iterator on the stack. If the component is a Menu, it will iterate over all its items. If the component is a MenuItem, we get the NullIterator, and no iteration happens. Then we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, so we don't implement it and leave it up to the default behavior in java.util.Iterator.



When we wrote the print() method in the MenuComponent class we used an iterator to step through each item in the component, and if that item was a Menu (rather than a MenuItem), then we recursively called the print() method to handle it. In other words, the MenuComponent handled the iteration itself, *internally*.

With this code we are implementing an *external* iterator so there is a lot more to keep track of. For starters, an external iterator must maintain its position in the iteration so that an outside client can drive the iteration by calling hasNext() and next(). But in this case, our code also needs to maintain that position over a composite, recursive structure. That's why we use stacks to maintain our position as we move up and down the composite hierarchy.

BRAIN POWER

Draw a diagram of the Menus and MenuItems. Then pretend you are the CompositeIterator, and your job is to handle calls to hasNext() and next(). Trace the way the CompositeIterator traverses the structure as this code is executed:

```
public void testCompositeIterator(MenuComponent component) {
       CompositeIterator iterator = new CompositeIterator(component.iterator);

       while(iterator.hasNext()) {
              MenuComponent component = iterator.next();
       }
}
```

The Null Iterator

Okay, now what is this Null Iterator all about? Think about it this way: a MenuItem has nothing to iterate over, right? So how do we handle the implementation of its createIterator() method? Well, we have two choices:

NOTE

NOTE: Another example of the Null Object "Design Pattern."

Choice one:

Return null
We could return null from createIterator(), but then we'd need conditional code in the client to see if null was returned or not.

Choice two:

Return an iterator that always returns false when hasNext() is called
This seems like a better plan. We can still return an iterator, but the client doesn't have to worry about whether or not null is ever returned. In effect, we're creating an iterator that is a "no op."

The second choice certainly seems better. Let's call it NullIterator and implement it.

This is the laziest Iterator you've ever seen. At every step of the way it punts.

```
import java.util.Iterator;

public class NullIterator implements <MenuComponent> {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

When next() is called, we return null.

Most importantly when hasNext() is called we always return false.

And the NullIterator wouldn't think of supporting remove. We don't need to implement this; we could leave it off and let the default java.util.Iterator remove handle it.

Give me the vegetarian menu

Now we've got a way to iterate over every item of the Menu. Let's take that and give our Waitress a method that can tell us exactly which items are vegetarian.

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator<MenuComponent> iterator = allMenus.createIterator();

        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

The printVegetarianMenu() method takes the allMenu's composite and gets its iterator. That will be our CompositeIterator.

Iterate through every element of the composite.

Call each element's isVegetarian() method and if true, we call its print() method.

print() is only called on MenuItems, never composites. Can you see why?

We implemented isVegetarian() on the Menus to always throw an exception. If that happens we catch the exception, but continue with our iteration.

The magic of Iterator & Composite together...

Whooo! It's been quite a development effort to get our code to this point. Now we've got a general menu structure that should last the growing Diner empire for some time. Now it's time to sit back and order up some veggie food:

```
File  Edit  Window  Help  HaveUhuggedYurIteratorToday?
% java MenuTestDrive
VEGETARIAN MENU
----
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla ice cream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

The Vegetarian Menu consists of the vegetarian items from every menu.

Let's take a look at what you're talking about:

```
try {
    if (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} catch (UnsupportedOperationException) {}
```

We call isVegetarian() on all MenuComponents, but Menus throw an exception because they don't support the operation.

If the menu component doesn't support the operation, we just throw away the exception and ignore it.

In general we agree; try/catch is meant for error handling, not program logic. What are our other options? We could have checked the runtime type of the menu component with instanceof to make sure it's a MenuItem before making the call to isVegetarian(). But in the process we'd lose *transparency* because we wouldn't be treating Menus and MenuItems uniformly.

We could also change isVegetarian() in the Menus so that it returns false. This provides a simple solution and we keep our transparency.

In our solution we are going for clarity: we really want to communicate that this is an unsupported operation on the Menu (which is different than saying isVegetarian() is false). It also allows for someone to come along and actually implement a reasonable isVegetarian() method for Menu and have it work with the existing code.

That's our story and we're stickin' to it.

PATTERNS EXPOSED

This week's interview: The Composite Pattern, on implementation issues

HeadFirst: We're here tonight speaking with the Composite Pattern. Why don't you tell us a little about yourself, Composite?

Composite: Sure... I'm the pattern to use when you have collections of objects with whole-part relationships and you want to be able to treat those objects uniformly.

HeadFirst: Okay, let's dive right in here... what do you mean by whole-part relationships?

Composite: Imagine a graphical user interface; there you'll often find a top level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the components that contain other components, *composite objects*, and components that don't contain other components, *leaf objects*.

HeadFirst: Is that what you mean by treating the objects uniformly? Having common methods you can call on composites and leaves?

Composite: Right. I can tell a composite object to display or a leaf object to display and it will do the right thing. The composite object will display by telling all its components to display.

HeadFirst: That implies that every object has the same interface. What if you have objects in your composite that do different things?

Composite: In order for the composite to work transparently to the client, you must implement the same interface for all objects in the composite; otherwise, the client has to worry about which interface each object is implementing, which kind of defeats the purpose. Obviously that means that at times you'll have objects for which some of the method calls don't make sense.

HeadFirst: So how do you handle that?

Composite: Well, there are a couple of ways to handle it; sometimes you can just do nothing, or return null or false — whatever makes sense in your application. Other times

you'll want to be more proactive and throw an exception. Of course, then the client has to be willing to do a little work and make sure that the method call didn't do something unexpected.

HeadFirst: But if the client doesn't know which kind of object they're dealing with, how would they ever know which calls to make without checking the type?

Composite: If you're a little creative you can structure your methods so that the default implementations do something that does make sense. For instance, if the client is calling getChild(), on the composite this makes sense. And it makes sense on a leaf too, if you think of the leaf as an object with no children.

HeadFirst: Ah... smart. But, I've heard some clients are so worried about this issue, that they require separate interfaces for different objects so they aren't allowed to make nonsensical method calls. Is that still the Composite Pattern?

Composite: Yes. It's a much safer version of the Composite Pattern, but it requires the client to check the type of every object before making a call so the object can be cast correctly.

HeadFirst: Tell us a little more about how these composite and leaf objects are structured.

Composite: Usually it's a tree structure, some kind of hierarchy. The root is the top-level composite, and all its children are either composites or leaf nodes.

HeadFirst: Do children ever point back up to their parents?

Composite: Yes, a component can have a pointer to a parent to make traversal of the structure easier. And, if you have a reference to a child, and you need to delete it, you'll need to get the parent to remove the child. Having the parent reference makes that easier too.

HeadFirst: There's really quite a lot to consider in your implementation. Are there other issues we should think about when implementing the Composite Pattern?

Composite: Actually there are... one is the ordering of children. What if you have a composite that needs to keep its children in a particular order? Then you'll need a more sophisticated management scheme for adding and removing children, and you'll have to be careful about how you traverse the hierarchy.

HeadFirst: A good point I hadn't thought of.

Composite: And did you think about caching?

HeadFirst: Caching?

Composite: Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores the result temporarily to save

traversals.

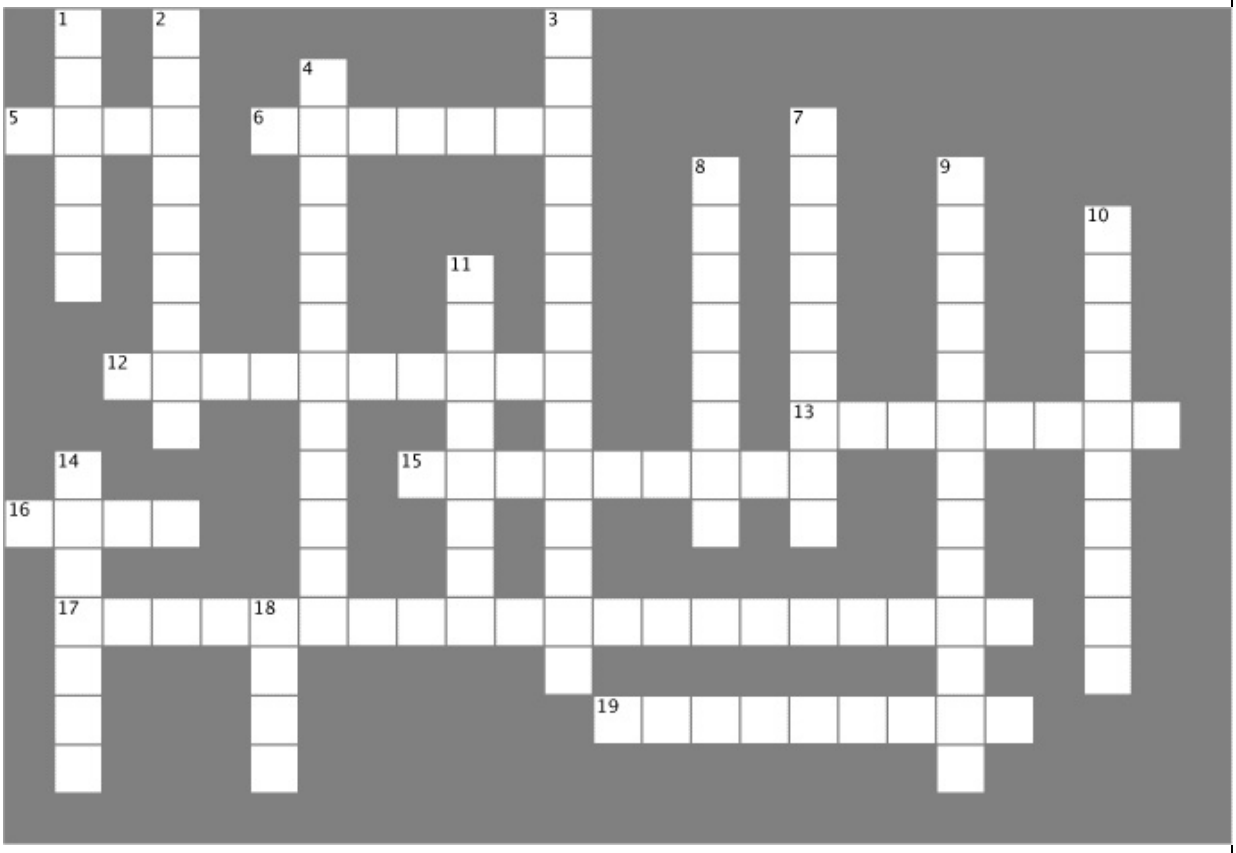
HeadFirst: Well, there's a lot more to the Composite Patterns than I ever would have guessed. Before we wrap this up, one more question: what do you consider your greatest strength?

Composite: I think I'd definitely have to say simplifying life for my clients. My clients don't have to worry about whether they're dealing with a composite object or a leaf object, so they don't have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

HeadFirst: That does sound like an important benefit. There's no doubt you're a useful pattern to have around for collecting and managing objects. And, with that, we're out of time... Thanks so much for joining us and come back soon for another Patterns Exposed.

DESIGN PATTERNS CROSSWORD

Wrap your brain around this composite crossword.



Across	Down
5. Third company acquired.	1. A class should have only one reason to do

6. This class indirectly supports Iterator. 12. HashMap and ArrayList both implement this interface. 13. A separate object that can traverse a collection. 15. We deleted PancakeHouseMenuIterator because this class already provides an Iterator. 16. Has no children. 17. Name of principle that states only one responsibility per class (two words). 19. CompositeIterator used a lot of this.	this. 2. We encapsulated this. 3. The Iterator Pattern decouples the client from the aggregate's _____. 4. Merged with the Diner (two words). 7. User interface packages often use this pattern for their components. 8. Collection and Iterator are in this package. 9. Iterators are usually created using this pattern (two words). 10. A composite holds this. 11. We Java-enabled her. 14. This menu caused us to change our entire implementation. 18. A component can be a composite or this.

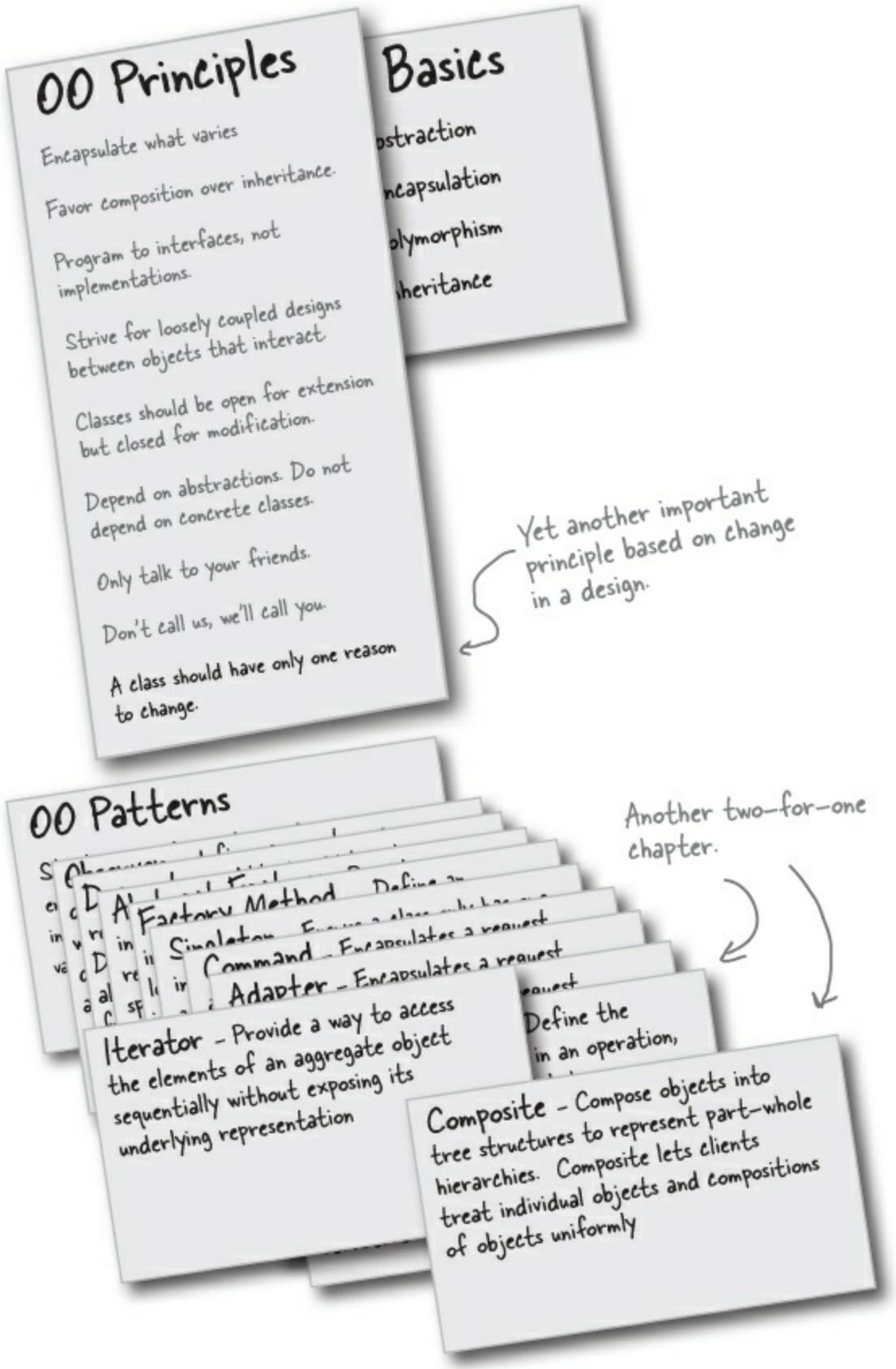
WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Strategy	Clients treat collections of objects and individual objects uniformly
Adapter	Provides a way to traverse a collection of objects without exposing the collection's implementation
Iterator	Simplifies the interface of a group of classes
Facade	Changes the interface of one or more classes
Composite	Allows a group of objects to be notified when some state changes
Observer	Encapsulates interchangeable behaviors and uses delegation to decide which one to use

Tools for your Design Toolbox

Two new patterns for your toolbox — two great ways to deal with collections of objects.


OO Principles
Encapsulate what varies
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
Basics
ostraction
ncapsulation
olymorphism
heritance
Yet another important principle based on change in a design.
OO Patterns
Another two-for-one chapter.
Factory Method
Command - Encapsulates a request
Adapter - Encapsulates a request
Iterator - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
Define the
in an operation,
Composite - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

BULLET POINTS

- An Iterator allows access to an aggregate's elements without exposing its internal structure.
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
- An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.
- We should strive to assign only one responsibility to each class.
- The Composite Pattern provides a structure to hold both individual objects and composites.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
- There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

SHARPEN YOUR PENCIL SOLUTION

Based on our implementation of printMenu(), which of the following apply?

☑	A.	We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
☐	B.	The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a standard.
☑	C.	If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.
☑	D.	The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
☑	E.	We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
☐	F.	The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

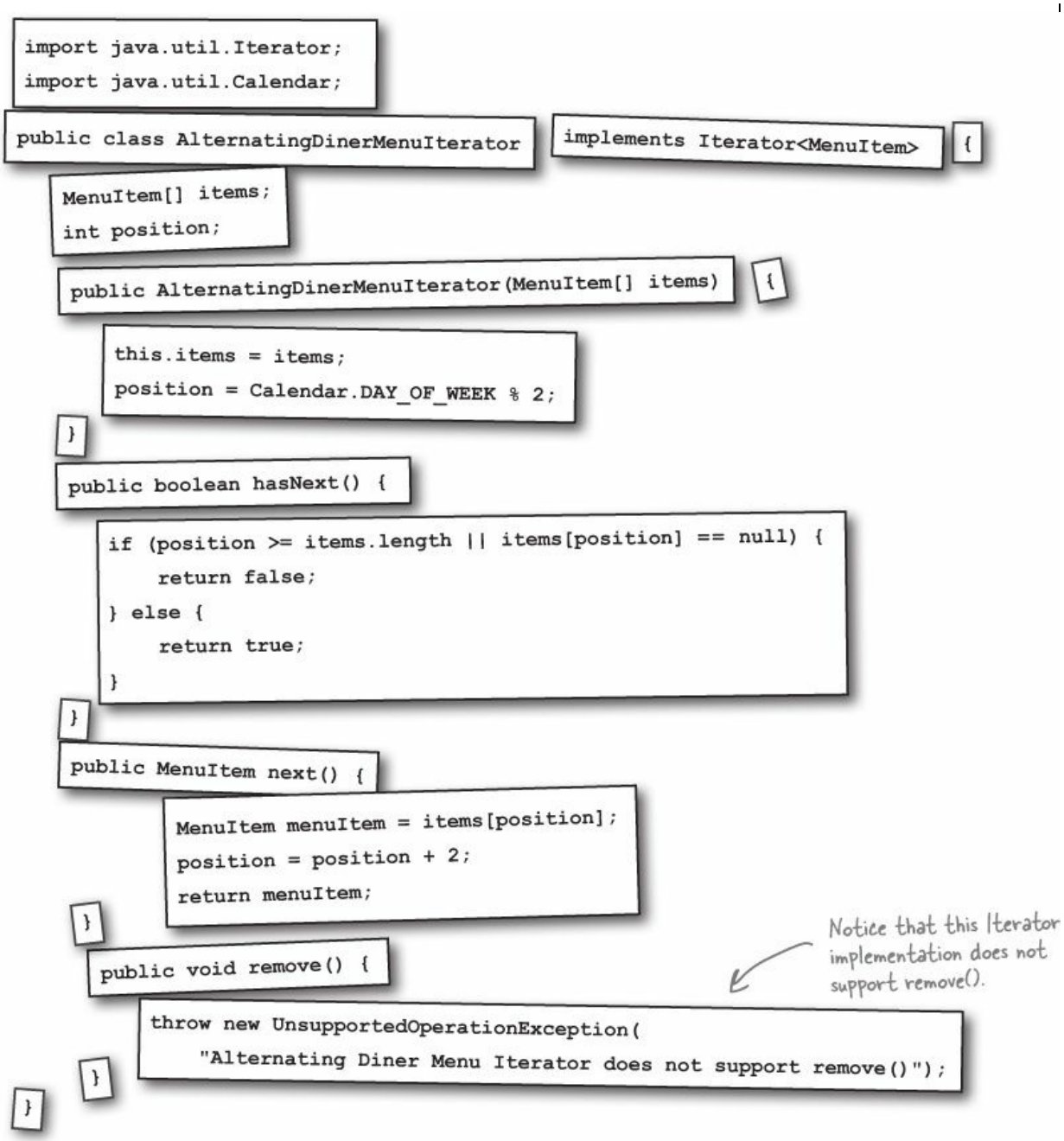
SHARPEN YOUR PENCIL SOLUTION

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. implement the Menu interface
2. get rid of getItems()
3. add createIterator() and return an Iterator that can step through the Hashtable values

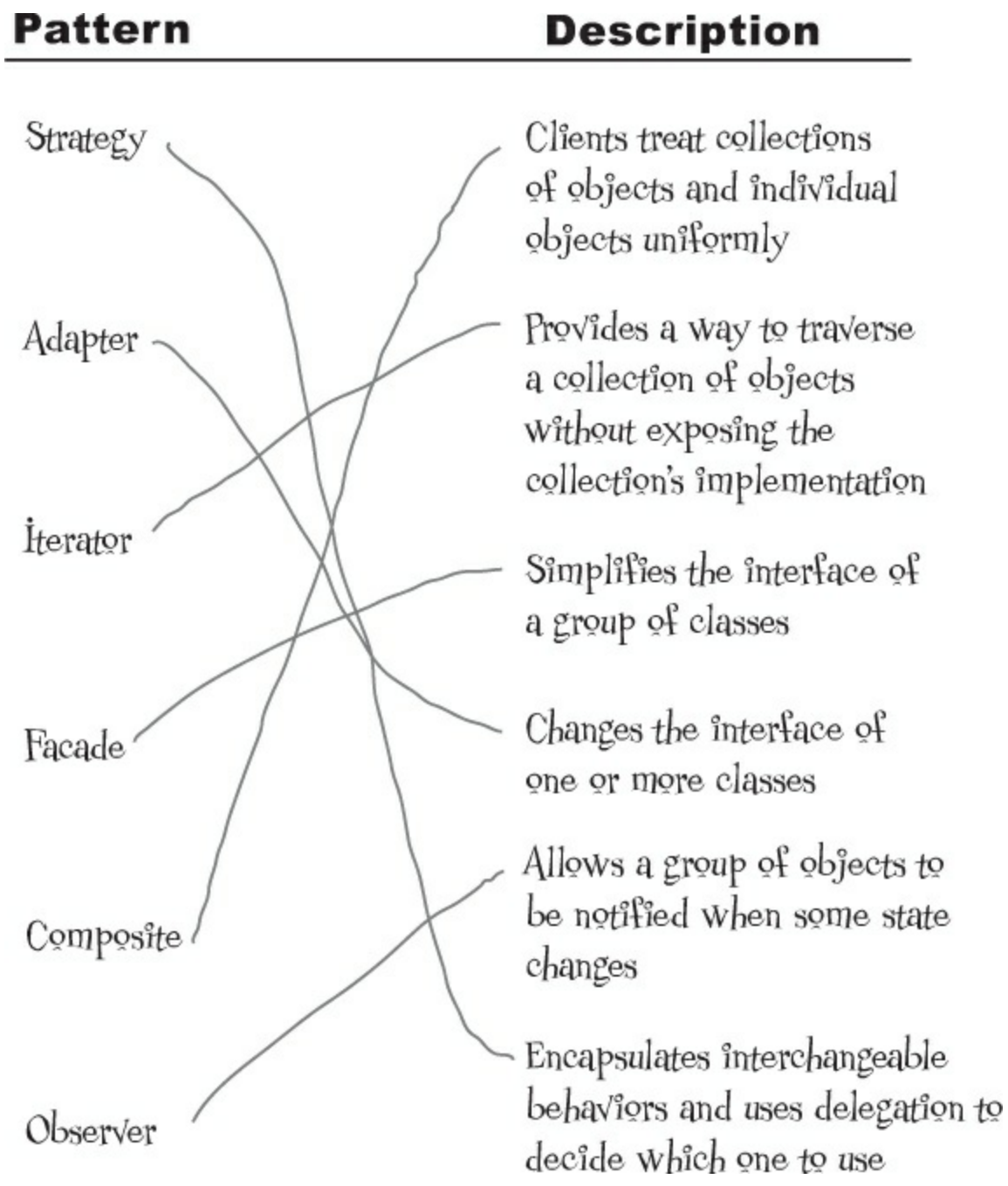
CODE MAGNETS SOLUTION

The unscrambled “Alternating” DinerMenu Iterator.



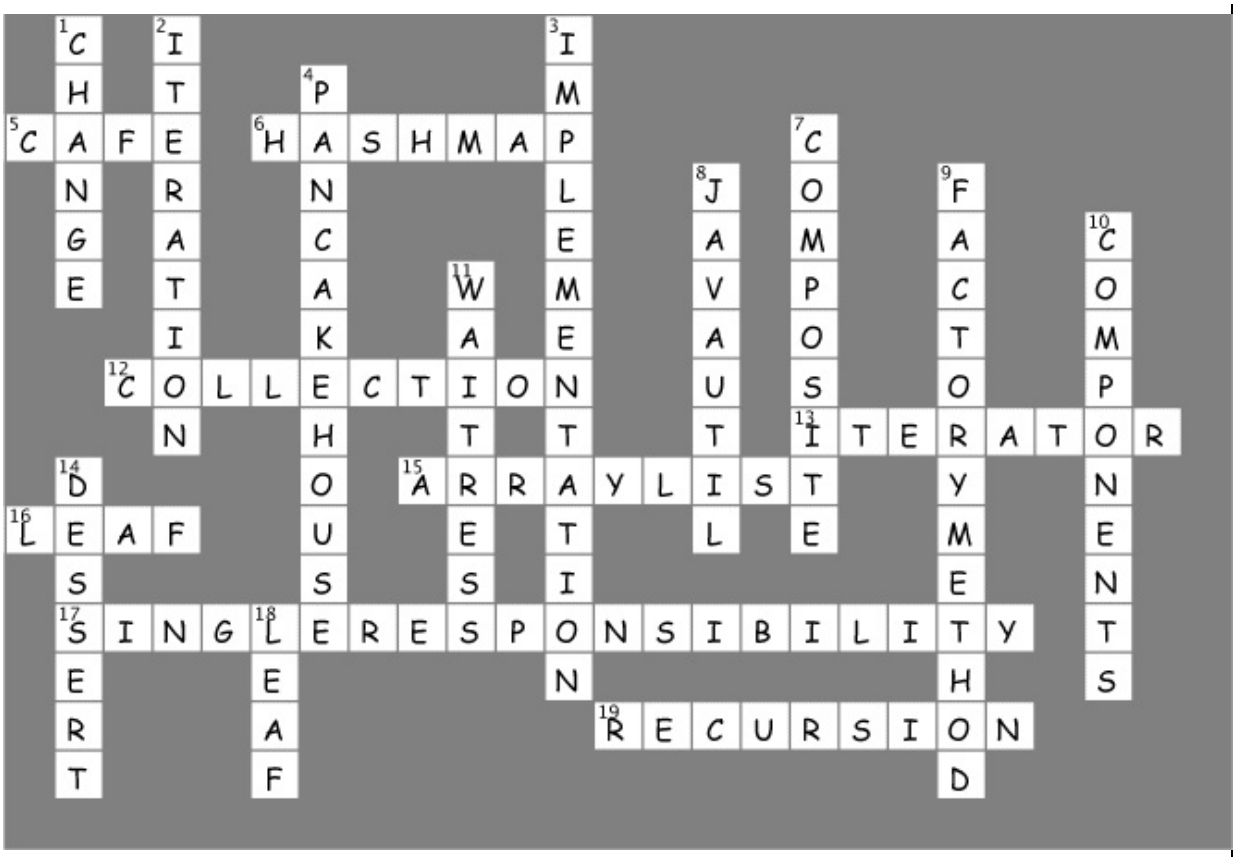
WHO DOES WHAT? SOLUTION

Match each pattern with its description:



DESIGN PATTERNS CROSSWORD SOLUTION

Wrap your brain around this composite crossword. Here's our solution.


1 C
2 I
3 I
H
T
4 P
M
5 C A F E
6 H A S H M A P
7 C
N
R
N
L
8 J
O
9 F
G
A
C
E
A
M
A
10 C
E
T
A
11 W
M
V
P
C
O
I
K
A
E
A
O
T
M
12 C O L L E C T I O N
U
S
O
P
N
H
T
T
T
13 I T E R A T O R
14 D
O
15 A R R A Y L I S T
Y
N
16 L E A F
U
E
T
L
E
M
E
S
S
S
I
E
N
17 S I N G
18 L E R E S P O N S I B I L I T Y
T
E
E
N
H
S
R
A
19 R E C U R S I O N
T
F
D

Chapter 10. The State Pattern: The State of Things

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

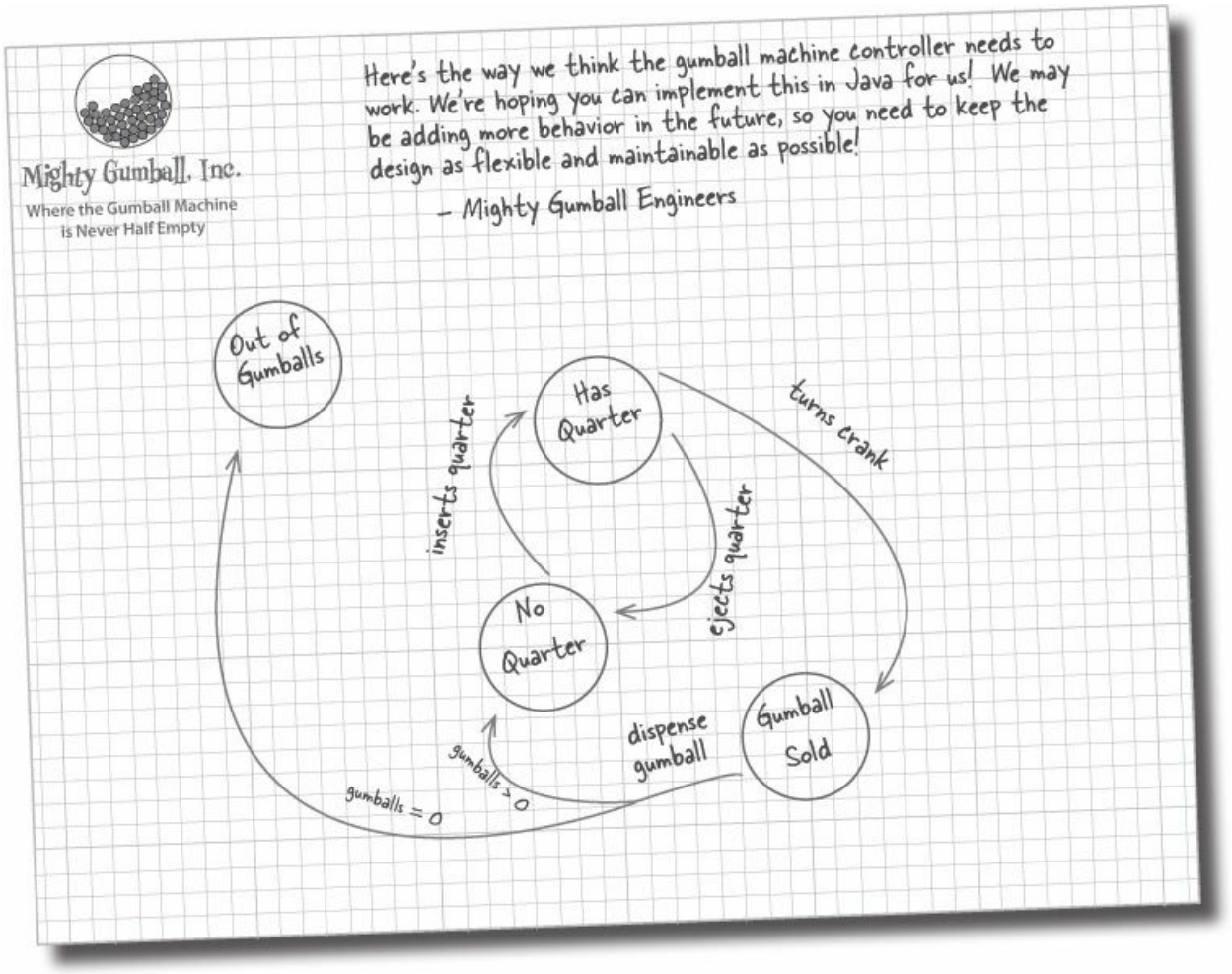
Jawva Breakers

Java toasters are so ’90s. Today people are building Java into *real* devices, like gumball machines. That’s right, gumball machines have gone high tech; the major manufacturers have found that by putting CPUs into their machines, they can increase sales, monitor inventory over the network and measure customer satisfaction more accurately.

NOTE

At least that’s their story – we think they just got bored with the circa 1800’s technology and needed to find a way to make their jobs more exciting.

But these manufacturers are gumball machine experts, not software developers, and they’ve asked for your help:



Cubicle Conversation

Judy: This diagram looks like a state diagram.

Joe: Right, each of those circles is a state...

Judy: ... and each of the arrows is a state transition.

Frank: Slow down, you two, it's been too long since I studied state diagrams. Can you remind me what they're all about?

Judy: Sure, Frank. Look at the circles; those are states. "No Quarter" is probably the starting state for the gumball machine because it's just sitting there waiting for you to put your quarter in. All states are just different configurations of the machine that behave in a certain way and need some action to take them to another state.

Joe: Right. See, to go to another state, you need to do something like put a quarter in the machine. See the arrow from "No Quarter" to "Has Quarter"?

Frank: Yes...

Joe: That just means that if the gumball machine is in the "No Quarter" state and you put a quarter in, it will change to the "Has Quarter" state. That's the state transition.

Frank: Oh, I see! And if I'm in the "Has Quarter" state, I can turn the crank

and change to the “Gumball Sold” state, or eject the quarter and change back to the “No Quarter” state.

Judy: You got it!

Frank: This doesn’t look too bad then. We’ve obviously got four states, and I think we also have four actions: “inserts quarter,” “ejects quarter,” “turns crank” and “dispense.” But... when we dispense, we test for zero or more gumballs in the “Gumball Sold” state, and then either go to the “Out of Gumballs” state or the “No Quarter” state. So we actually have five transitions from one state to another.

Judy: That test for zero or more gumballs also implies we’ve got to keep track of the number of gumballs too. Any time the machine gives you a gumball, it might be the last one, and if it is, we need to transition to the “Out of Gumballs” state.

Joe: Also, don’t forget that you could do nonsensical things, like try to eject the quarter when the gumball machine is in the “No Quarter” state, or insert two quarters.

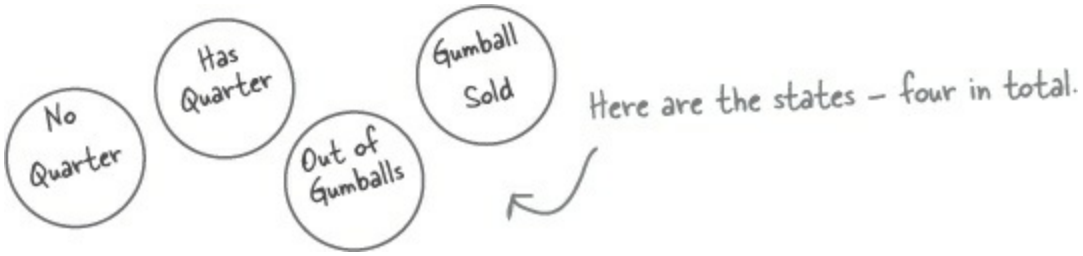
Frank: Oh, I didn’t think of that; we’ll have to take care of those too.

Joe: For every possible action we’ll just have to check to see which state we’re in and act appropriately. We can do this! Let’s start mapping the state diagram to code...

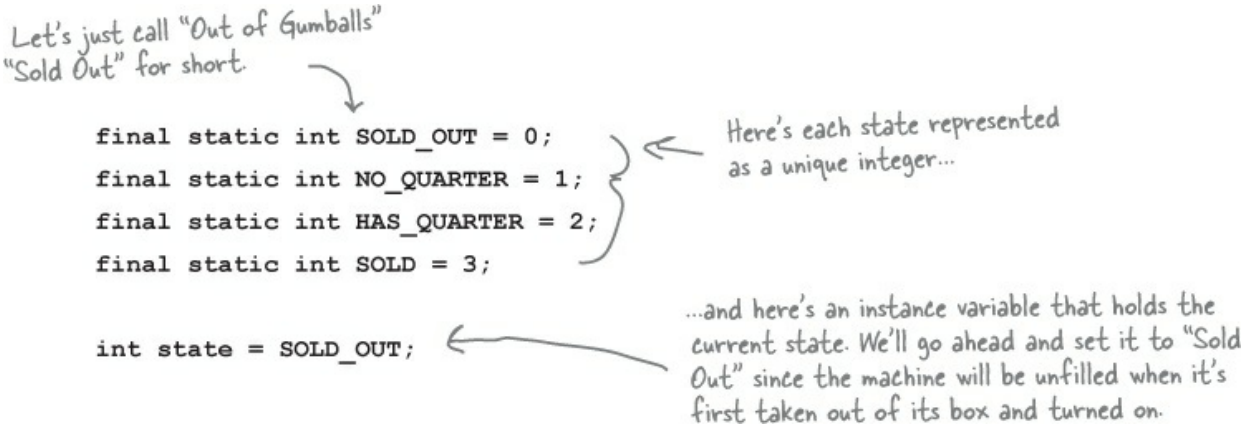
State machines 101

How are we going to get from that state diagram to actual code? Here’s a quick introduction to implementing state machines:

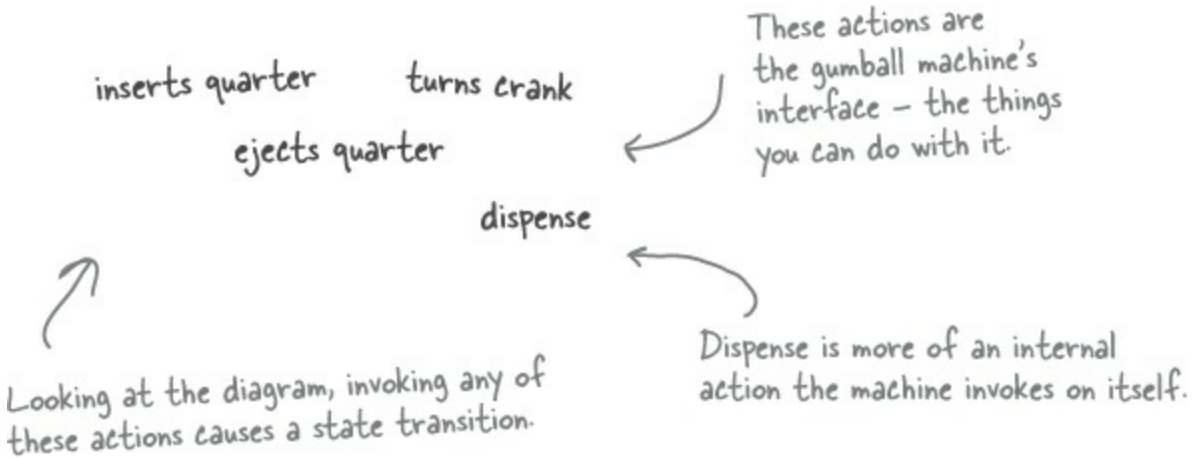
① First, gather up your states:



② Next, create an instance variable to hold the current state, and define values for each of the states:



③ Now we gather up all the actions that can happen in the system:



④ Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this:

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

With that quick review, let's go implement the Gumball Machine!

Writing the code

It’s time to implement the Gumball Machine. We know we’re going to have an instance variable that holds the current state. From there, we just need to handle all the actions, behaviors and state transitions that can happen. For actions, we need to implement inserting a quarter, removing a quarter, turning the crank, and dispensing a gumball; we also have the empty Gumball Machine condition to implement.


```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

```
    public void ejectQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("Quarter returned");
            state = NO_QUARTER;
        } else if (state == NO_QUARTER) {
            System.out.println("You haven't inserted a quarter");
        } else if (state == SOLD) {
            System.out.println("Sorry, you already turned the crank");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't eject, you haven't inserted a quarter yet");
        }
    }

    public void turnCrank() {
        if (state == SOLD) {
            System.out.println("Turning twice doesn't get you another gumball!");
        } else if (state == NO_QUARTER) {
            System.out.println("You turned but there's no quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You turned, but there are no gumballs");
        } else if (state == HAS_QUARTER) {
            System.out.println("You turned...");
            state = SOLD;
            dispense();
        }
    }

    public void dispense() {
        if (state == SOLD) {
            System.out.println("A gumball comes rolling out the slot");
            count = count - 1;
            if (count == 0) {
                System.out.println("Oops, out of gumballs!");
                state = SOLD_OUT;
            } else {
                state = NO_QUARTER;
            }
        } else if (state == NO_QUARTER) {
            System.out.println("You need to pay first");
        } else if (state == SOLD_OUT) {
            System.out.println("No gumball dispensed");
        } else if (state == HAS_QUARTER) {
            System.out.println("No gumball dispensed");
        }
    }

    // other methods here like toString() and refill()
}
```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

If the customer just turned the crank, we can't give a refund; he already has the gumball!

You can't eject if the machine is sold out, it doesn't accept quarters!

The customer tries to turn the crank...

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

We're in the SOLD state; give 'em a gumball!

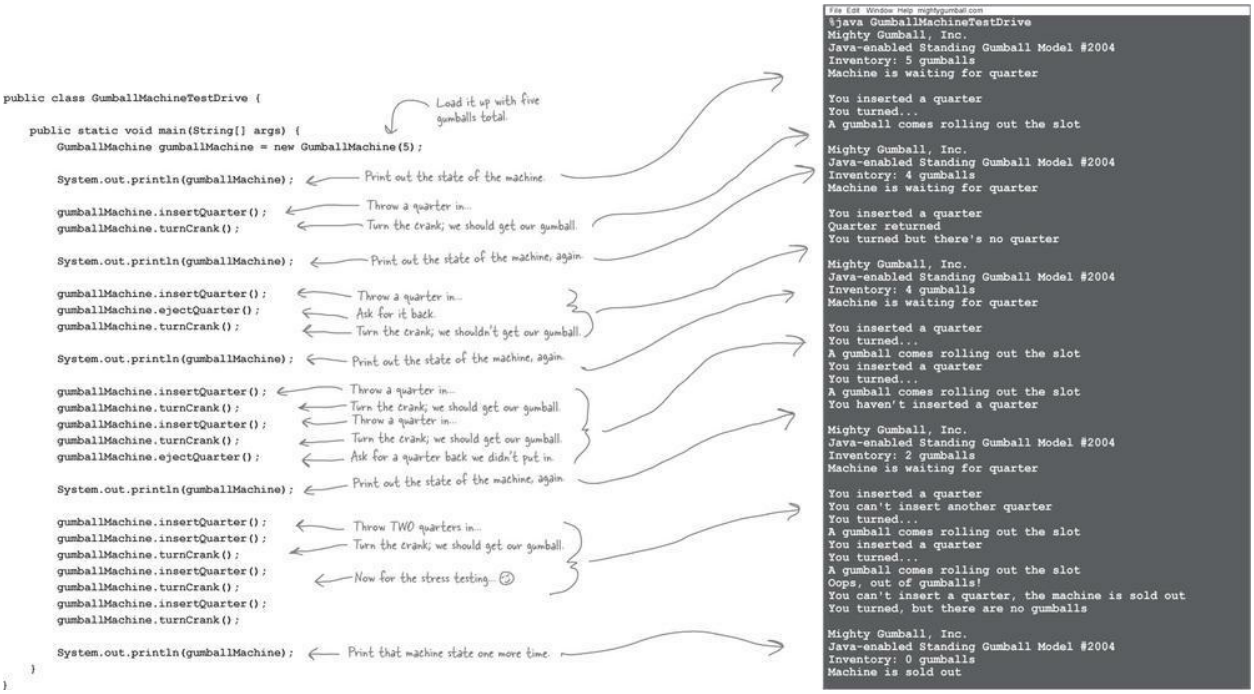
Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

In-house testing

That feels like a nice solid design using a well-thought-out methodology, doesn't it? Let's do a little in-house testing before we hand it off to Mighty Gumball to be loaded into their actual gumball machines. Here's our test

harness:



```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Load it up with five gumballs total.

Print out the state of the machine.

Throw a quarter in...

Turn the crank; we should get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Ask for it back.

Turn the crank; we shouldn't get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Turn the crank; we should get our gumball.

Throw a quarter in...

Turn the crank; we should get our gumball.

Ask for a quarter back we didn't put in.

Print out the state of the machine, again.

Throw TWO quarters in...

Turn the crank; we should get our gumball.

Now for the stress testing... ☺

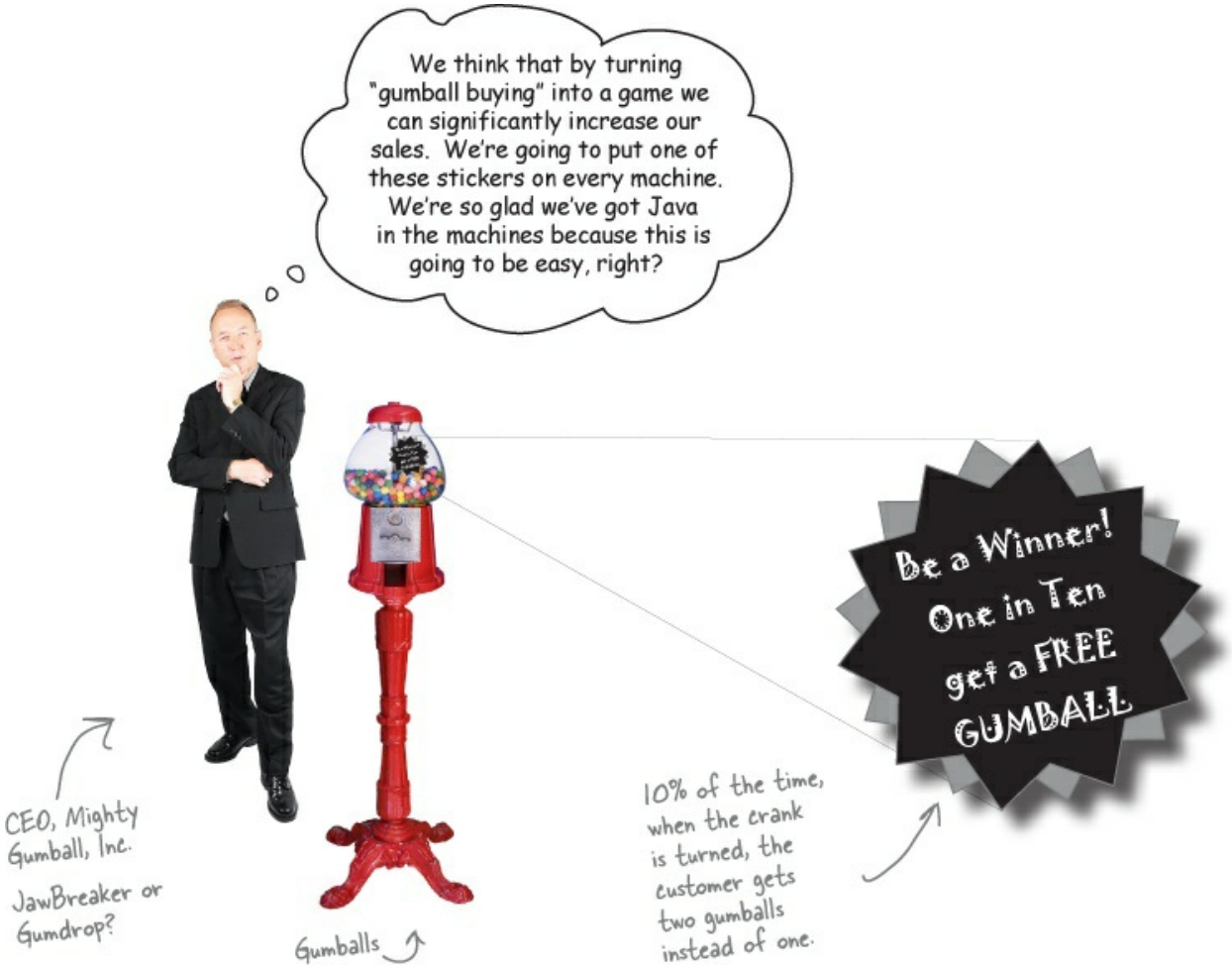
Print that machine state one more time.

```
File Edit Window Help mightygumball.com
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

You knew it was coming... a change request!

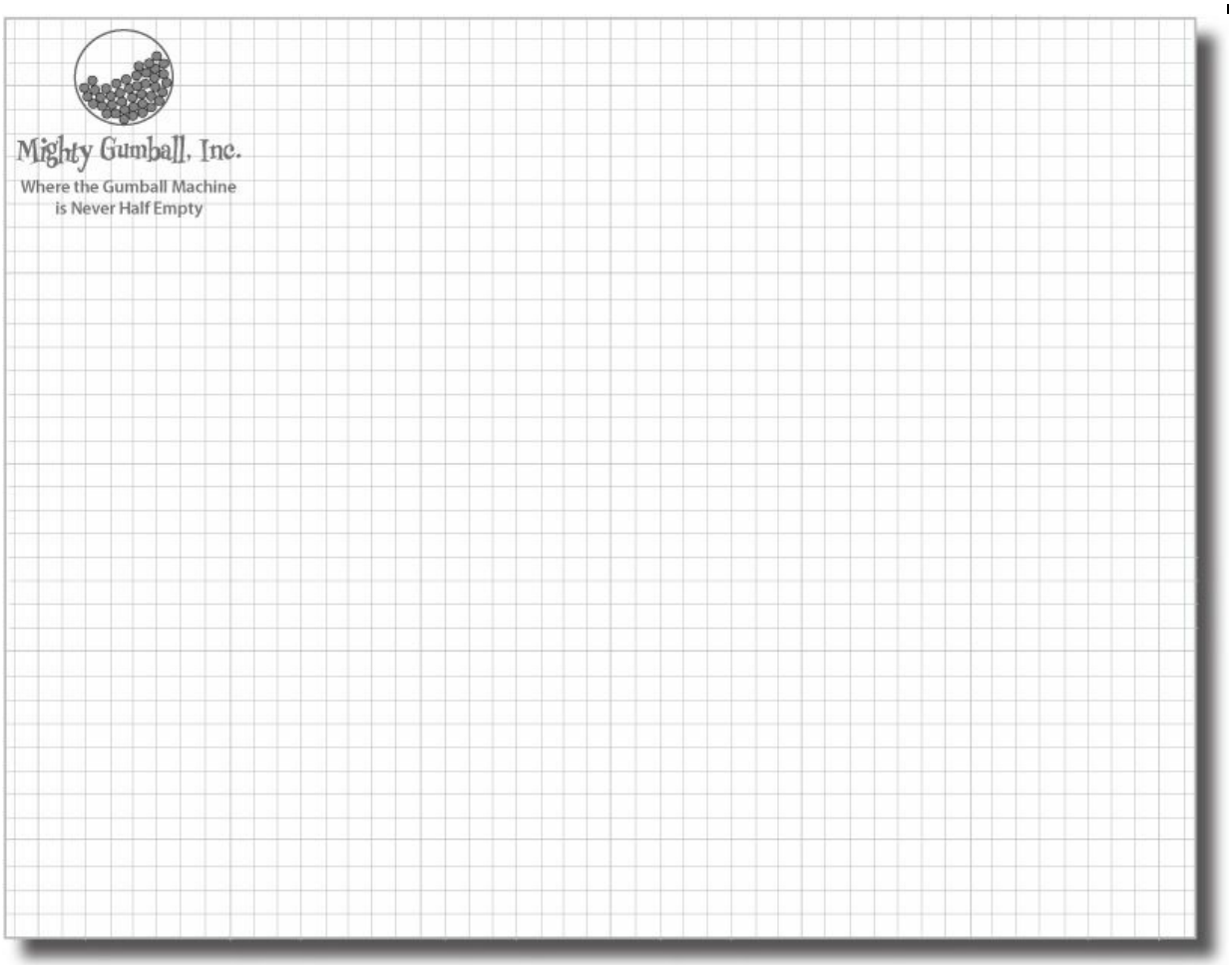
Mighty Gumball, Inc., has loaded your code into their newest machine and their quality assurance experts are putting it through its paces. So far, everything's looking great from their perspective.

In fact, things have gone so smoothly they'd like to take things to the next level...



DESIGN PUZZLE

Draw a state diagram for a Gumball Machine that handles the 1 in 10 contest. In this contest, 10% of the time the Sold state leads to two balls being released, not one. Check your answer with ours (at the end of the chapter) to make sure we agree before you go further...



Use Mighty Gumball's stationery to draw your state diagram.

The messy STATE of things...

Just because you've written your gumball machine using a well-thought-out methodology doesn't mean it's going to be easy to extend. In fact, when you go back and look at your code and think about what you'll have to do to modify it, well...

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

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

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

SHARPEN YOUR PENCIL

Which of the following describe the state of our implementation? (Choose all that apply.)

❑	A.	This code certainly isn't adhering to the Open Closed Principle.
❑	B.	This code would make a FORTRAN programmer proud.
❑	C.	This design isn't even very object-oriented.
❑	D.	State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
❑	E.	We haven't encapsulated anything that varies here.
❑	F.	Further additions are likely to cause bugs in working code.

Frank: You're right about that! We need to refactor this code so that it's easy to maintain and modify.

Judy: We really should try to localize the behavior for each state so that if we make changes to one state, we don't run the risk of messing up the other code.

Frank: Right; in other words, follow that ol' "encapsulate what varies" principle.

Judy: Exactly.

Frank: If we put each state's behavior in its own class, then every state just implements its own actions.

Judy: Right. And maybe the Gumball Machine can just delegate to the state object that represents the current state.

Frank: Ah, you're good: favor composition... more principles at work.

Judy: Cute. Well, I'm not 100% sure how this is going to work, but I think

we’re on to something.

Frank: I wonder if this will make it easier to add new states?

Judy: I think so... We’ll still have to change code, but the changes will be much more limited in scope because adding a new state will mean we just have to add a new class and maybe change a few transitions here and there.

Frank: I like the sound of that. Let’s start hashing out this new design!

The new design

It looks like we’ve got a new plan: instead of maintaining our existing code, we’re going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

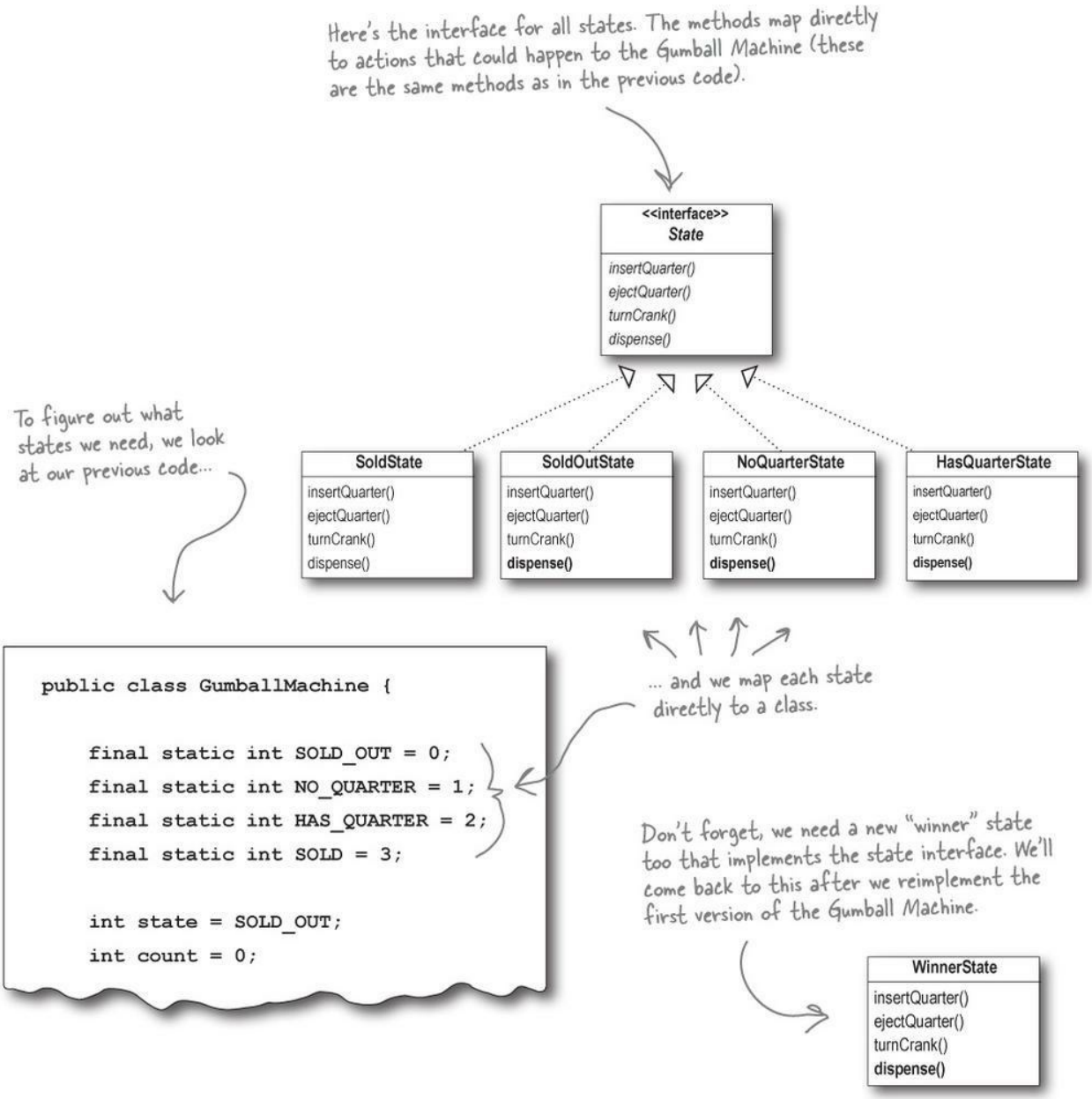
We’re following our design principles here, so we should end up with a design that is easier to maintain down the road. Here’s how we’re going to do it:

> **① First, we’re going to define a State interface that contains a method for every action in the Gumball Machine.**
> **② Then we’re going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
> **③ Finally, we’re going to get rid of all of our conditional code and instead delegate to the State class to do the work for us.**

Not only are we following design principles, as you’ll see, we’re actually implementing the State Pattern. But we’ll get to all the official State Pattern stuff after we rework our code...

Defining the State interfaces and classes

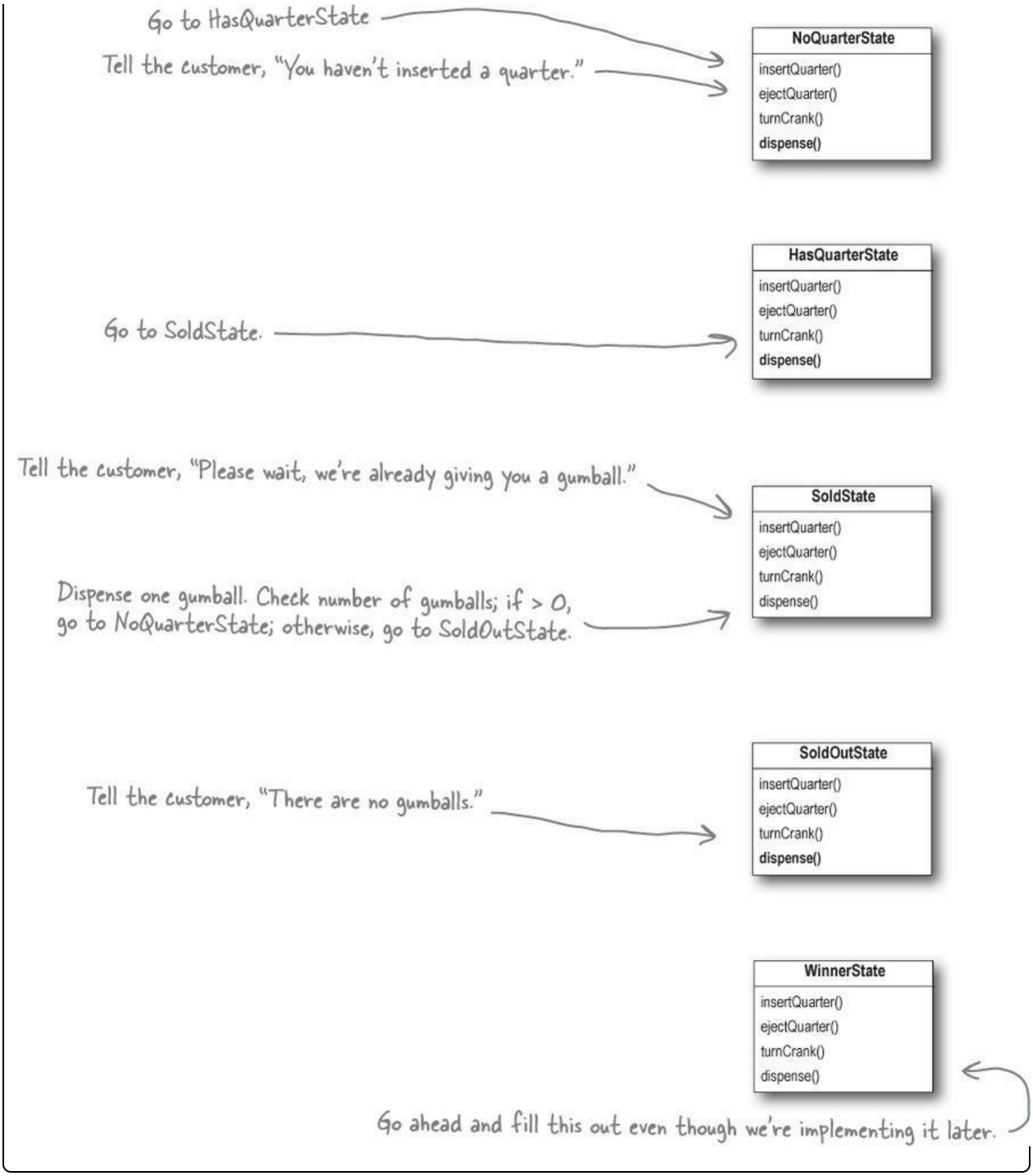
First let's create an interface for State, which all our states implement:



Then take each state in our design and encapsulate it in a class that implements the State interface.

SHARPEN YOUR PENCIL

To implement our states, we first need to specify the behavior of the classes when each action is called. Annotate the diagram below with the behavior of each action in each class; we've already filled in a few for you.



Implementing our State classes

Time to implement a state: we know what behaviors we want; we just need to get it down in code. We're going to closely follow the state machine code we wrote, but this time everything is broken out into different classes.

Let's start with the NoQuarterState:

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

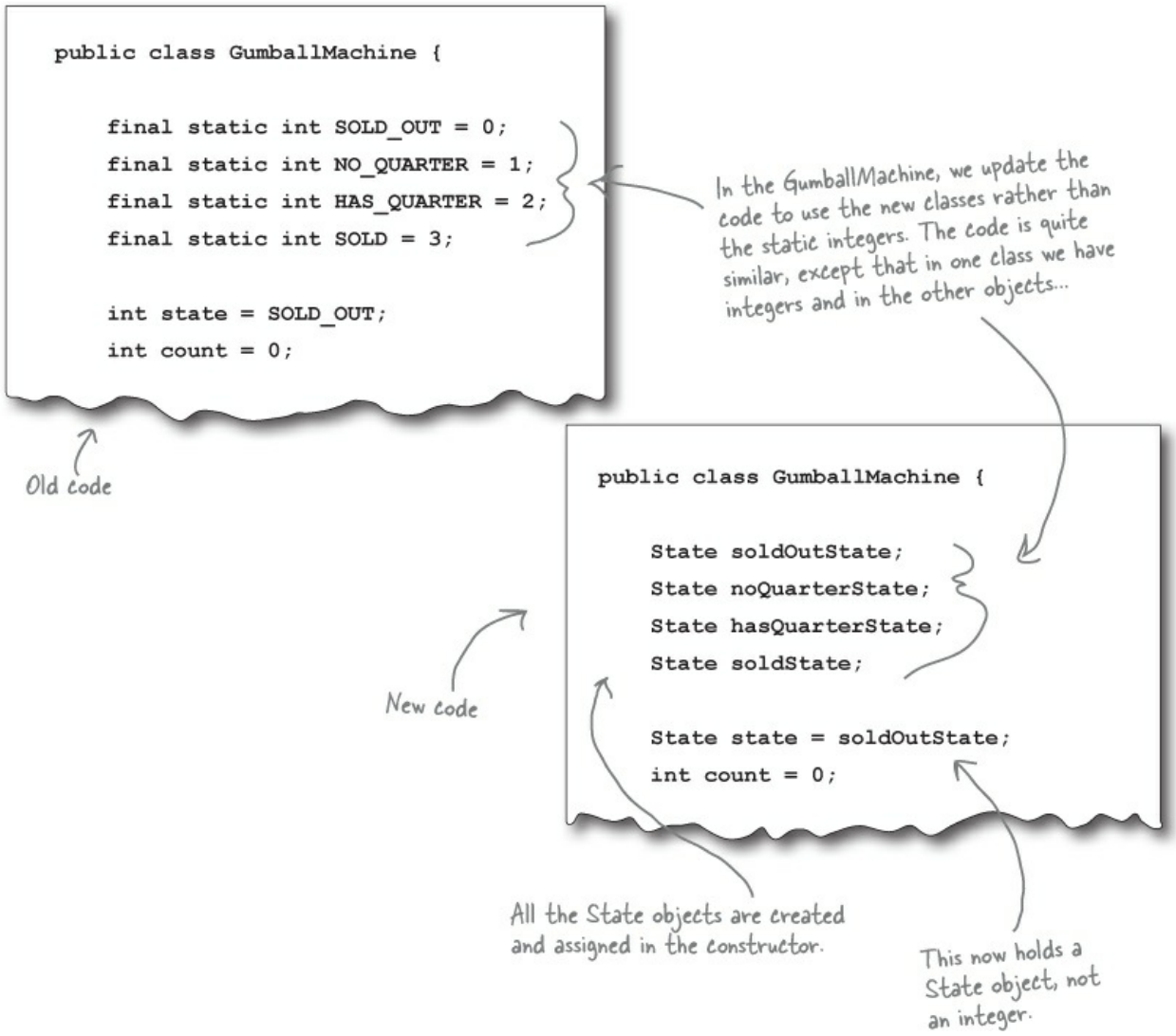
And you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

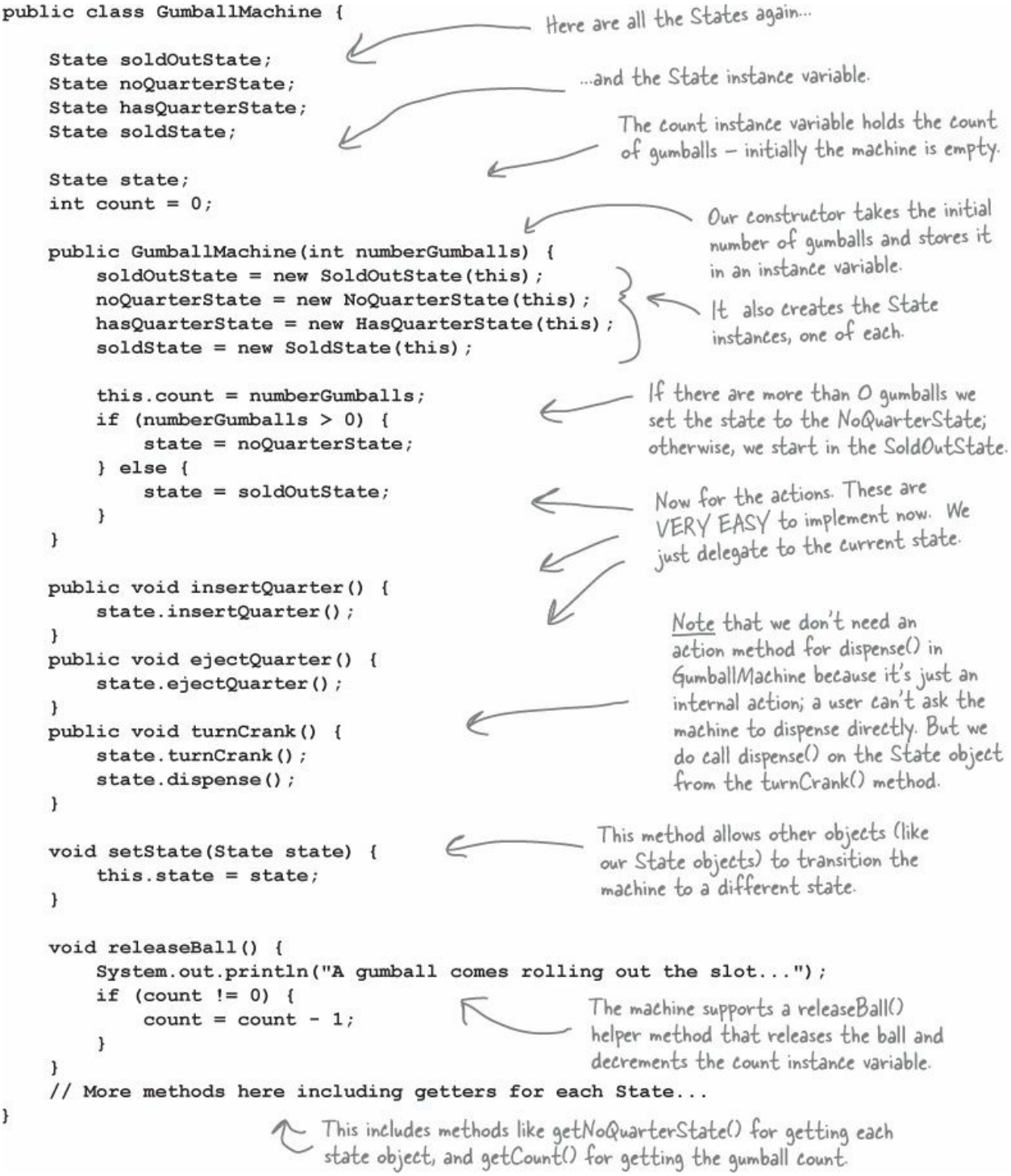
Reworking the Gumball Machine

Before we finish the State classes, we're going to rework the Gumball Machine — that way you can see how it all fits together. We'll start with the state-related instance variables and switch the code from using integers to

using state objects:

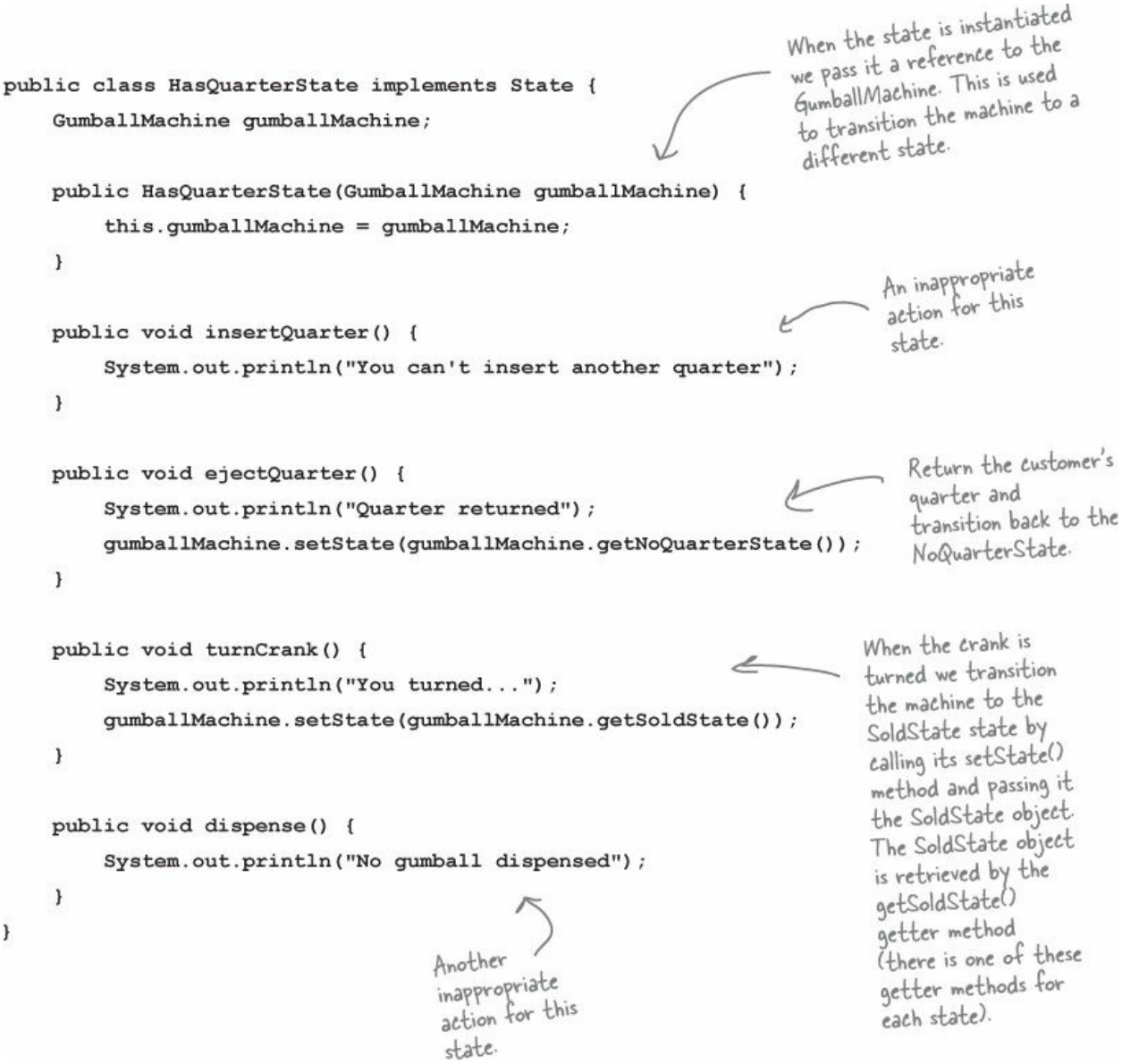


Now, let's look at the complete GumballMachine class...



Implementing more states

Now that you're starting to get a feel for how the Gumball Machine and the states fit together, let's implement the HasQuarterState and the SoldState classes...




```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

Now, let's check out the SoldState class...

```
public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Here are all the inappropriate actions for this state.

And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

BRAIN POWER

Look back at the GumballMachine implementation. If the crank is turned and not successful (say the customer didn't insert a quarter first), we call dispense anyway, even though it's unnecessary. How might you fix this?

SHARPEN YOUR PENCIL

We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Check your answer before moving on...

```
public class SoldOutState implements _____________ {
    GumballMachine gumballMachine;

     public SoldOutState(GumballMachine gumballMachine) {
```

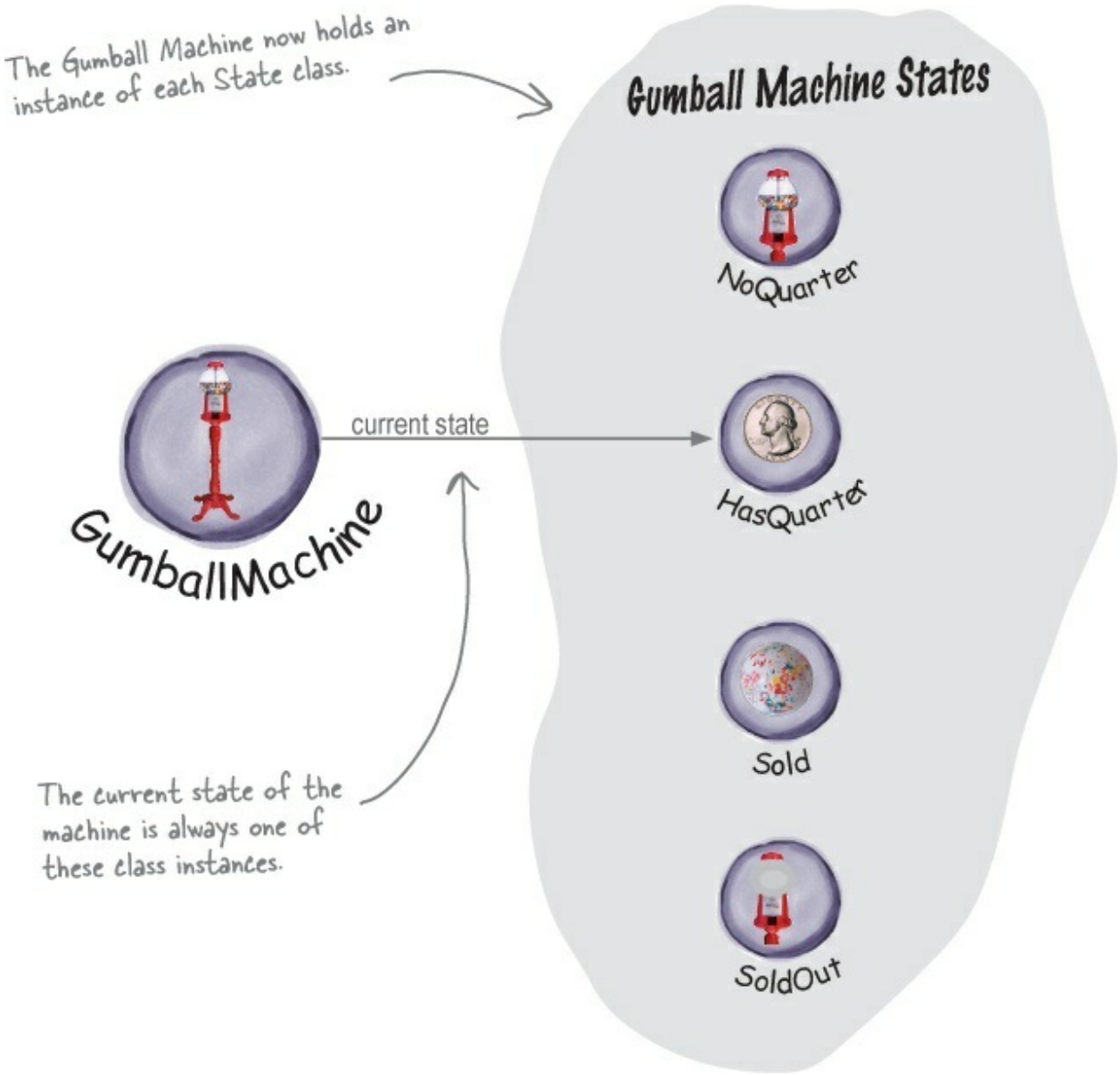
```
        }

        public void insertQuarter() {


        }

        public void ejectQuarter() {

        }

        public void turnCrank() {



        }

        public void dispense() {



        }
}
```

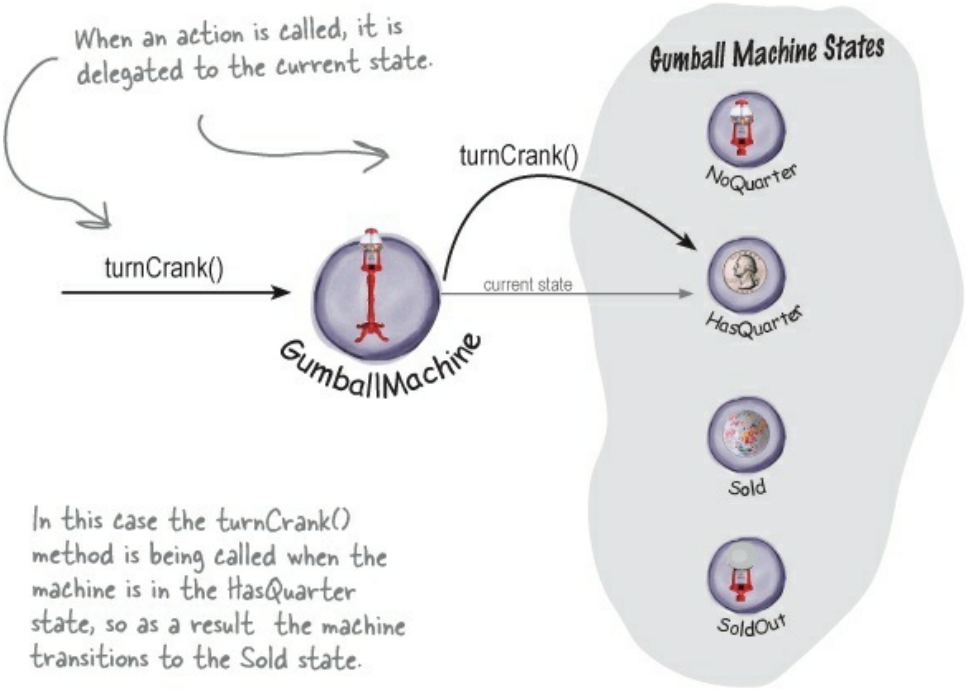
Let’s take a look at what we’ve done so far...

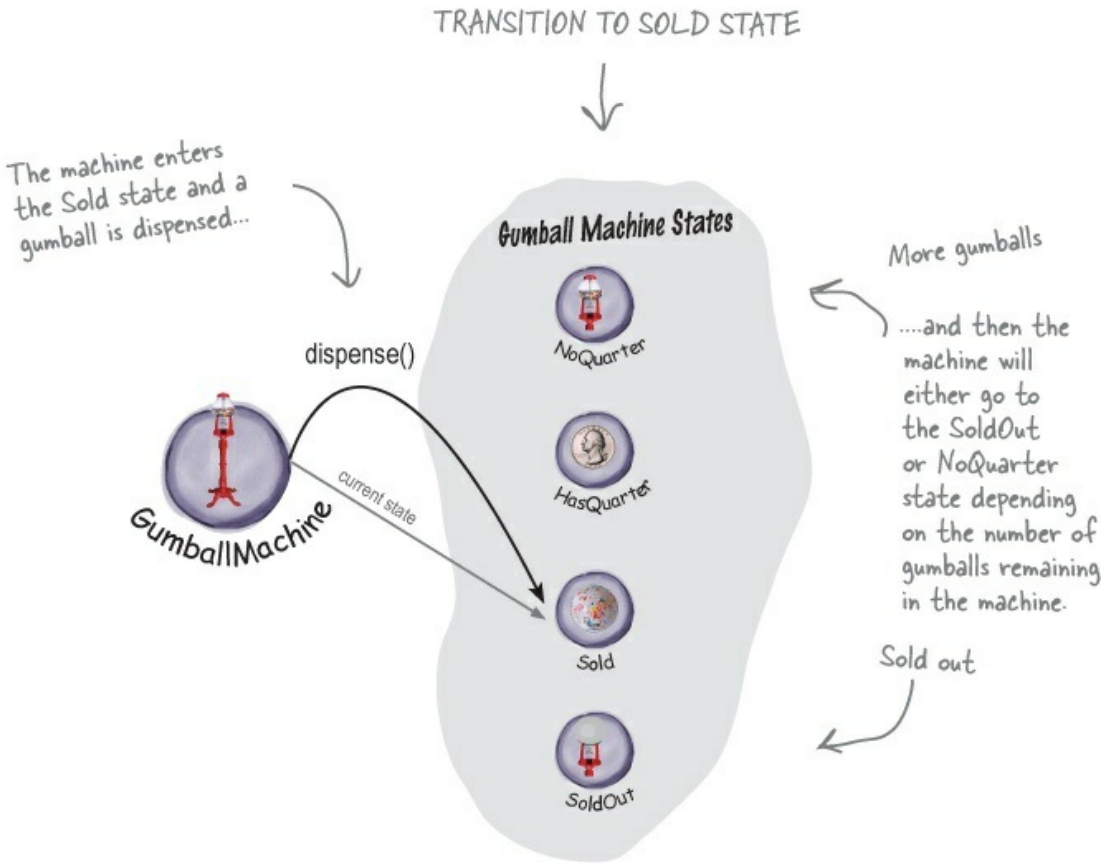
For starters, you now have a Gumball Machine implementation that is *structurally* quite different from your first version, and yet *functionally it is exactly the same*. By structurally changing the implemention, you’ve:

- Localized the behavior of each state into its own class.
- Removed all the troublesome `if` statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes (and we’ll do this in a second).
- Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.

Now let’s look a little more at the functional aspect of what we did:


The Gumball Machine now holds an instance of each State class.
Gumball Machine States
NoQuarter
current state
HasQuarter
GumballMachine
Sold
The current state of the machine is always one of these class instances.
SoldOut

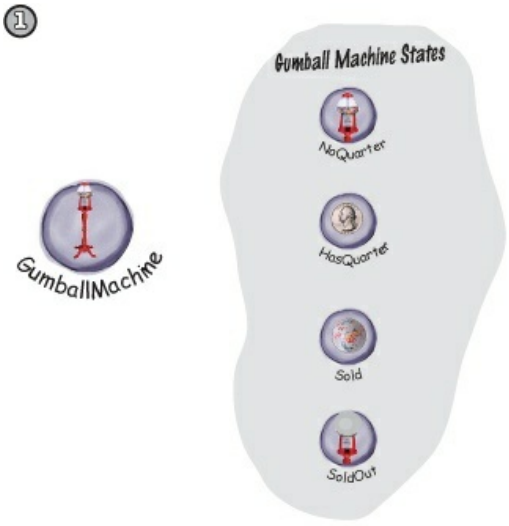


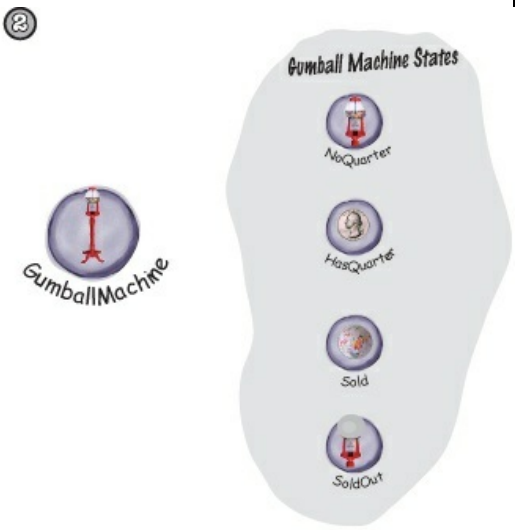


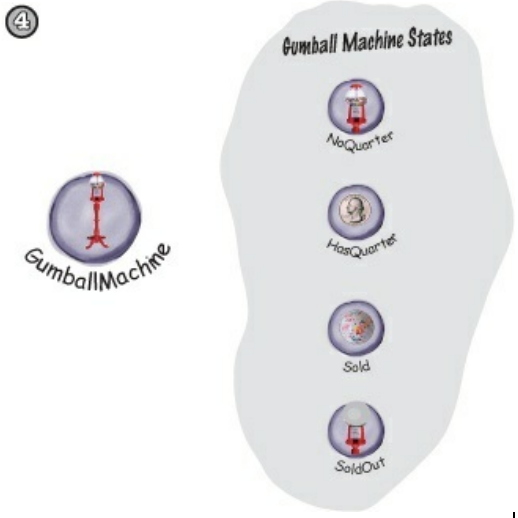
SHARPEN YOUR PENCIL

Behind the Scenes: Self-Guided Tour

Trace the steps of the Gumball Machine starting with the NoQuarter state. Also annotate the diagram with actions and output of the machine. For this exercise you can assume there are plenty of gumballs in the machine.







The State Pattern defined

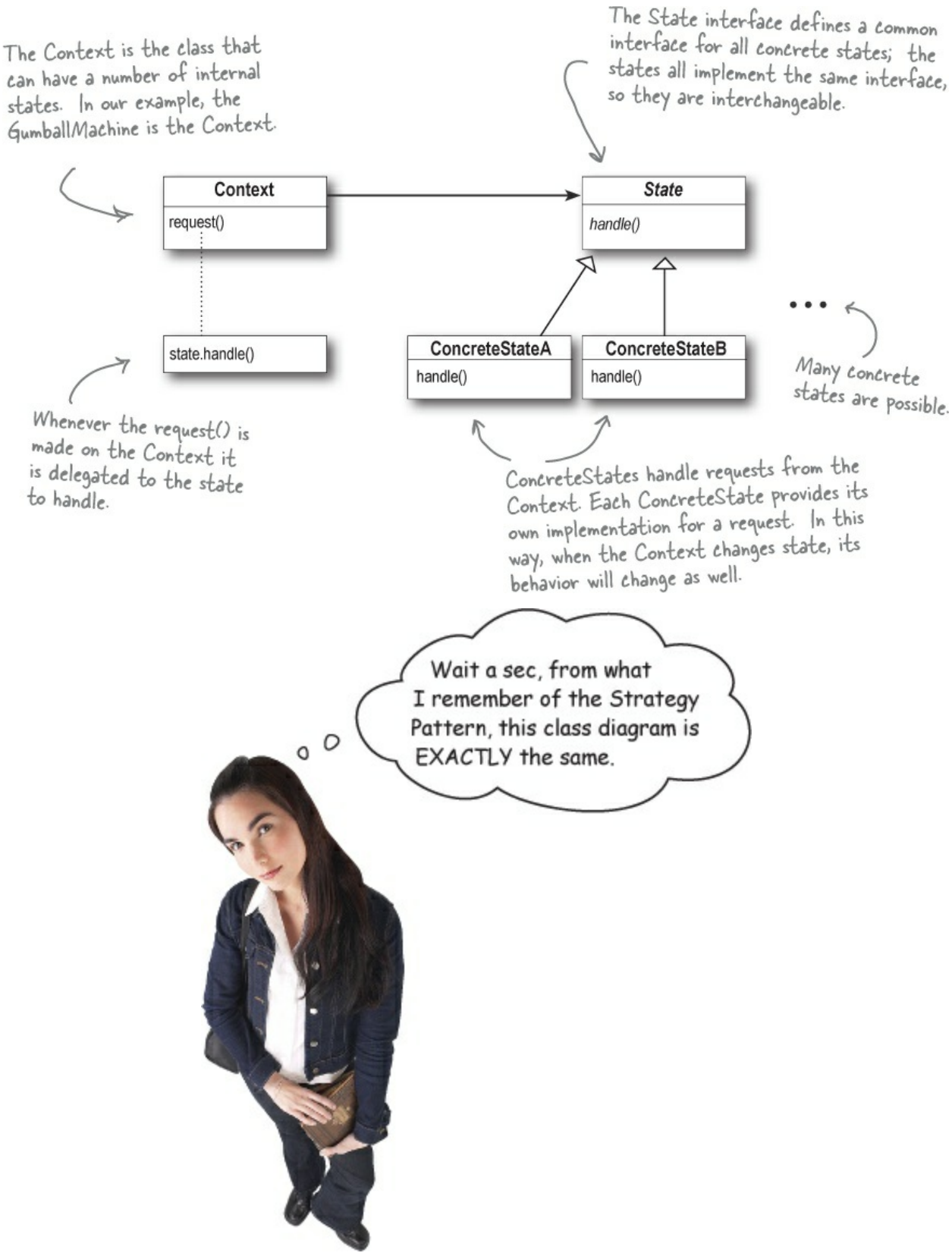
Yes, it's true, we just implemented the State Pattern! So now, let's take a look at what it's all about:

> **NOTE**
>
> **The State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example: when the gumball machine is in the NoQuarterState and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the HasQuarterState (the machine rejects the quarter).

What about the second part of the definition? What does it mean for an object to "appear to change its class"? Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Okay, now it's time to check out the State Pattern class diagram:



You've got a good eye! Yes, the class diagrams are essentially the same, but

the two patterns differ in their *intent*.

With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

THERE ARE NO DUMB QUESTIONS

Q: **Q: In the GumballMachine, the states decide what the next state should be. Do the ConcreteStates always decide what state to go to next?**

A: **A:** No, not always. The alternative is to let the Context decide on the flow of state transitions.

As a general guideline, when the state transitions are fixed they are appropriate for putting in the Context; however, when the transitions are more dynamic, they are typically placed in the state classes themselves (for instance, in the GumballMachine the choice of the transition to NoQuarter or SoldOut depended on the runtime count of gumballs).

The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes. In our implementation of the GumballMachine we tried to minimize this by using getter methods on the Context, rather than hardcoding explicit concrete state classes.

Notice that by making this decision, you are making a decision as to which classes are closed for modification — the Context or the state classes — as the system evolves.

Q: **Q: Do clients ever interact directly with the states?**

A: **A:** No. The states are used by the Context to represent its internal state and behavior, so all requests to the states come from the Context. Clients don't directly change the state of the Context. It is the Context's job to oversee its state, and you don't usually want a client changing the state of a Context without that Context's knowledge.

Q: **Q: If I have lots of instances of the Context in my application, is it possible to share the state objects across them?**

A: **A:** Yes, absolutely, and in fact this is a very common scenario. The only requirement is that your state objects do not keep their own internal context; otherwise, you'd need a unique instance per context.

To share your states, you'll typically assign each state to a static instance variable. If your state needs to make use of methods or instance variables in your Context, you'll also have to give it a reference to the Context in each handler() method.

Q: **Q: It seems like using the State Pattern always increases the number of classes in our designs. Look how many more classes our GumballMachine had than the original design!**

A: **A:** You're right, by encapsulating state behavior into separate state classes, you'll always end up with more classes in your design. That's often the price you pay for flexibility. Unless your code is some "one off" implementation you're going to throw away (yeah, right), consider building it with the additional classes and you'll probably thank yourself down the road. Note that often what is important is the number of classes that you expose to your clients, and there are ways to hide these extra classes from your clients (say, by declaring them package visible).

Also, consider the alternative: if you have an application that has a lot of state and you decide not to use separate objects, you'll instead end up with very large, monolithic conditional statements. This makes your code hard to maintain and understand. By using objects, you make states explicit and reduce the effort needed to understand and maintain your code.

Q: **Q: The State Pattern class diagram shows that State is an abstract class. But didn't you use an interface in the implementation of the gumball machine's state?**

A: **A:** Yes. Given we had no common functionality to put into an abstract class, we went with an interface. In your own implementation, you might want to consider an abstract class. Doing so has the benefit of allowing you to add methods to the abstract class later, without breaking the concrete state implementations.

We still need to finish the Gumball 1 in 10 game

Remember, we're not done yet. We've got a game to implement, but now that we've got the State Pattern implemented, it should be a breeze. First, we need to add a state to the GumballMachine class:

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;
    // methods here
}
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

Now let’s implement the WinnerState class; it’s remarkably similar to the SoldState class:

```
public class WinnerState implements State {

    // instance variables and constructor
    // insertQuarter error message
    // ejectQuarter error message
    // turnCrank error message

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            System.out.println("YOU'RE A WINNER! You got two gumballs for your quarter");
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

If we have a second gumball we release it.

If we were able to release two gumballs, we let the user know he was a winner.

Finishing the game

We’ve just got one more change to make: we need to implement the random chance game and add a transition to the WinnerState. We’re going to add both to the HasQuarterState since that is where the customer turns the crank:


```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

Wow, that was pretty simple to implement! We just added a new state to the GumballMachine and then implemented it. All we had to do from there was to implement our chance game and transition to the correct state. It looks like our new code strategy is paying off...

Demo for the CEO of Mighty Gumball, Inc.

The CEO of Mighty Gumball has dropped by for a demo of your new gumball game code. Let's hope those states are all in order! We'll keep the demo short and sweet (the short attention span of CEOs is well documented), but hopefully long enough so that we'll win at least once.

This code really hasn't changed at all; we just shortened it a bit.

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Once, again, start with a gumball machine with 5 gumballs.

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

The whole engineering team is waiting outside the conference room to see if the new State Pattern–based design is going to work!!


```
File Edit Window Help Whenisagumballajawbreaker?
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You got two gumballs for your quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You got two gumballs for your quarter
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why do we need the WinnerState? Couldn't we just have the SoldState dispense two gumballs?**

A: **A:** That's a great question. SoldState and WinnerState are almost identical, except that WinnerState dispenses two gumballs instead of one. You certainly could put the code to dispense two gumballs into the SoldState. The downside is, of course, that now you've got TWO states represented in one State class: the state in which you're a winner, and the state in which you're not. So you are sacrificing clarity in your State class to reduce code duplication. Another thing to consider is the principle you learned in the previous chapter: One class, One responsibility. By putting the WinnerState responsibility into the SoldState, you've just given the SoldState TWO responsibilities. What happens when the promotion ends? Or the stakes of the contest change? So, it's a tradeoff and comes down to a design decision.

Sanity check...

Yes, the CEO of Mighty Gumball probably needs a sanity check, but that's not what we're talking about here. Let's think through some aspects of the GumballMachine that we might want to shore up before we ship the gold version:

- We've got a lot of duplicate code in the Sold and Winning states and we might want to clean those up. How would we do it? We could make State into an abstract class and build in some default behavior for the methods; after all, error messages like, "You already inserted a quarter," aren't going to be seen by the customer. So all "error response" behavior could be generic and inherited from the abstract State class.

  NOTE

  Dammit Jim, I'm a gumball machine, not a computer!

- The dispense() method always gets called, even if the crank is turned when there is no quarter. While the machine operates correctly and doesn't dispense unless it's in the right state, we could easily fix this by having turnCrank() return a boolean, or by introducing exceptions. Which

do you think is a better solution?

- All of the intelligence for the state transitions is in the State classes. What problems might this cause? Would we want to move that logic into the Gumball Machine? What would be the advantages and disadvantages of that?
- Will you be instantiating a lot of GumballMachine objects? If so, you may want to move the state instances into static instance variables and share them. What changes would this require to the GumballMachine and the States?

FIRESIDE CHATS

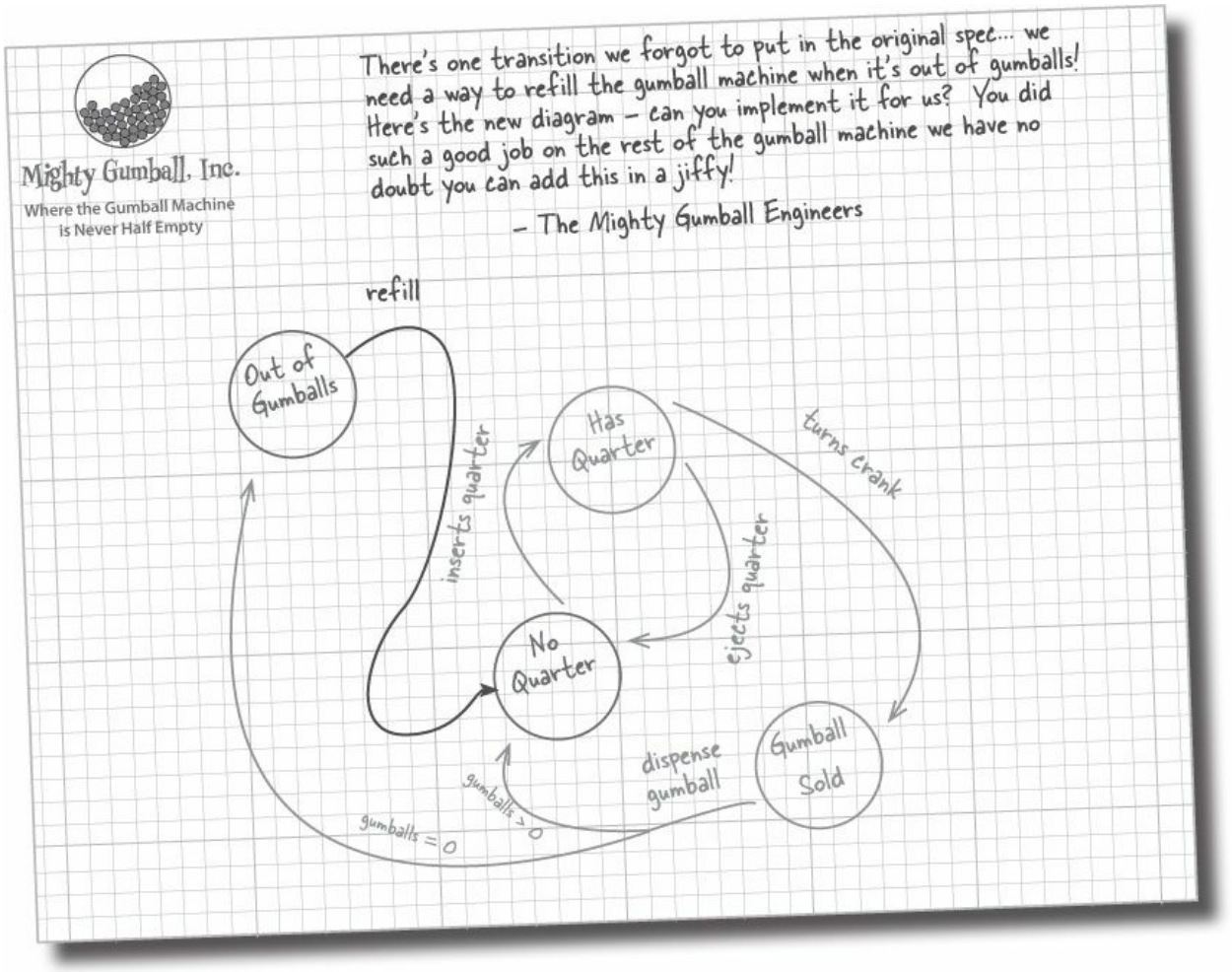
Tonight's talk: **A Strategy and State Pattern Reunion.**

Strategy:	State:
Hey bro. Did you hear I was in Chapter 1?	
	Yeah, word is definitely getting around.
I was just over giving the Template Method guys a hand — they needed me to help them finish off their chapter. So, anyway, what is my noble brother up to?	
	Same as always — helping classes to exhibit different behaviors in different states.
I don't know, you always sound like you've just copied what I do and you're using different words to describe it. Think about it: I allow objects to incorporate different behaviors or algorithms through composition and delegation. You're just copying me.	
	I admit that what we do is definitely related, but my intent is totally different than yours. And, the way I teach my clients to use composition and delegation is totally different.
Oh yeah? How so? I don't get it.	
	Well, if you spent a little more time thinking about

	something other than *yourself*, you might. Anyway, think about how you work: you have a class you're instantiating and you usually give it a strategy object that implements some behavior. Like, in Chapter 1 you were handing out quack behaviors, right? Real ducks got a real quack; rubber ducks got a quack that squeaked.
Yeah, that was some *fine* work... and I'm sure you can see how that's more powerful than inheriting your behavior, right?	
	Yes, of course. Now, think about how I work; it's totally different.
Sorry, you're going to have to explain that.	
	Okay, when my Context objects get created, I may tell them the state to start in, but then they change their own state over time.
Hey, come on, I can change behavior at runtime too; that's what composition is all about!	
	Sure you can, but the way I work is built around discrete states; my Context objects change state over time according to some well-defined state transitions. In other words, changing behavior is built in to my scheme — it's how I work!
Well, I admit, I don't encourage my objects to have a well-defined set of transitions between states. In fact, I typically like to control what strategy my objects are using.	
	Look, we've already said we're alike in structure, but what we do is quite different in intent. Face it, the world has uses for both of us.
Yeah, yeah, keep living your pipe dreams, brother. You act like you're a big pattern like me, but check it out: I'm in Chapter 1; they stuck you way out in Chapter 10. I mean, how many people are actually going to read this far?	

	Are you kidding? This is a Head First book and Head First readers rock. Of course they're going to get to Chapter 10!
That's my brother, always the dreamer.	

We almost forgot!



SHARPEN YOUR PENCIL

We need you to write the refill() method for the Gumball machine. It has one argument — the number of gumballs you're adding to the machine — and should update the gumball machine count and reset the machine's state.

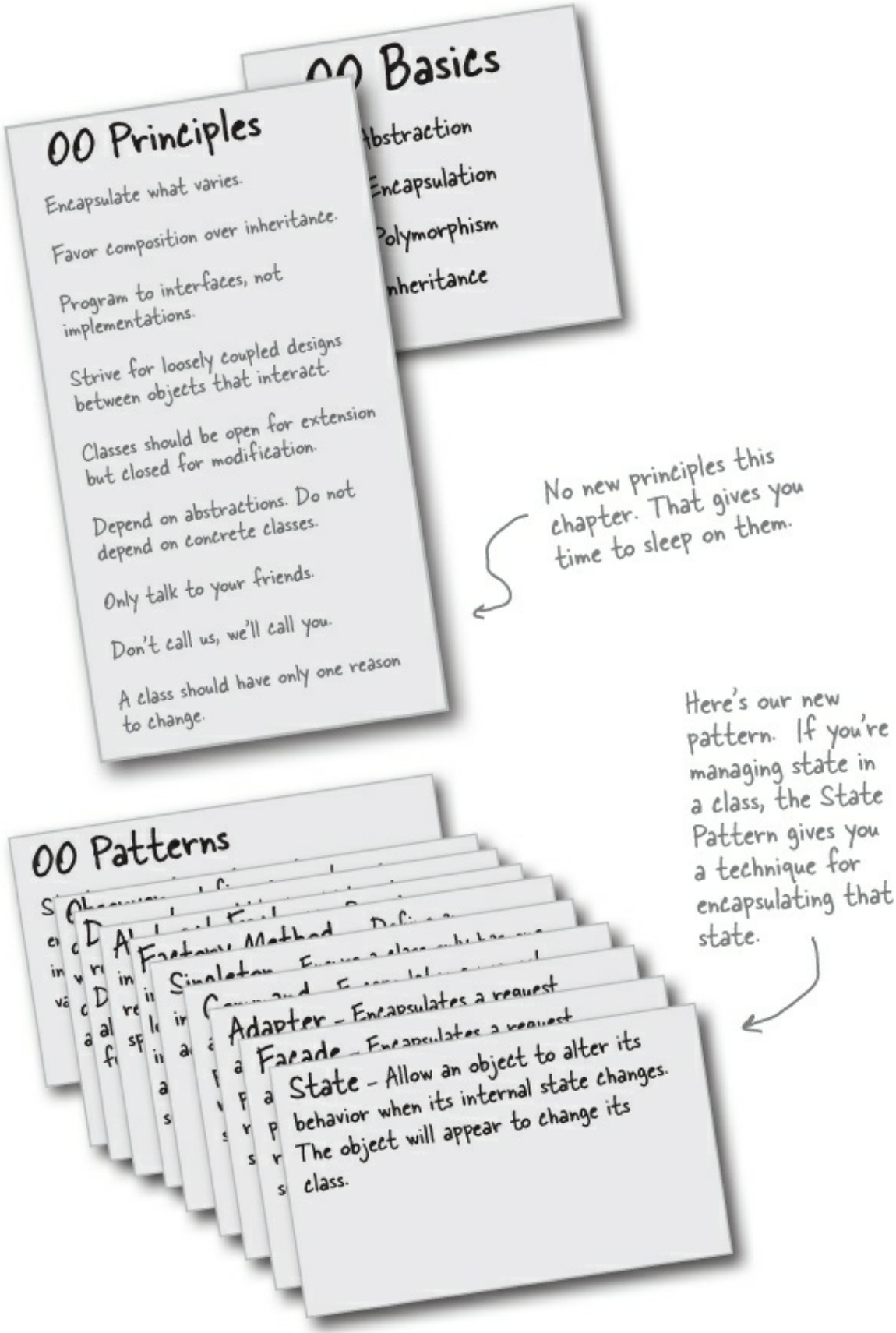
WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
State	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Template Method	Encapsulate state-based behavior and delegate behavior to the current state.

Tools for your Design Toolbox

It's the end of another chapter; you've got enough patterns here to breeze through any job interview!

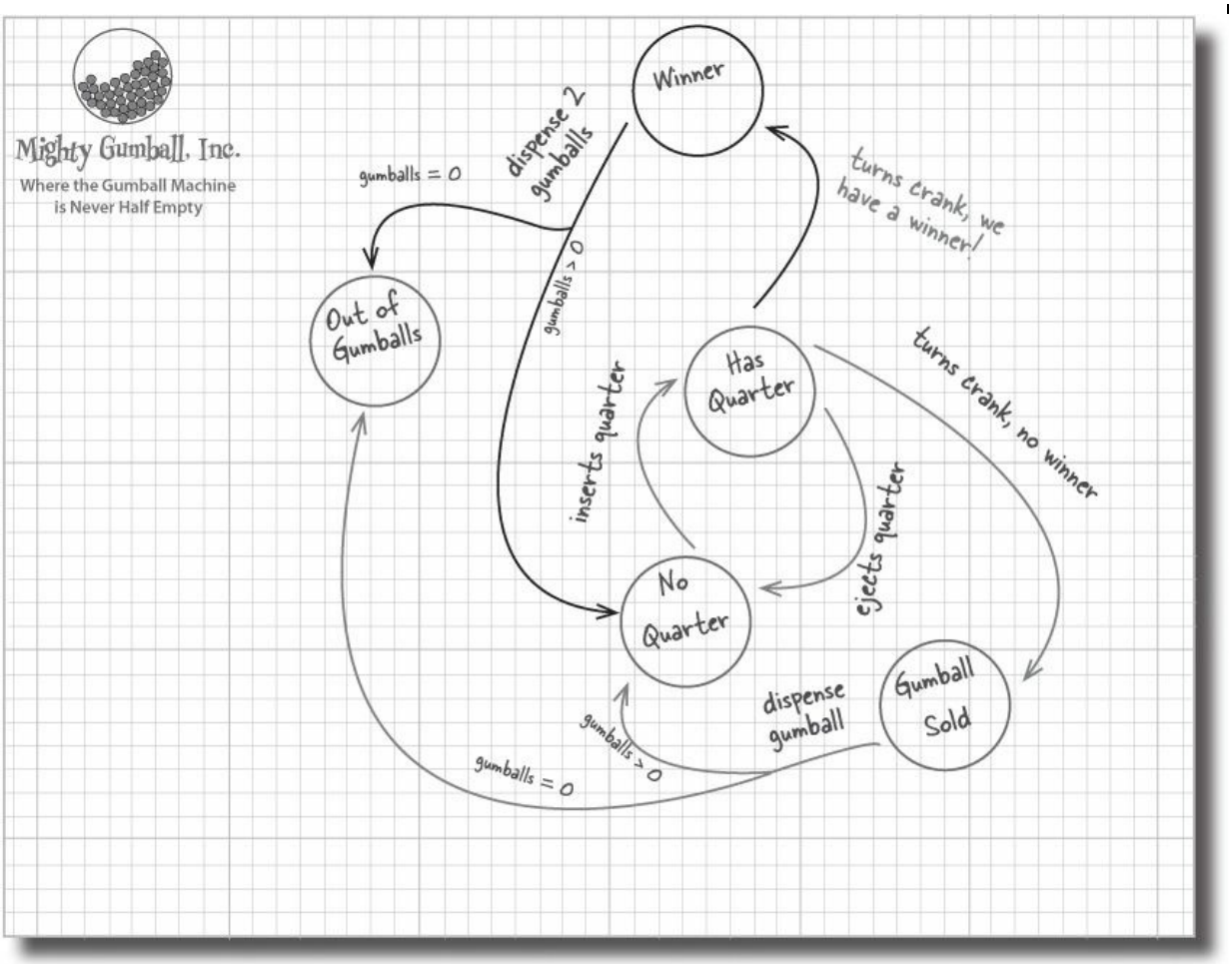

OO Basics
bstraction
Encapsulation
olymorphism
nheritance
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
No new principles this chapter. That gives you time to sleep on them.
OO Patterns
Adapter – Encapsulates a request
State – Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Here's our new pattern. If you're managing state in a class, the State Pattern gives you a technique for encapsulating that state.

BULLET POINTS

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class.
- The Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

DESIGN PUZZLE SOLUTION

Draw a state diagram for a Gumball Machine that handles the 1-in-10 contest. In this contest, 10% of the time the Sold state leads to two balls being released, not one. Here's our solution.



SHARPEN YOUR PENCIL SOLUTION

Which of the following describe the state of our implementation? (Choose all that apply.) Here's our solution.

☑	A.	This code certainly isn't adhering to the Open Closed Principle.
☑	B.	This code would make a FORTRAN programmer proud.
☑	C.	This design isn't even very object-oriented.
☑	D.	State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
☑	E.	We haven't encapsulated anything that varies here.
☑	F.	Further additions are likely to cause bugs in working code.

SHARPEN YOUR PENCIL SOLUTION

We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Here's our solution.

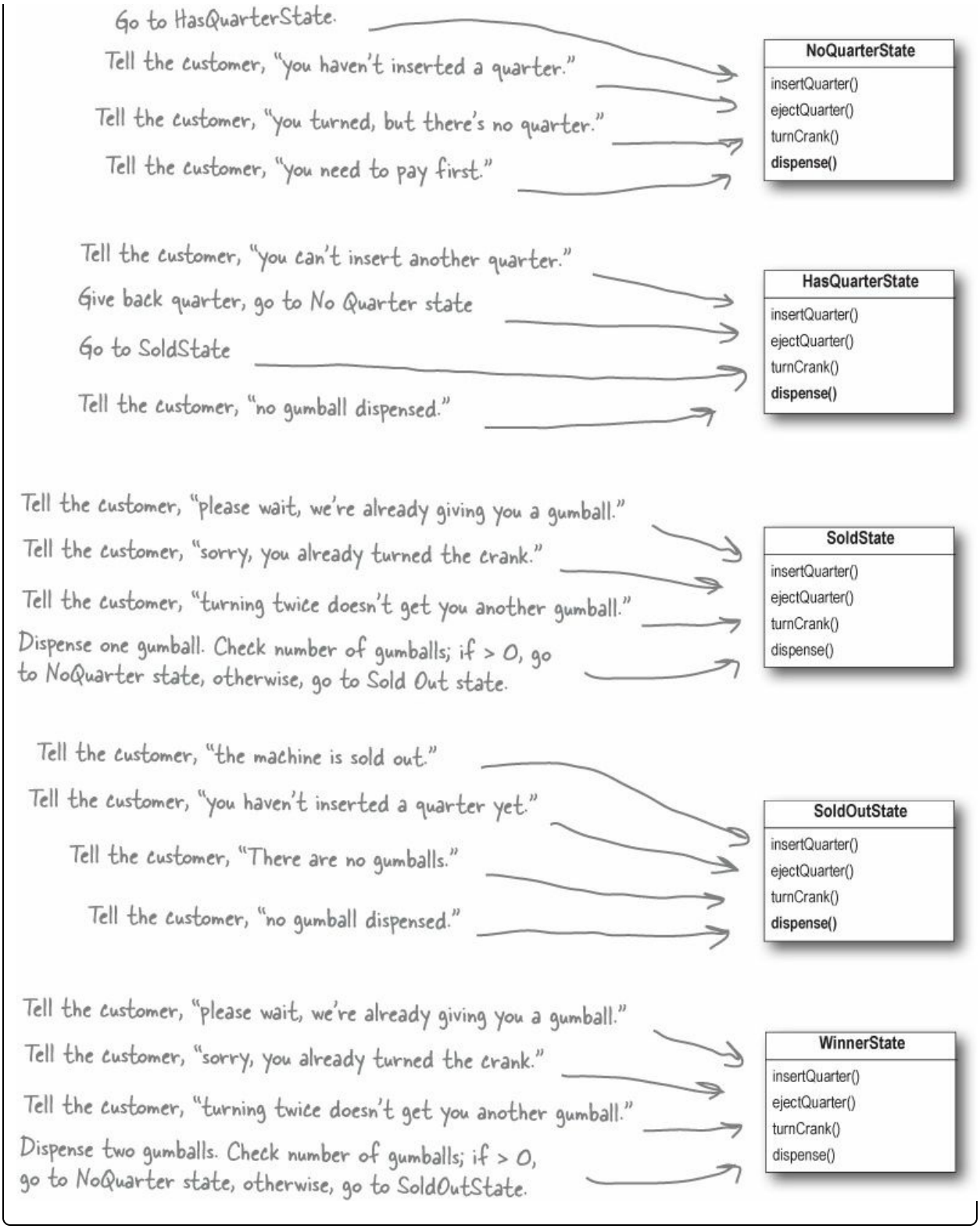
```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

     public SoldOutState(GumballMachine gumballMachine) {
         this.gumballMachine = gumballMachine;
     }

     public void insertQuarter() {
         System.out.println("You can't insert a quarter, the machine is sold
out");
     }

     public void ejectQuarter() {
         System.out.println("You can't eject, you haven't inserted a quarter
yet");
     }

     public void turnCrank() {
         System.out.println("You turned, but there are no gumballs");
     }

     public void dispense() {
         System.out.println("No gumball dispensed");
     }

     public String toString() {
         return "sold out";
     }
}
```

NOTE

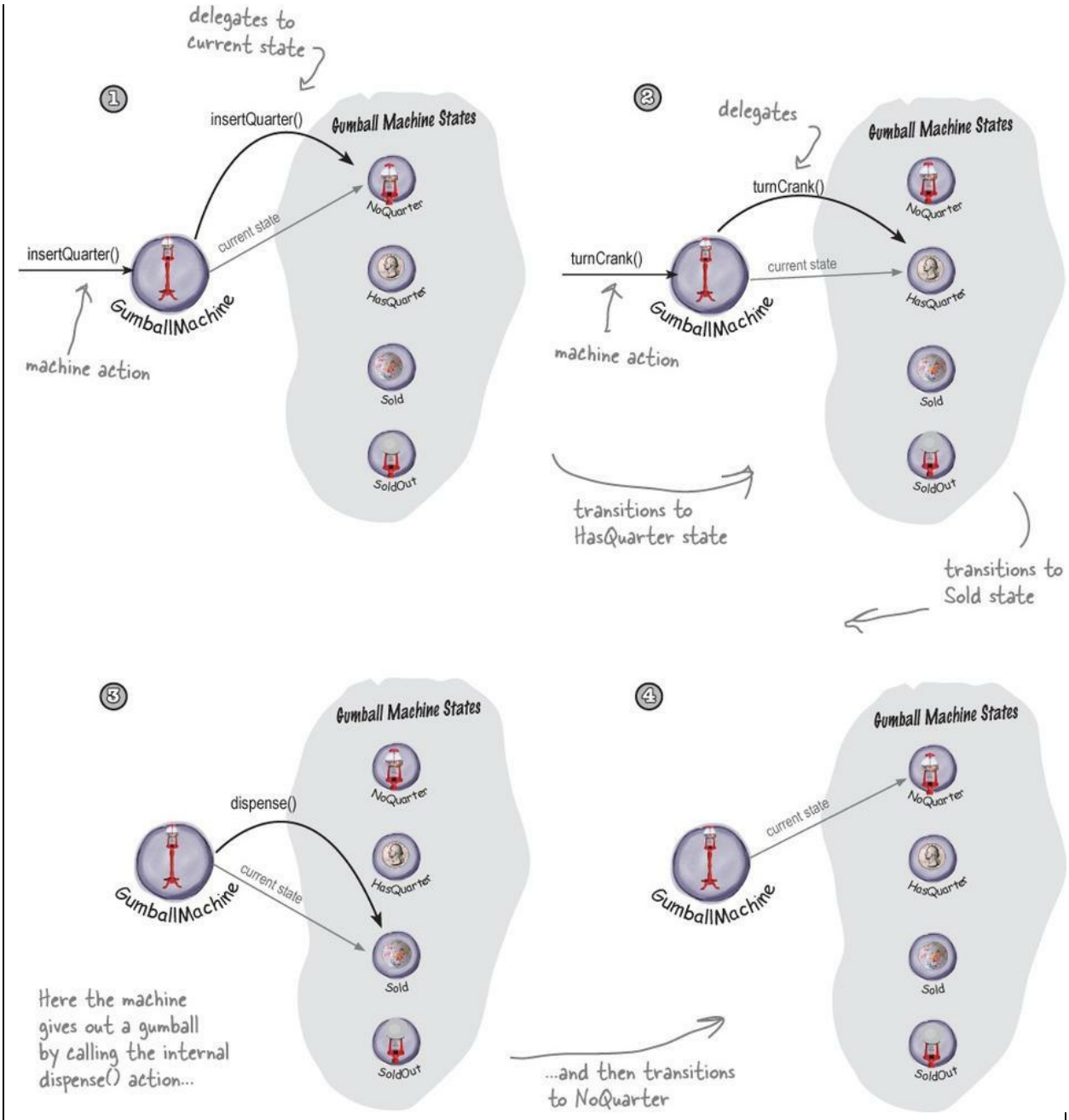
In the Sold Out state, we really can't do anything until someone refills the Gumball Machine.

SHARPEN YOUR PENCIL SOLUTION

To implement the states, we first need to define what the behavior will be when the corresponding action is called. Annotate the diagram below with the behavior of each action in each class; here's our solution.

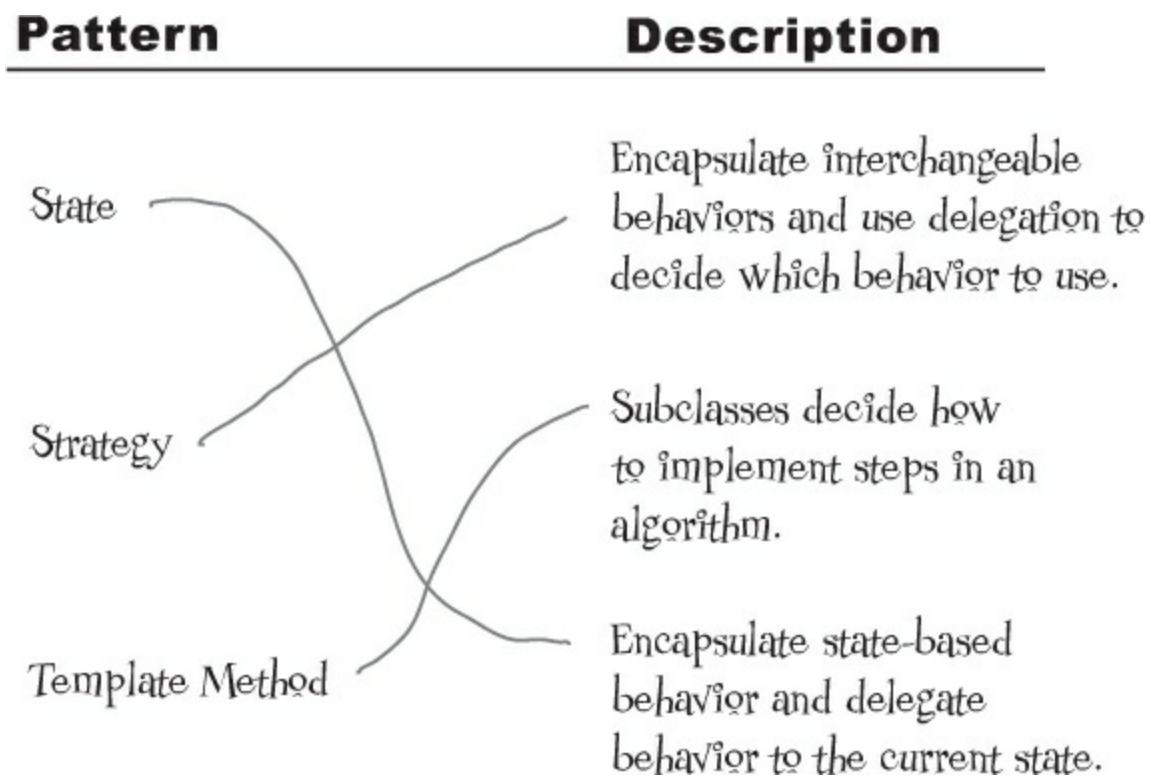


BEHIND THE SCENES: SELF-GUIDED TOUR SOLUTION



WHO DOES WHAT? SOLUTION

Match each pattern with its description:



SHARPEN YOUR PENCIL SOLUTION

To refill the Gumball Machine, we add a refill() method to the State interface, which each State must implement. In every state except the SoldOutState, the method does nothing. In SoldOutState, refill() transitions to NoQuarterState. We also add a refill() method to GumballMachine that adds to the count of gumballs, and then calls the current state's refill() method.

```
public void refill() {
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}

void refill(int count) {
    this.count += count;
    System.out.println("The gumball machine was just refilled; it's new count is: " + this.count);
    state.refill();
}
```

We add this method to the SoldOutState.

And add this method to the GumballMachine.

Chapter 11. The Proxy Pattern: Controlling Object Access

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

Sounds easy enough. If you remember, we've already got methods in the gumball machine code for getting the count of gumballs (getCount()), and getting the current state of the machine (getState()).

All we need to do is create a report that can be printed out and sent back to the CEO. Hmmm, we should probably add a location field to each gumball machine as well; that way the CEO can keep the machines straight.

Let's just jump in and code this. We'll impress the CEO with a very fast turnaround.

Coding the Monitor

Let's start by adding support to the GumballMachine class so that it can handle locations:

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

The location is passed into the constructor and stored in the instance variable.

Let's also add a getter method to grab the location when we need it.

Now let's create another class, GumballMonitor, that retrieves the machine's location, inventory of gumballs, and current machine state and prints them in a nice little report:

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report method just prints a report with location, inventory and the machine's state.

Testing the Monitor

We implemented that in no time. The CEO is going to be thrilled and amazed by our development skills.

Now we just need to instantiate a GumballMonitor and give it a machine to monitor:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);

        // rest of test code here

        monitor.report();
    }
}
```

Pass in a location and initial # of gumballs on the command line.

Don't forget to give the constructor a location and count...

...and instantiate a monitor and pass it a machine to provide a report on.

When we need a report on the machine, we call the report() method.

```
File Edit Window Help FlyingFish
%java GumballMachineTestDrive Seattle 112
Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

And here's the output!

The monitor output looks great, but I guess I wasn't clear. I need to monitor gumball machines REMOTELY! In fact, we already have the networks in place for monitoring. Come on guys, you're supposed to be the Internet generation!

Frank: A remote what?

Joe: Remote proxy. Think about it: we've already got the monitor code written, right? We give the GumballMonitor a reference to a machine and it gives us a report. The problem is that the monitor runs in the same JVM as the gumball machine and the CEO wants to sit at his desk and remotely monitor the machines! So what if we left our GumballMonitor class as is, but handed it a proxy to a remote object?

Frank: I'm not sure I get it.

Jim: Me neither.

Joe: Let's start at the beginning... a proxy is a stand in for a real object. In this case, the proxy acts just like it is a Gumball Machine object, but behind the scenes it is communicating over the network to talk to the real, remote GumballMachine.

Jim: So you're saying we keep our code as it is, and we give the monitor a reference to a proxy version of the GumballMachine...

Frank: And this proxy pretends it's the real object, but it's really just communicating over the net to the real object.

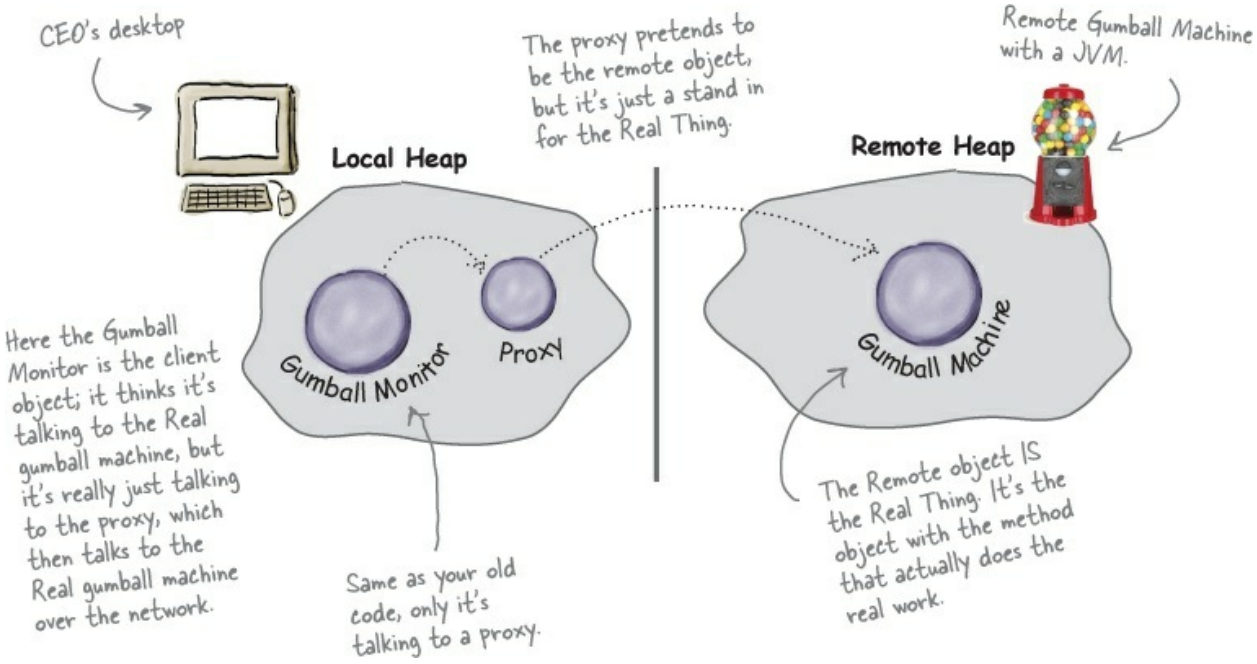
Joe: Yeah, that's pretty much the story.

Frank: It sounds like something that is easier said than done.

Joe: Perhaps, but I don't think it'll be that bad. We have to make sure that the gumball machine can act as a service and accept requests over the network; we also need to give our monitor a way to get a reference to a proxy object, but we've got some great tools already built into Java to help us. Let's talk a little more about remote proxies first...

The role of the 'remote proxy'

A remote proxy acts as a *local representative to a remote object*. What's a "remote object"? It's an object that lives in the heap of a different Java Virtual Machine (or more generally, a remote object that is running in a different address space). What's a "local representative"? It's an object that you can call local methods on and have them forwarded on to the remote object.



Your client object acts like it's making remote method calls. But what it's really doing is calling methods on a heap-local 'proxy' object that handles all the low-level details of network communication.

BRAIN POWER

Before going further, think about how you'd design a system to enable remote method invocation. How would you make it easy on the developer so that she has to write as little code as possible? How would you make the remote invocation look seamless?

BRAIN POWER

Should making remote calls be totally transparent? Is that a good idea? What might be a problem with that approach?

Adding a remote proxy to the Gumball Machine monitoring code

On paper this looks good, but how do we create a proxy that knows how to invoke a method on an object that lives in another JVM?

Hmmm. Well, you can't get a reference to something on another heap, right? In other words, you can't say:

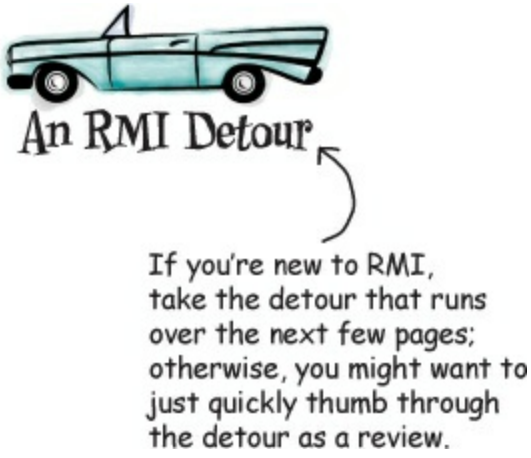
```
Duck d = <object in another heap>
```

Whatever the variable `d` is referencing must be in the same heap space as the code running the statement. So how do we approach this? Well, that's where Java's Remote Method Invocation comes in... RMI gives us a way to find objects in a remote JVM and allows us to invoke their methods.

You may have encountered RMI in Head First Java; if not, take a slight detour and get up to speed on RMI before adding the proxy support to the Gumball Machine code.

So, here's what we're going to do:

① **First, we're going to take the RMI Detour and check RMI out. Even if you are familiar with RMI, you might want to follow along and check out the scenery.**



② **Then we're going to take our GumballMachine and make it a remote service that provides a set of methods calls that can be invoked remotely.**

③ **Then, we going to create a proxy that can talk to a remote GumballMachine, again using RMI, and put the monitoring system back together so that the CEO can monitor any number of remote machines.**

Remote methods 101

Let’s say we want to design a system that allows us to call a local object that forwards each request to a remote object. How would we design it? We’d need a couple of helper objects that actually do the communicating for us. The helpers make it possible for the client to act as though it’s calling a method on a local object (which in fact, it is). The client calls a method on the client helper, as if the client helper were the actual service. The client helper then takes care of forwarding that request for us.

In other words, the client object thinks it’s calling a method on the remote service, because the client helper is pretending to be the service object. Pretending to be the thing with the method the client wants to call.

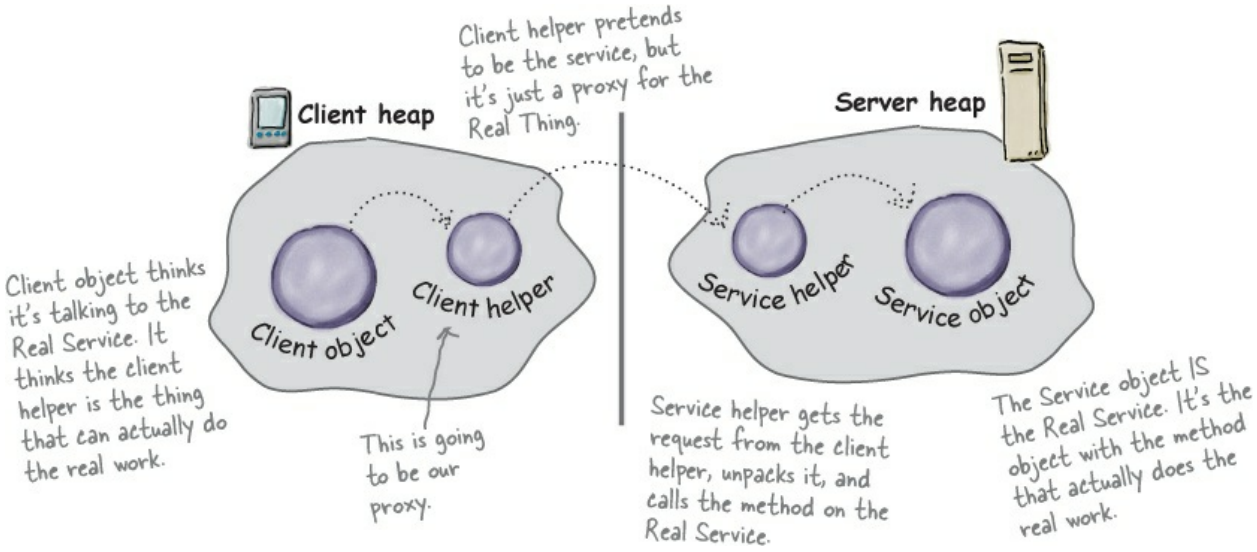
But the client helper isn’t really the remote service. Although the client helper acts like it (because it has the same method that the service is advertising), the client helper doesn’t have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the real method on the real service object. So, to the service object, the call is local. It’s coming from the service helper, not a remote client.

The service helper gets the return value from the service, packs it up, and ships it back (over a Socket’s output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

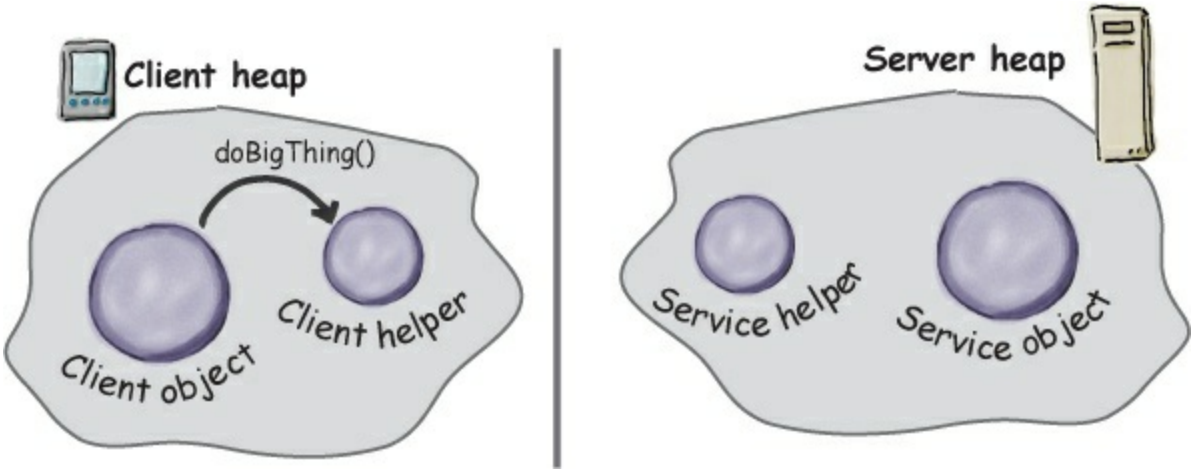
NOTE

This should look familiar...

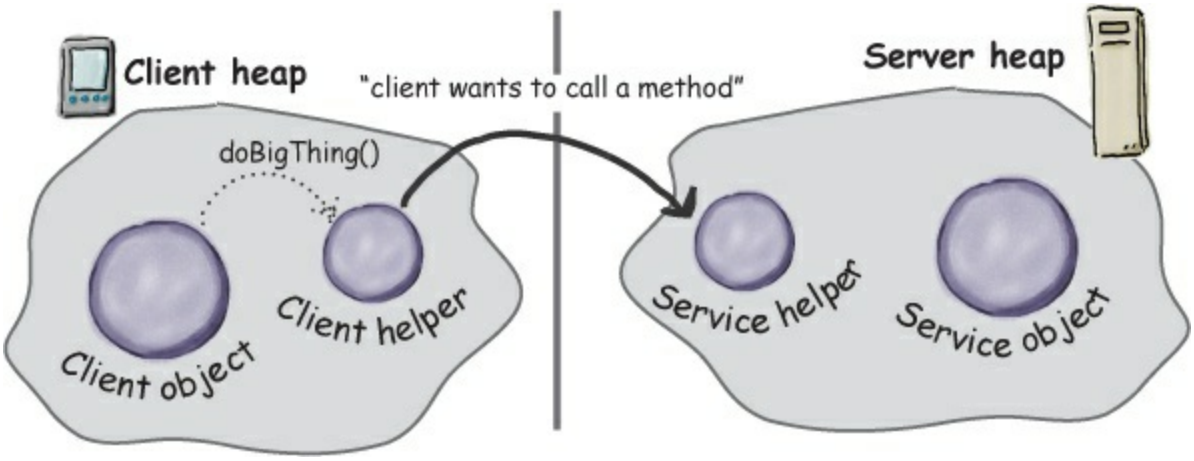


How the method call happens

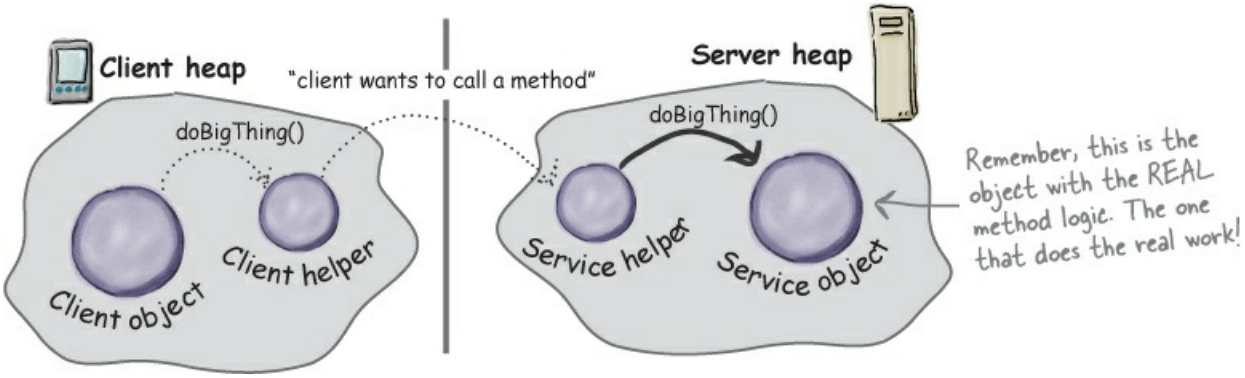
① Client object calls doBigThing() on the client helper object.



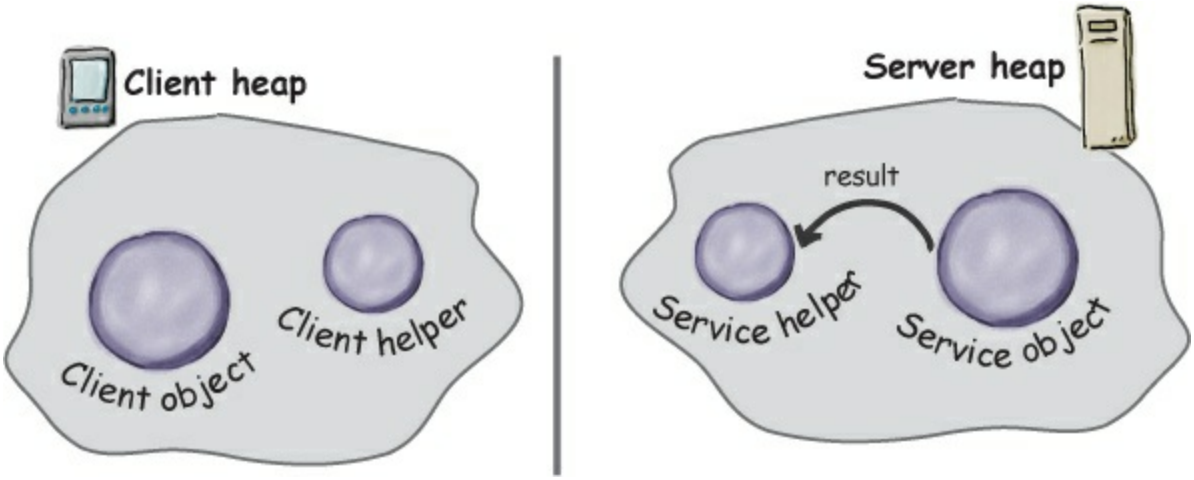
② Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



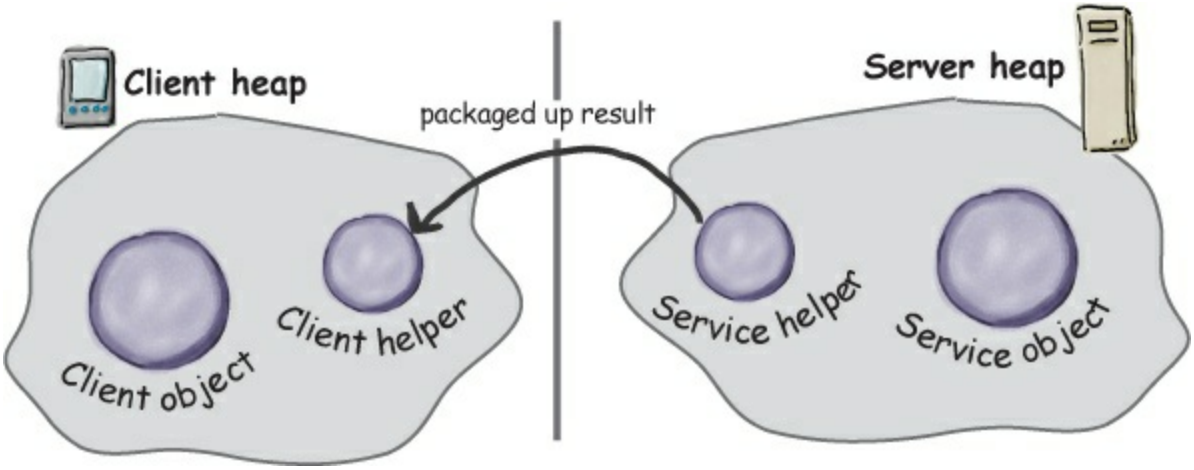
③ Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



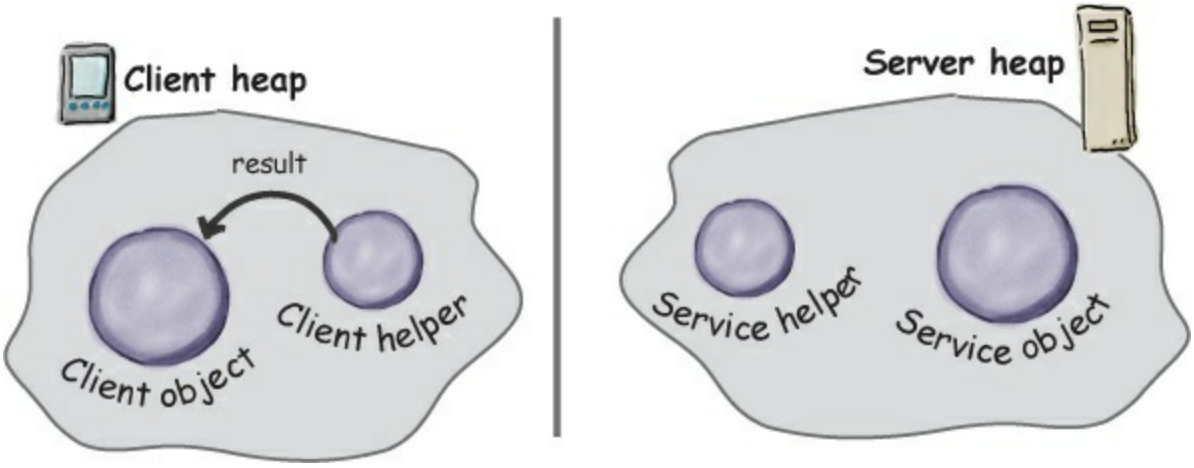
④ The method is invoked on the service object, which returns some result to the service helper.



⑤ Service helper packages up information returned from the call and ships it back over the network to the client helper.



⑥ Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.



Java RMI, the Big Picture

Okay, you've got the gist of how remote methods work; now you just need to understand how to use RMI to enable remote method invocation.

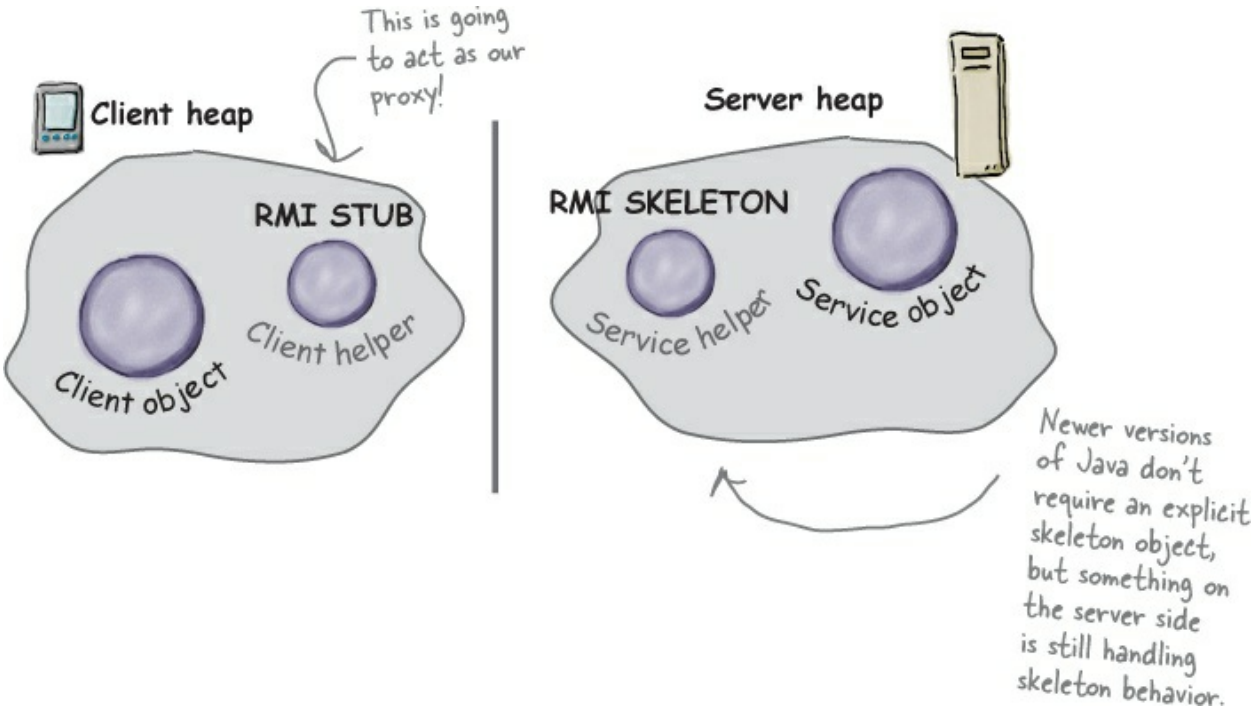
What RMI does for you is build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. The nice thing about RMI is that you don't have to write any of the networking or I/O code yourself. With your client, you call remote methods (i.e., the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

RMI also provides all the runtime infrastructure to make it all work, including a lookup service that the client can use to find and access the remote objects.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky! They can fail! And so, they throw exceptions all over the place. As a result, the client does have to acknowledge the risk. We'll see how in a few pages.

RMI Nomenclature: in RMI, the client helper is a 'stub' and the service helper is a 'skeleton'.



Now let's go through all the steps needed to make an object into a service that can accept remote calls and also the steps needed to allow a client to make remote calls.

You might want to make sure your seat belt is fastened; there are a lot of steps and a few bumps and curves — but nothing to be too worried about.

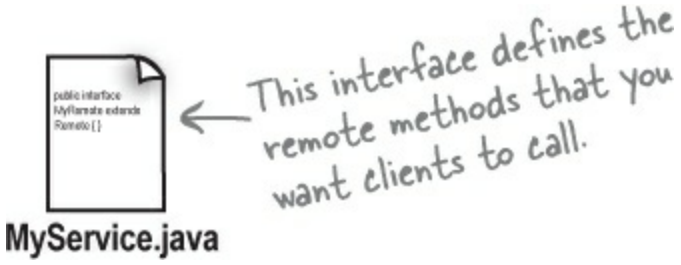
Making the Remote service


An RMI Detour

This is an **overview** of the five steps for making the remote service. In other words, the steps needed to take an ordinary object and supercharge it so it can be called by a remote client. We'll be doing this later to our GumballMachine. For now, let's get the steps down and then we'll explain each one in detail.

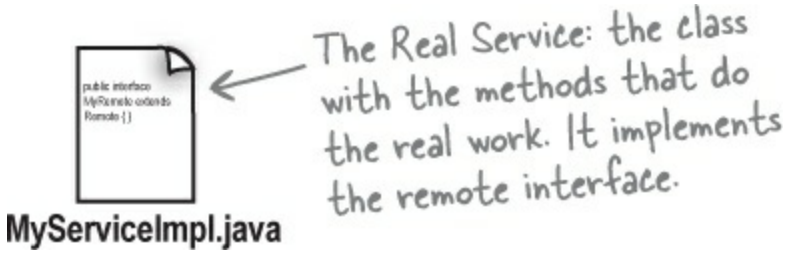
Step one:

Make a **Remote Interface**
The remote interface defines the methods that a client can call remotely. It's what the client will use as the class type for your service. Both the Stub and actual service will implement this!



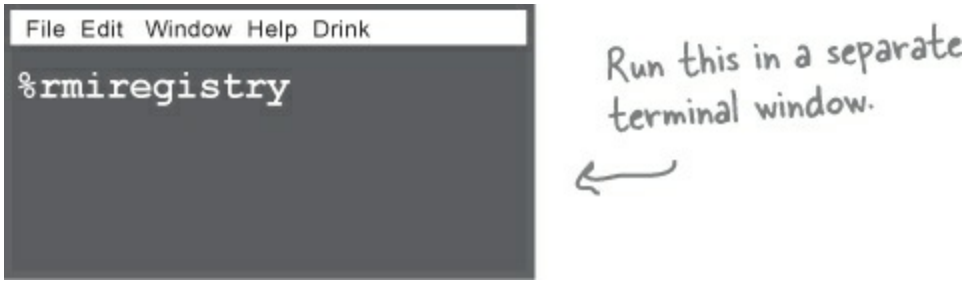
Step two:

Make a **Remote Implementation**
This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client wants to call methods on (e.g., our GumballMachine!).



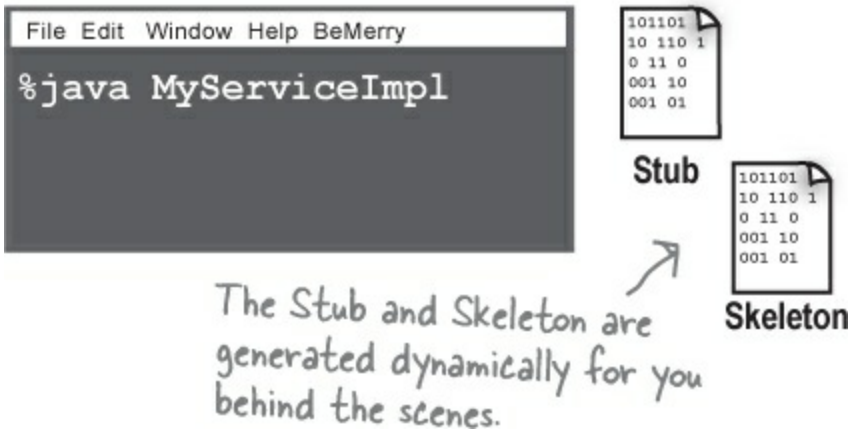
Step three:

Start the **RMI registry** (rmiregistry)
The *rmiregistry* is like the white pages of a phone book. It's where the client goes to get the proxy (the client stub/helper object).



Step four:

Start the **remote service**
You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.



Step one: make a Remote interface

① **Extend java.rmi.Remote**

Remote is a 'marker' interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say 'extends' here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```

This tells us that the interface is going to be used to support remote calls.

② Declare that all methods **throw a RemoteException**

The remote interface is the one the client uses as the type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of Bad Things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*;
```

Remote interface is in java.rmi

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Every remote method call is considered 'risky'. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

③ **Be sure arguments and return values are primitives or Serializable**
Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through Serialization. Same thing with return values. If you use primitives, Strings, and the majority of types in the API (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

> **NOTE**
>
> Check out Head First Java if you need to refresh your memory on Serializable.

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

Step two: make a Remote implementation

An RMI Detour

① **Implement the Remote interface**
Your service has to implement the remote interface — the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

② **Extend UnicastRemoteObject**
In order to work as a remote service object, your object needs some functionality related to 'being remote'. The simplest way is to extend UnicastRemoteObject (from the java.rmi.server package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;
```

UnicastRemoteObject implements Serializable so we need the serialVersionUID field.

③ Write a no-arg constructor that declares a RemoteException

Your new superclass, UnicastRemoteObject, has one little problem — its constructor throws a RemoteException. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the RemoteException. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

④ Register the service with the RMI registry

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static rebind() method of the java.rmi.Naming class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

Step three: run rmiregistry

① Bring up a terminal and start the rmiregistry.

Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your `classes` directory.

Step four: start the service

① Bring up another terminal and start your service

This might be from a main() method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

WATCH IT!

Before Java 5, we had to generate static stubs and skeletons using rmic. Now, we don't have to do this anymore and in fact, we *shouldn't* do it anymore, because *static* stubs and skeletons are deprecated.

Instead, stubs and skeletons are generated dynamically. This happens automatically when we subclass the UnicastRemoteObject (like we're doing for the MyRemoteImpl class).

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why are you showing stubs and skeletons in the diagrams for the RMI code? I thought we got rid of those way back.**

A: **A:** You're right; for the skeleton, the RMI runtime can dispatch the client calls directly to the remote service using reflection, and stubs are generated dynamically using Dynamic Proxy (which you'll learn more about a bit later in the chapter). The remote object's stub is a java.lang.reflect.Proxy instance (with an invocation handler) that is automatically generated to handle all the details of getting the local method calls by the client to the remote object. But we like to show both the stub and skeleton, because conceptually it helps you to understand that there is something under the covers that's making that communication between the client stub and the remote service happen.

Complete code for the server side

The Remote interface:

RemoteException and Remote interface are in java.rmi package.

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

Your interface MUST extend java.rmi.Remote.

All of your remote methods must declare a RemoteException.

The Remote service (the implementation):

UnicastRemoteObject is in the java.rmi.server package.

Extending UnicastRemoteObject is the easiest way to make a remote object.

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
   private static final long serialVersionUID = 1L;

   public String sayHello() {
       return "Server says, 'Hey'";
   }


   public MyRemoteImpl() throws RemoteException { }


   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("RemoteHello", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

You MUST implement your remote interface!!

You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

Your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor).

Make the remote object, then 'bind' it to the rmiregistry using the static Naming.rebind(). The name you register it under is the name clients will use to look it up in the RMI registry.

How does the client get the stub object?

The client has to get the stub object (our proxy), since that's the thing the

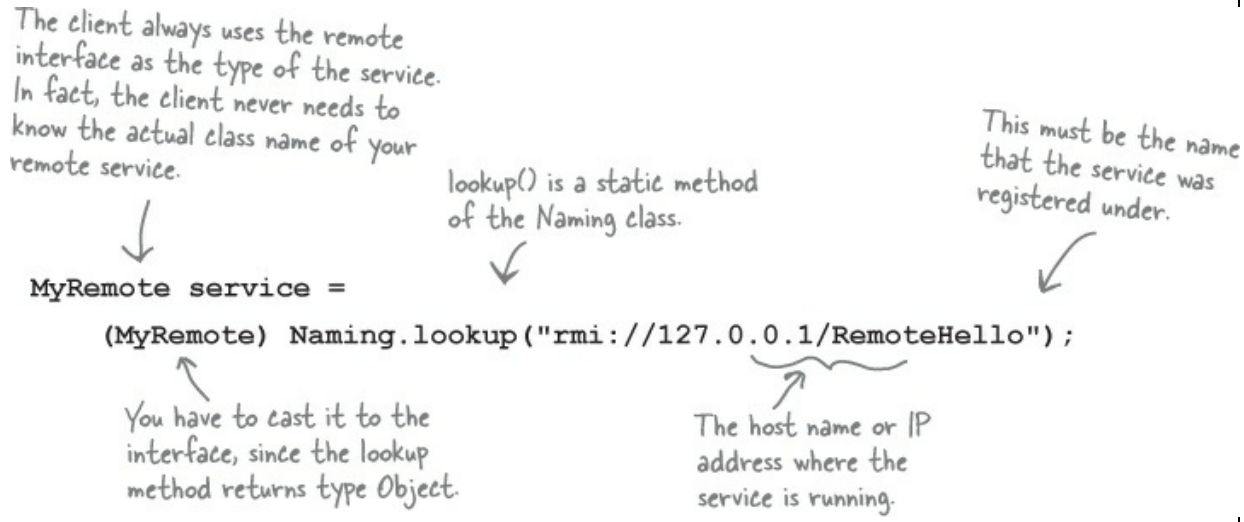
client will call methods on. And that's where the RMI registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."

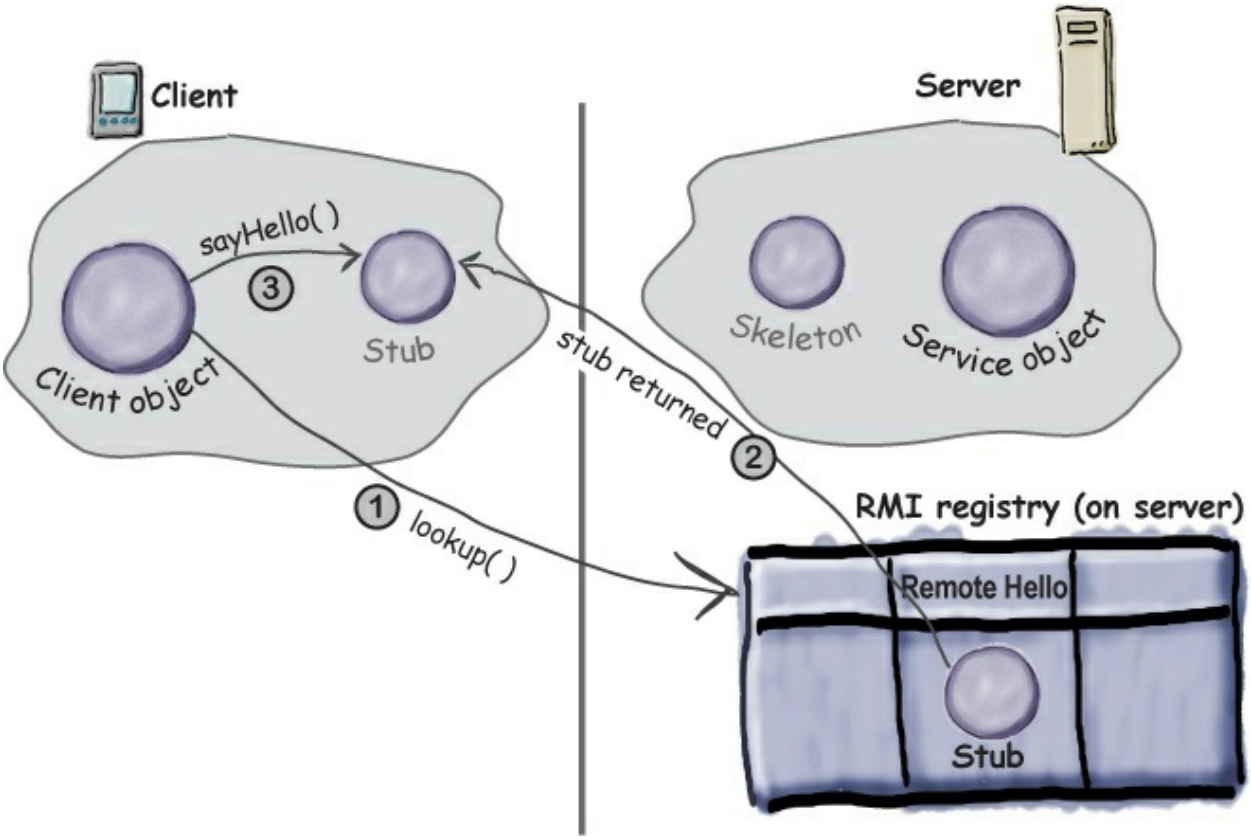
Let's take a look at the code we need to look-up and retrieve a stub object.

CODE UP CLOSE

NOTE

Here's how it works.





How it works...

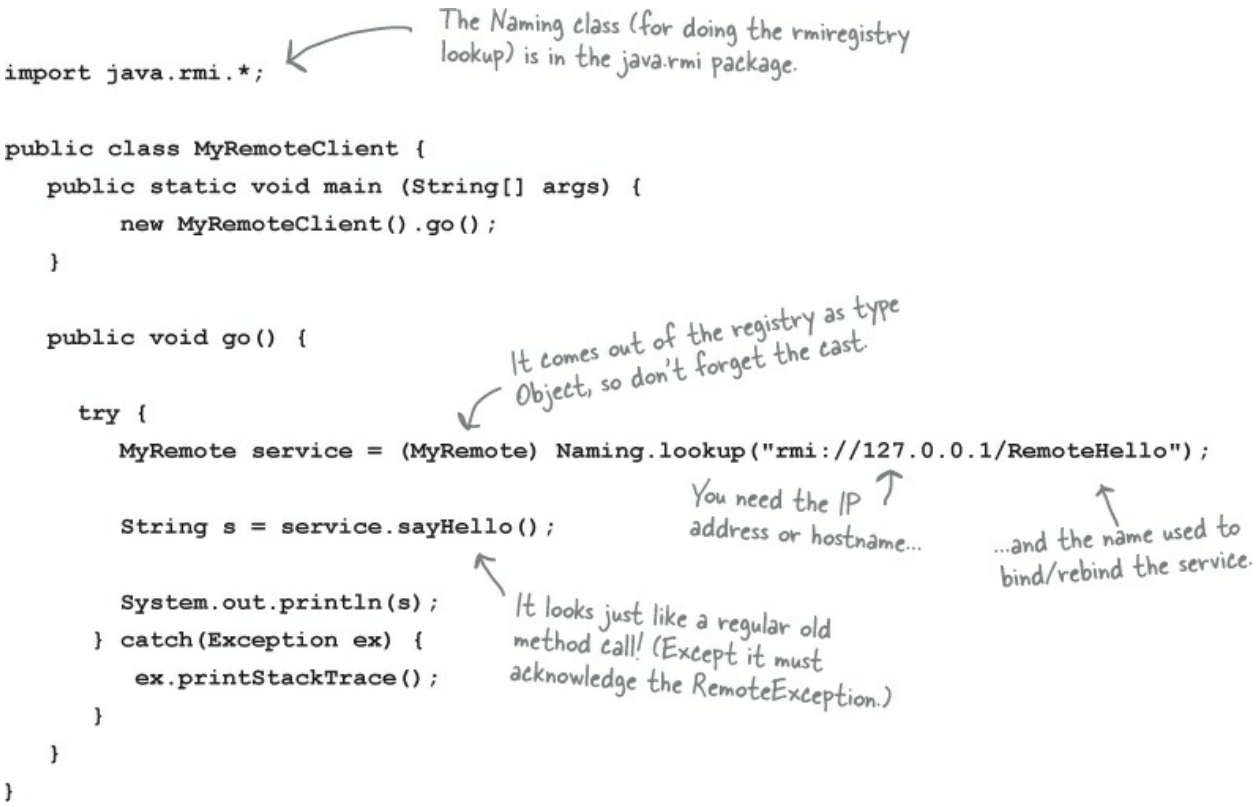
① **Client does a lookup on the RMI registry**

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

② **RMI registry returns the stub object**
(as the return value of the lookup method) and RMI deserializes the stub automatically.

③ **Client invokes a method on the stub, as if the stub IS the real service**

Complete client code



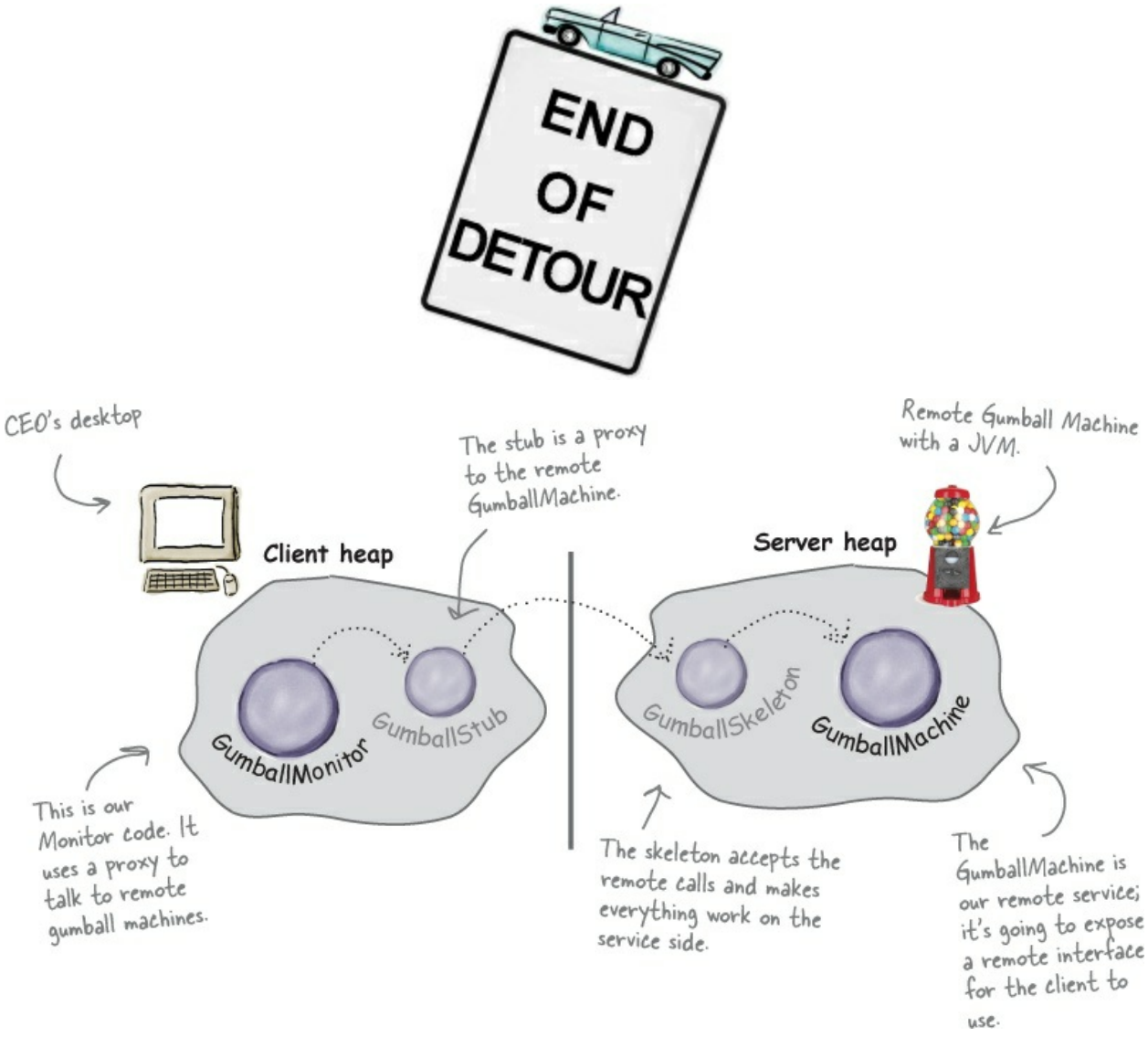
WATCH IT!

The things programmers do wrong with RMI are:

1. *Forget to start rmiregistry before starting remote service (when the service is registered using Naming.rebind(), the rmiregistry must be running!)*
2. *Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)*

Back to our GumballMachine remote proxy

Okay, now that you have the RMI basics down, you've got the tools you need to implement the gumball machine remote proxy. Let's take a look at how the GumballMachine fits into this framework:



Getting the GumballMachine ready to be a remote service

The first step in converting our code to use the remote proxy is to enable the GumballMachine to service remote requests from clients. In other words, we're going to make it into a service. To do that, we need to:

1. Create a remote interface for the GumballMachine. This will provide a set of methods that can be called remotely.
2. Make sure all the return types in the interface are serializable.
3. Implement the interface in a concrete class.

We'll start with the remote interface:

Don't forget to import java.rmi.*

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

This is the remote interface.

All return types need to be primitive or Serializable...

Here are the methods were going to support. Each one throws RemoteException.

We have one return type that isn't Serializable: the State class. Let's fix it up...

```
import java.io.*;

public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

Serializable is in the java.io package.

Then we just extend the Serializable interface (which has no methods in it). And now State in all the subclasses can be transferred over the network.

Actually, we're not done with Serializable yet; we have one problem with State. As you may remember, each State object maintains a reference to a gumball machine so that it can call the gumball machine's methods and change its state. We don't want the entire gumball machine serialized and transferred with the State object. There is an easy way to fix this:

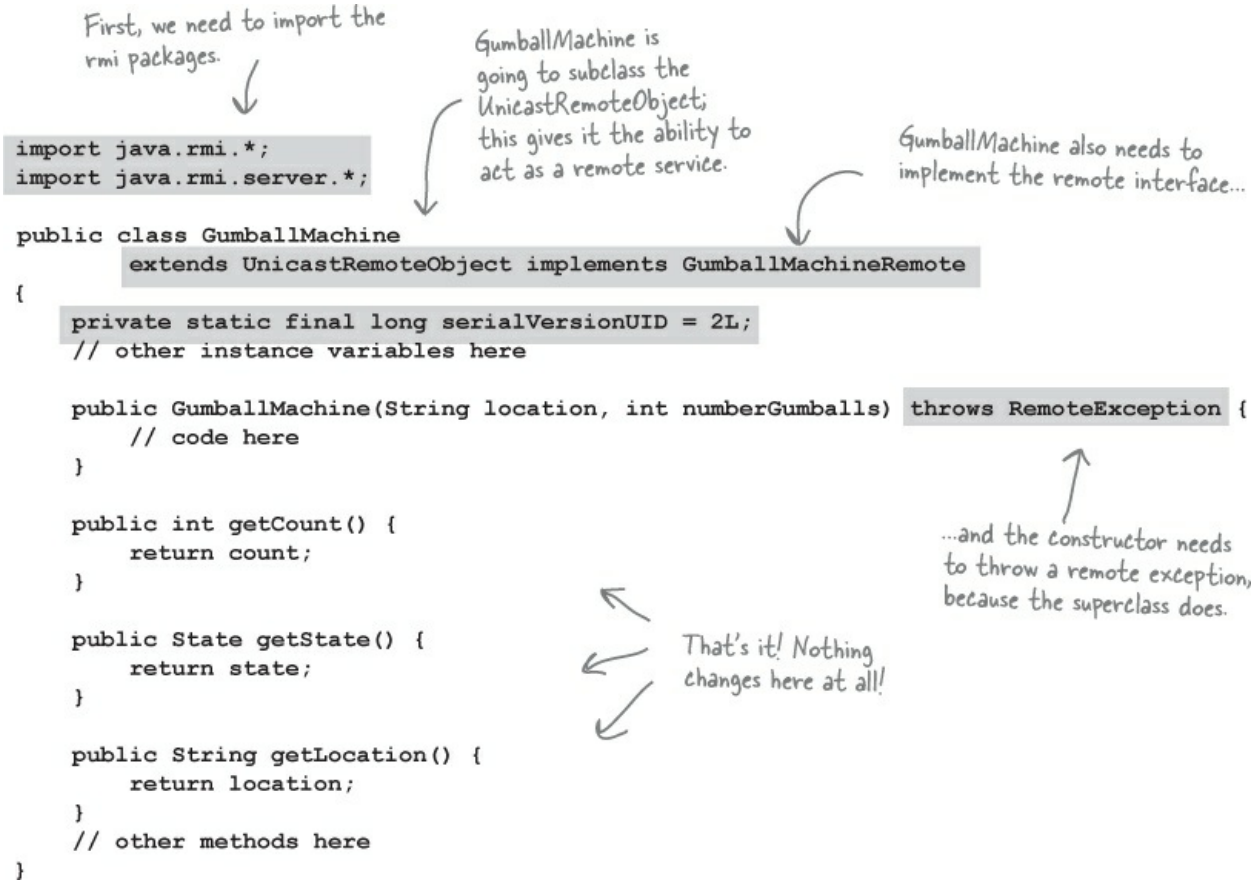
```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;
    // all other methods here
}
```

In each implementation of State, we add the serialVersionUID and the transient keyword to the GumballMachine instance variable. The transient keyword tells the JVM not to serialize this field. Note that this can be slightly dangerous if you try to access this field once the object's been serialized and transferred.

We've already implemented our GumballMachine, but we need to make sure

it can act as a service and handle requests coming from over the network. To do that, we have to make sure the GumballMachine is doing everything it needs to implement the GumballMachineRemote interface.

As you've already seen in the RMI detour, this is quite simple; all we need to do is add a couple of things...




```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    private static final long serialVersionUID = 2L;
    // other instance variables here

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // code here
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }
    // other methods here
}
```

Registering with the RMI registry...

That completes the gumball machine service. Now we just need to fire it up so it can receive requests. First, we need to make sure we register it with the RMI registry so that clients can locate it.

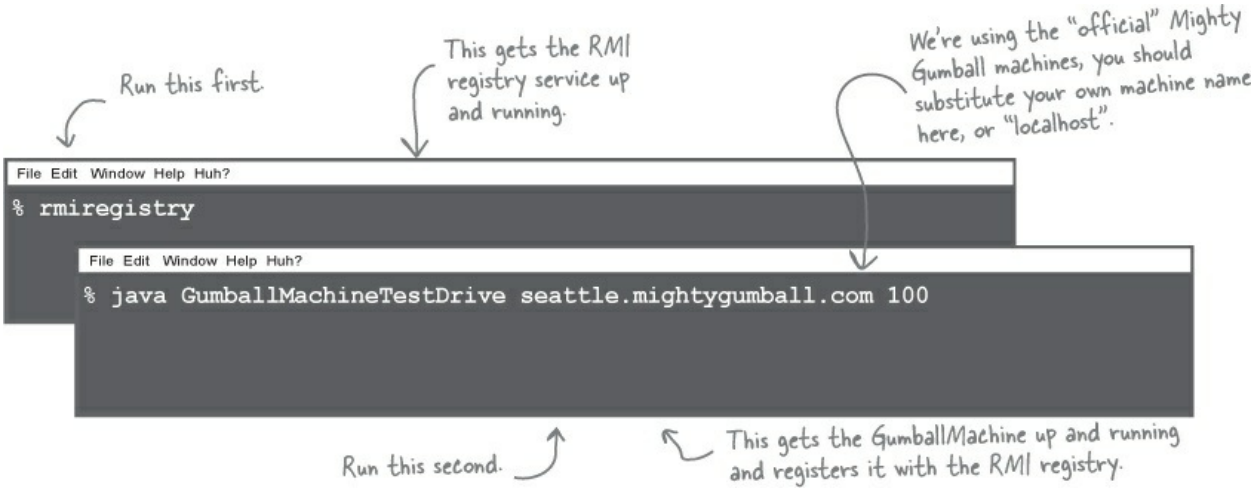
We're going to add a little code to the test drive that will take care of this for us:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine = new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

First we need to add a try/catch block around the gumball instantiation because our constructor can now throw exceptions.

We also add the call to Naming.rebind, which publishes the GumballMachine stub under the name gumballmachine.

Let's go ahead and get this running...



Now for the GumballMonitor client...

Remember the GumballMonitor? We wanted to reuse it without having to rewrite it to work over a network. Well, we're pretty much going to do that, but we do need to make a few changes.

```
import java.rmi.*;
```

We need to import the RMI package because we are using the RemoteException class below...

```
public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Now we're going to rely on the remote interface rather than the concrete GumballMachine class.

We also need to catch any remote exceptions that might happen as we try to invoke methods that are ultimately happening over the network.

Writing the Monitor test drive

Now we've got all the pieces we need. We just need to write some code so the CEO can monitor a bunch of gumball machines:

Here's the monitor test drive. The CEO is going to run this!

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0; i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for (int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

Here's all the locations were going to monitor.

We create an array of locations, one for each machine.

We also create an array of monitors.

Now we need to get a proxy to each remote machine.

Then we iterate through each machine and print out its report.

CODE UP CLOSE

This returns a proxy to the remote Gumball Machine (or throws an exception if one can't be located).

Remember, Naming.lookup() is a static method in the RMI package that takes a location and service name and looks it up in the rmiregistry at that location.

```
try {
    GumballMachineRemote machine =
                (GumballMachineRemote) Naming.lookup(location[i]);

    monitor[i] = new GumballMonitor(machine);

} catch (Exception e) {
    e.printStackTrace();
}
```

Once we get a proxy to the remote machine, we create a new GumballMonitor and pass it the machine to monitor.

Another demo for the CEO of Mighty Gumball...

Okay, it's time to put all this work together and give another demo. First let's make sure a few gumball machines are running the new code:

On each machine, run rmiregistry in the background or from a separate terminal window...

...and then run the GumballMachine, giving it a location and an initial gumball count.

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive santafe.mightygumball.com 100

File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive boulder.mightygumball.com 100

File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive seattle.mightygumball.com 250
```

Popular machine!

And now let's put the monitor in the hands of the CEO. Hopefully, this time he'll love it

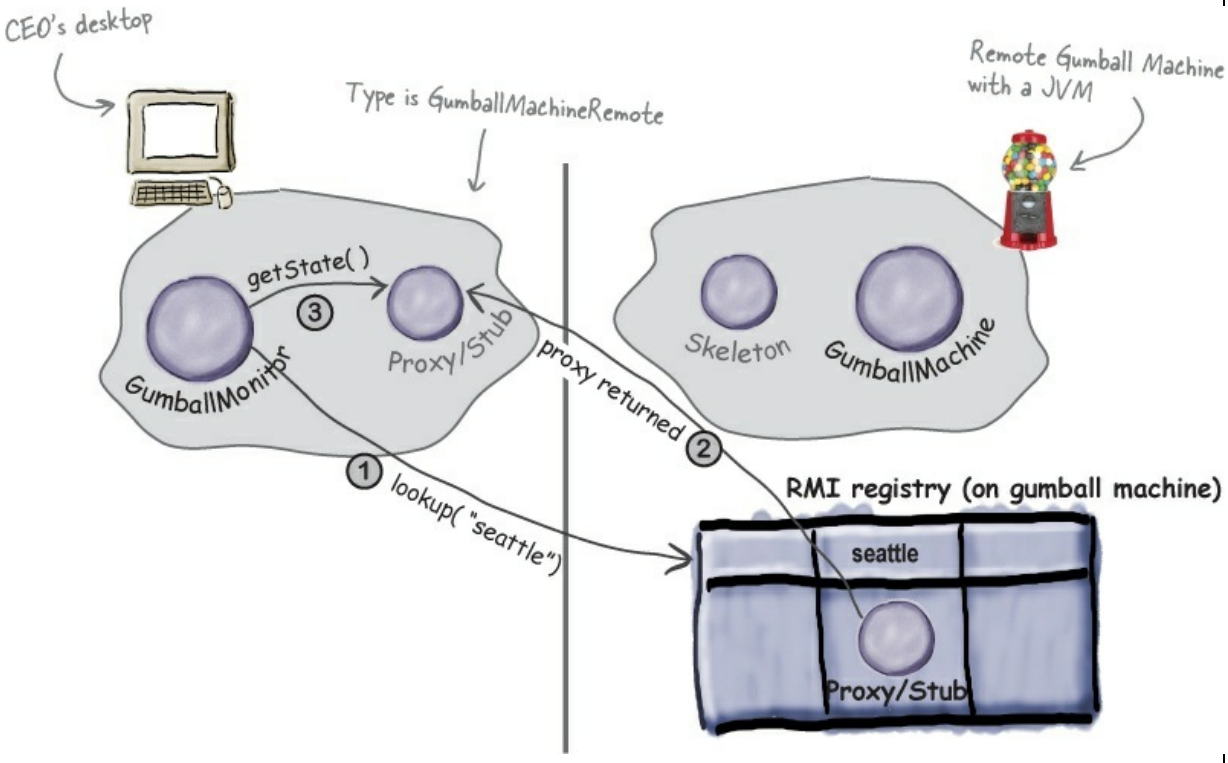
```
File Edit Window Help GumballsAndBeyond
% java GumballMonitorTestDrive
Gumball Machine: santafe.mightygumball.com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of crank

Gumball Machine: seattle.mightygumball.com
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

The monitor iterates over each remote machine and calls its getLocation(), getCount() and getState() methods.

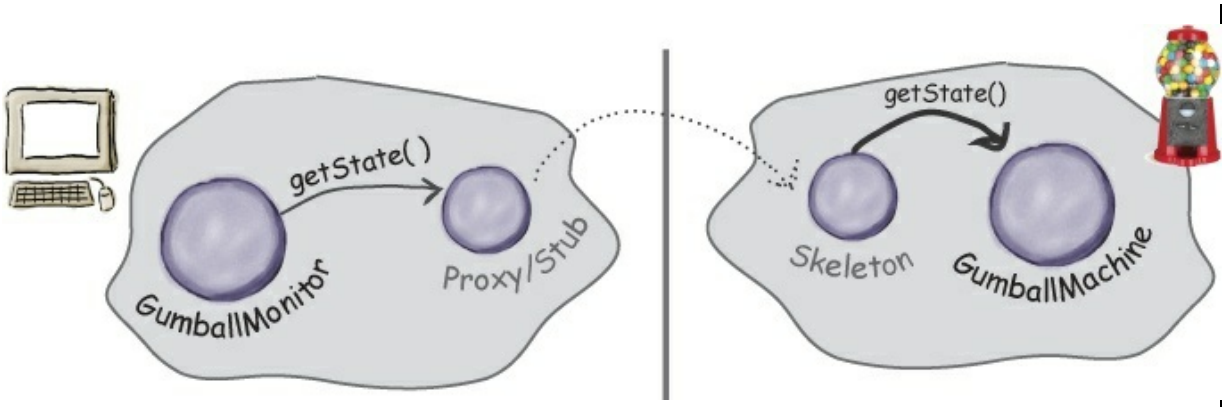
By invoking methods on the proxy, we make a remote call across the wire, and get back a String, an integer, and a State object. Because we are using a proxy, the GumballMonitor doesn't know, or care, that calls are remote (other than having to worry about remote exceptions).

BEHIND THE SCENES

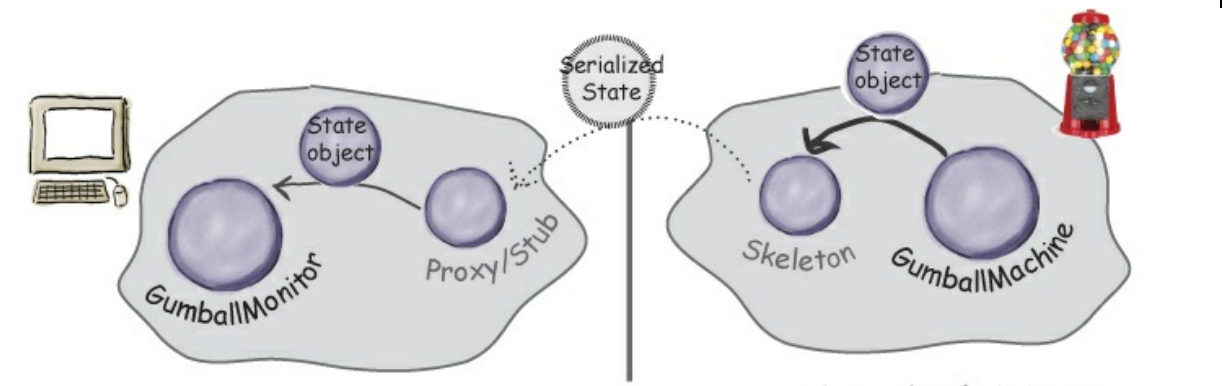
① The CEO runs the monitor, which first grabs the proxies to the remote gumball machines and then calls getState() on each one (along with getCount() and getLocation()).



② getState() is called on the proxy, which forwards the call to the remote service. The skeleton receives the request and then forwards it to the gumball machine.



③ GumballMachine returns the state to the skeleton, which serializes it and transfers it back over the wire to the proxy. The proxy deserializes it and returns it as an object to the monitor.



NOTE

We also have a small bit of code to register and locate stubs using the RMI registry. But no matter what, if we were writing something to work over the Internet, we'd need some kind of locator service.

The Proxy Pattern defined

We've already put a lot of pages behind us in this chapter; as you can see, explaining the Remote Proxy is quite involved. Despite that, you'll see that the definition and class diagram for the Proxy Pattern is actually fairly

straightforward. Note that Remote Proxy is one implementation of the general Proxy Pattern; there are actually quite a few variations of the pattern, and we'll talk about them later. For now, let's get the details of the general pattern down.

Here's the Proxy Pattern definition:

> **Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing.**

NOTE

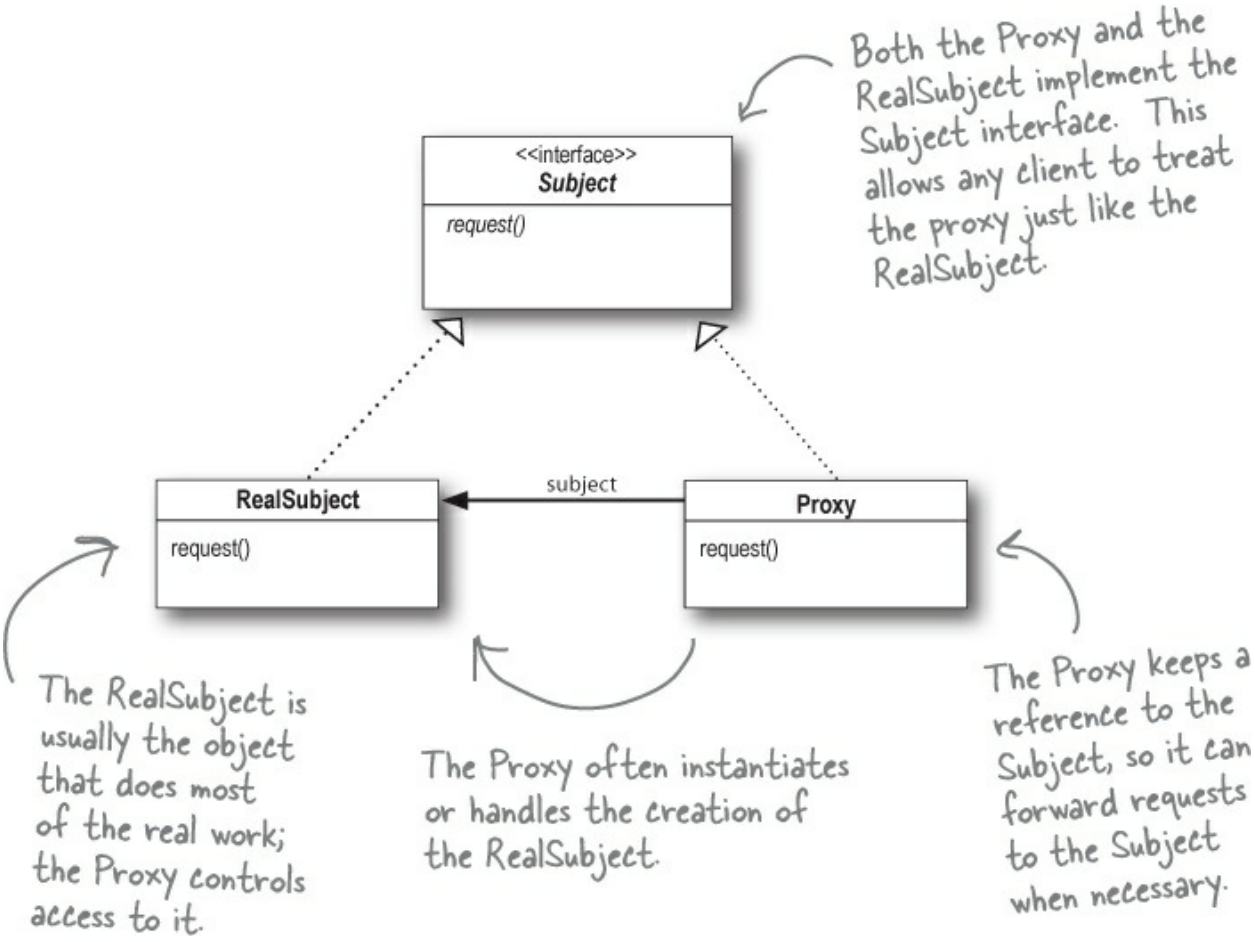
The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a "representative" for another object.

But what about a proxy controlling access? That sounds a little strange. No worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy "controls access." We're going to talk more about this later, but for now here are a few ways proxies control access:

- As we know, a remote proxy controls access to a remote object.
- A virtual proxy controls access to a resource that is expensive to create.
- A protection proxy controls access to a resource based on access rights.

Now that you've got the gist of the general pattern, check out the class diagram...



Let’s step through the diagram...

First we have a Subject, which provides an interface for the RealSubject and the Proxy. By implementing the same interface, the Proxy can be substituted for the RealSubject anywhere it occurs.

The RealSubject is the object that does the real work. It’s the object that the Proxy represents and controls access to.

The Proxy holds a reference to the RealSubject. In some cases, the Proxy may be responsible for creating and destroying the RealSubject. Clients interact with the RealSubject through the Proxy. Because the Proxy and RealSubject implement the same interface (Subject), the Proxy can be substituted anywhere the subject can be used. The Proxy also controls access to the RealSubject; this control may be needed if the Subject is running on a remote machine, if the Subject is expensive to create in some way or if access to the subject needs to be protected in some way.

Now that you understand the general pattern, let’s look at some other ways of

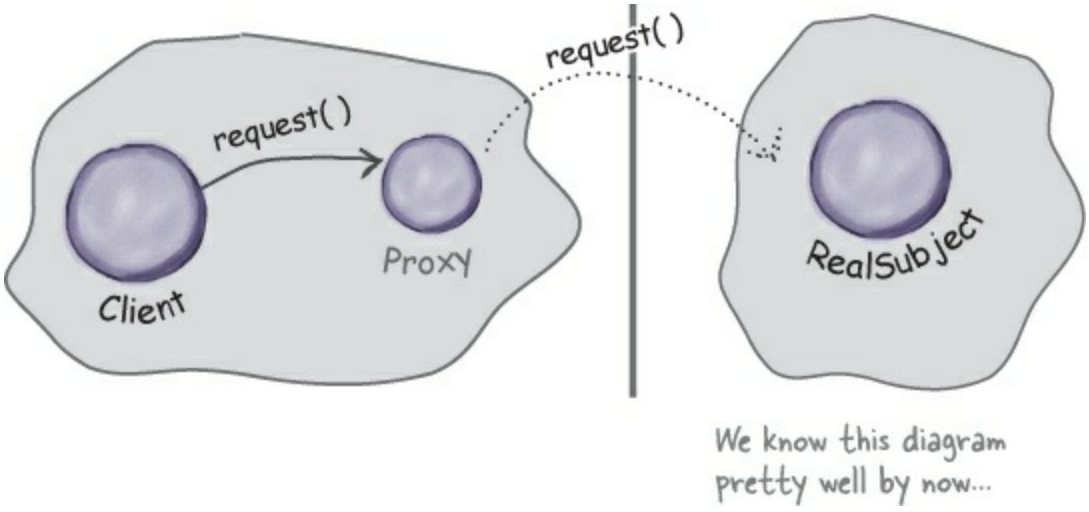
using proxy beyond the Remote Proxy...

Get ready for Virtual Proxy

Okay, so far you've seen the definition of the Proxy Pattern and you've taken a look at one specific example: the *Remote Proxy*. Now we're going to take a look at a different type of proxy, the *Virtual Proxy*. As you'll discover, the Proxy Pattern can manifest itself in many forms, yet all the forms follow roughly the general proxy design. Why so many forms? Because the Proxy Pattern can be applied to a lot of different use cases. Let's check out the Virtual Proxy and compare it to Remote Proxy:

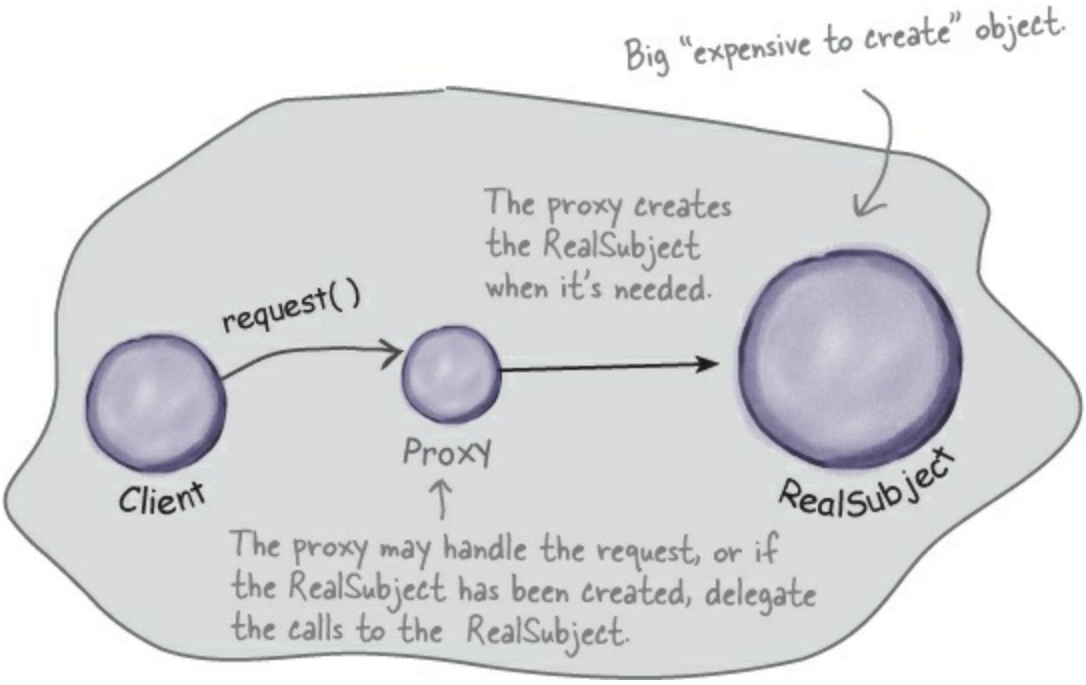
Remote Proxy

With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.



Virtual Proxy

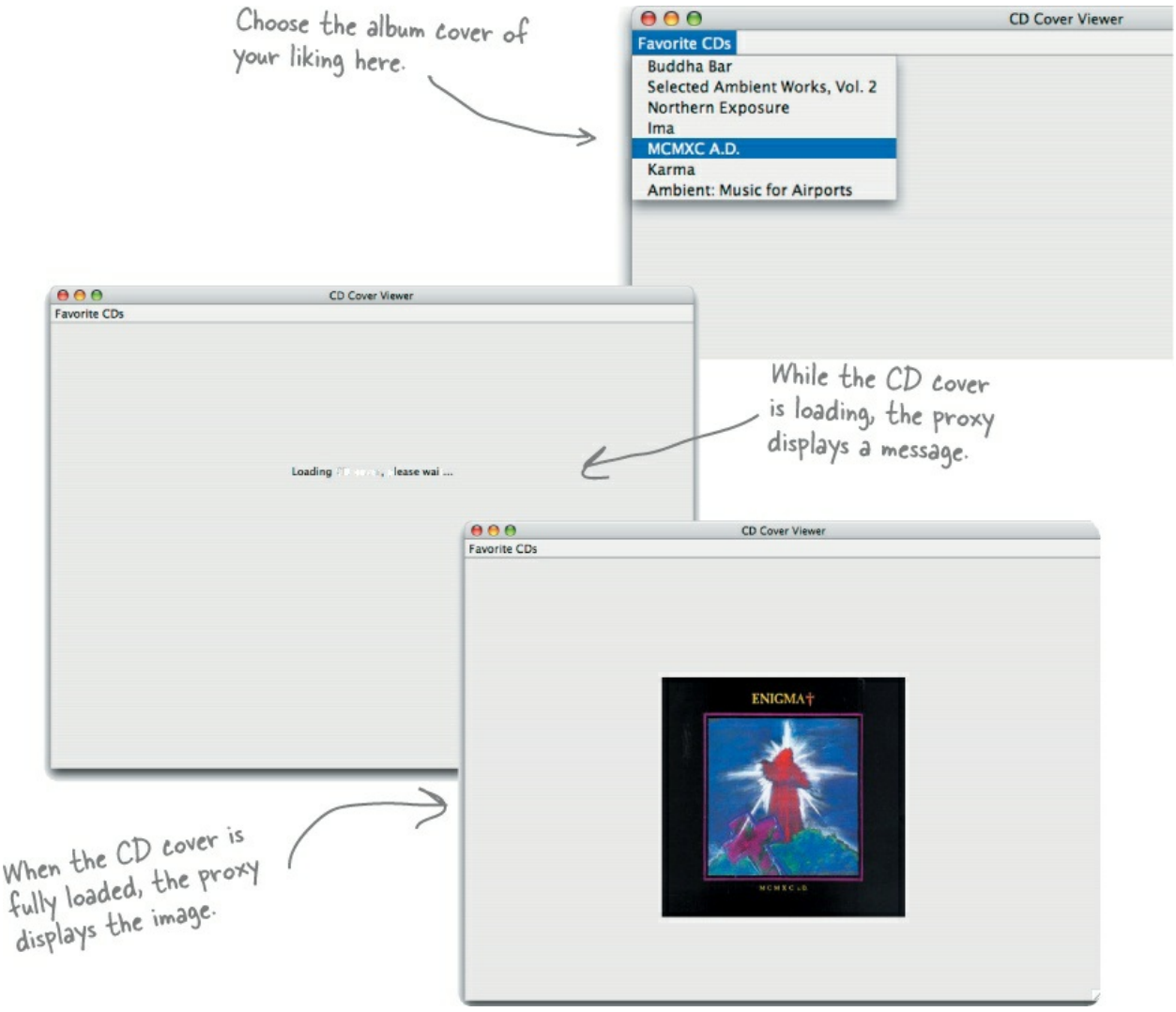
Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.



Displaying CD covers

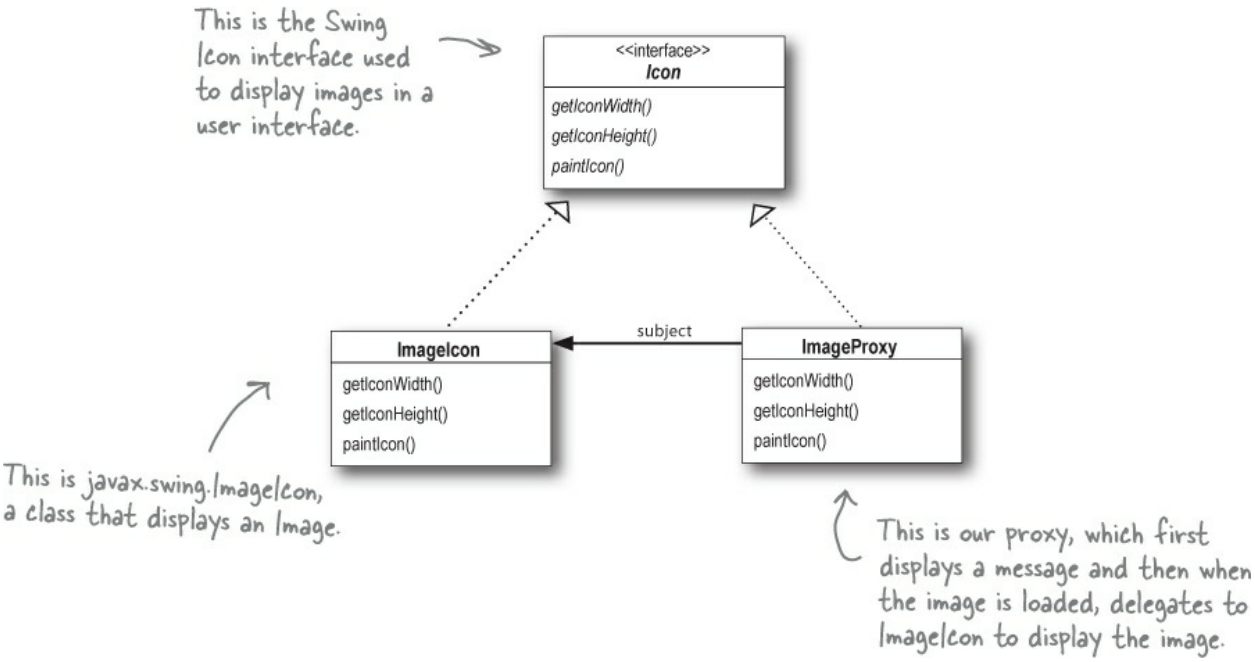
Let's say you want to write an application that displays your favorite compact disc covers. You might create a menu of the CD titles and then retrieve the images from an online service like Amazon.com. If you're using Swing, you might create an Icon and ask it to load the image from the network. The only problem is, depending on the network load and the bandwidth of your connection, retrieving a CD cover might take a little time, so your application should display something while you are waiting for the image to load. We also don't want to hang up the entire application while it's waiting on the image. Once the image is loaded, the message should go away and you should see the image.

An easy way to achieve this is through a virtual proxy. The virtual proxy can stand in place of the icon, manage the background loading, and before the image is fully retrieved from the network, display "Loading CD cover, please wait...". Once the image is loaded, the proxy delegates the display to the Icon.



Designing the CD cover Virtual Proxy

Before writing the code for the CD Cover Viewer, let's look at the class diagram. You'll see this looks just like our Remote Proxy class diagram, but here the proxy is used to hide an object that is expensive to create (because we need to retrieve the data for the Icon over the network) as opposed to an object that actually lives somewhere else on the network.

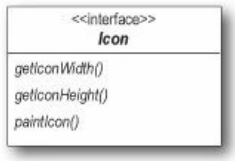


How ImageProxy is going to work

① ImageProxy first creates an ImageIcon and starts loading it from a network URL.
② While the bytes of the image are being retrieved, ImageProxy displays “Loading CD cover, please wait...”.
③ When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(), getWidth() and getHeight().
④ If the user requests a new image, we’ll create a new proxy and start the process over.

Writing the Image Proxy

The ImageProxy implements the Icon interface.



The imageIcon is the REAL icon that we eventually want to display when it's loaded.

```
class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }
    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }
    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    synchronized void setImageIcon(ImageIcon imageIcon) {
        this.imageIcon = imageIcon;
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            setImageIcon(new ImageIcon(imageURL, "CD Cover"));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

We pass the URL of the image into the constructor. This is the image we need to display once it's loaded!

We return a default width and height until the imageIcon is loaded; then we turn it over to the imageIcon.

imageIcon is used by two different threads so along with making the variable volatile (to protect reads), we use a synchronized setter (to protect writes).

Here's where things get interesting. This code paints the icon on the screen (by delegating to the imageIcon). However, if we don't have a fully created ImageIcon, then we create one. Let's look at this closer on the next page...

CODE UP CLOSE

This method is called when it's time to paint the icon on the screen.

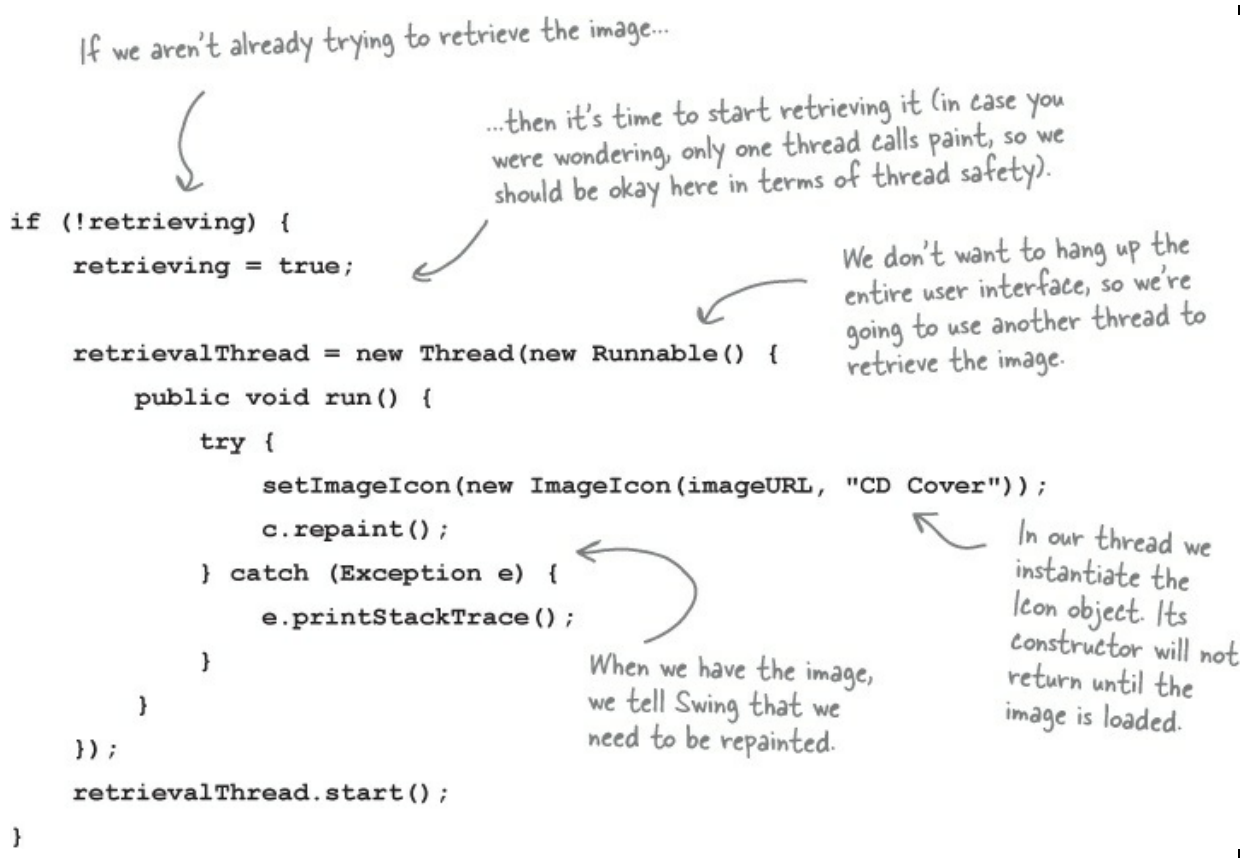
```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
     if (imageIcon != null) {

         imageIcon.paintIcon(c, g, x, y);

     } else {

         g.drawString("Loading CD cover, please wait...", x+300, y+190);
         if (!retrieving) {

             retrieving = true;
             retrievalThread = new Thread(new Runnable() {
                 public void run() {
                     try {
                         setImageIcon(new ImageIcon(imageURL, "CD Cover"));
                         c.repaint();
                     } catch (Exception e) {
                         e.printStackTrace();
                     }
                 }
             });

             retrievalThread.start();
         }
     }
 }
```

If we've got an icon already, we go ahead and tell it to paint itself.

Otherwise we display the "loading" message.

Here's where we load the REAL icon image. Note that the image loading with IconImage is synchronous: the ImageIcon constructor doesn't return until the image is loaded. That doesn't give us much of a chance to do screen updates and have our message displayed, so we're going to do this asynchronously. See the "Code Way Up Close" on the next page for more...

CODE WAY UP CLOSE



NOTE

So, the next time the display is painted after the ImageIcon is instantiated, the paintIcon method will paint the image, not the loading message.

DESIGN PUZZLE

The ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

```
class ImageProxy implements Icon {
    // instance variables & constructor here

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            // more code here
        }
    }
}
```

Two states

Two states

Two states

Testing the CD Cover Viewer

READY BAKE CODE

Okay, it's time to test out this fancy new virtual proxy. Behind the scenes we've been baking up a new ImageProxyTestDrive that sets up the window, creates a frame, installs the menus and creates our proxy. We don't go through all that code in gory detail here, but you can always grab the source code and have a look, or check it out at the end of the chapter where we list all the source code for the Virtual Proxy.

Here's a partial view of the test drive code:

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception {

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Here we create an image proxy and set it to an initial URL. Whenever you choose a selection from the CD menu, you'll get a new image proxy.

Next we wrap our proxy in a component so it can be added to the frame. The component will take care of the proxy's width, height and similar details.

Finally we add the proxy to the frame so it can be displayed.

Now let's run the test drive:

```
File Edit Window Help JustSomeOfTheCDsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

Running ImageProxyTestDrive should give you a window like this.

Things to try...

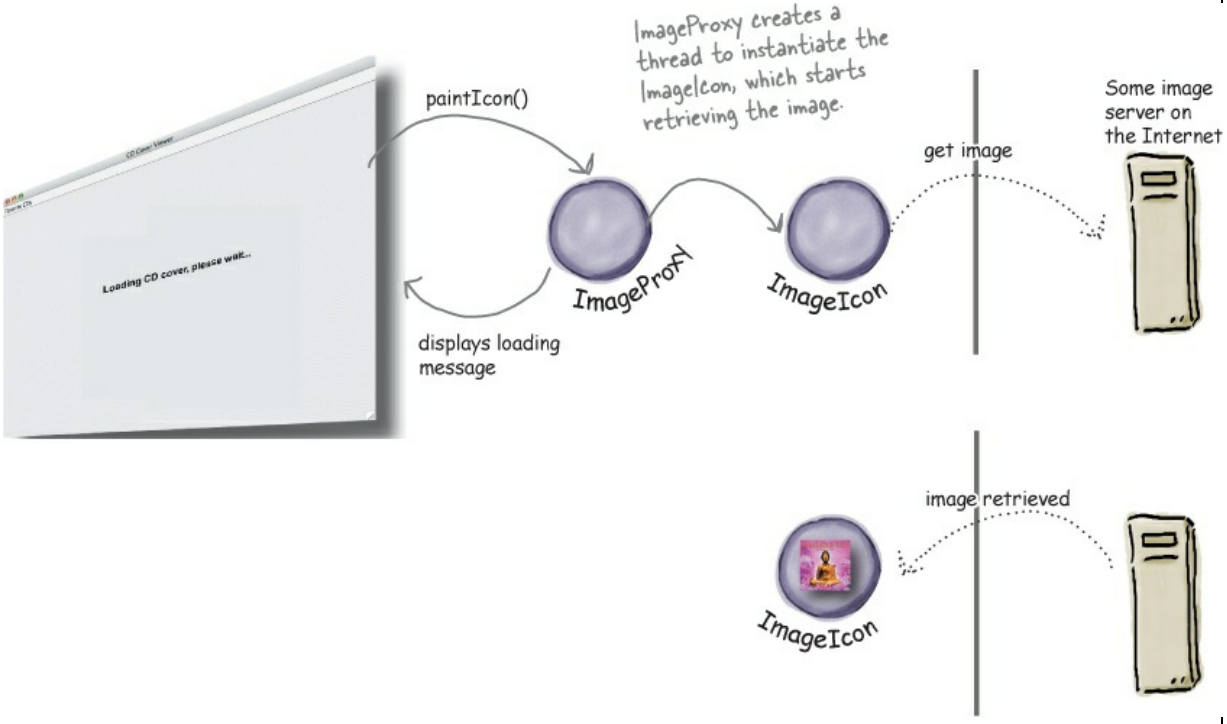
① Use the menu to load different CD covers; watch the proxy display "loading" until the image has arrived.
② Resize the window as the "loading" message is displayed. Notice that the proxy is handling the loading without hanging up the Swing window.
③ Add your own favorite CDs to the ImageProxyTestDrive.

What did we do?

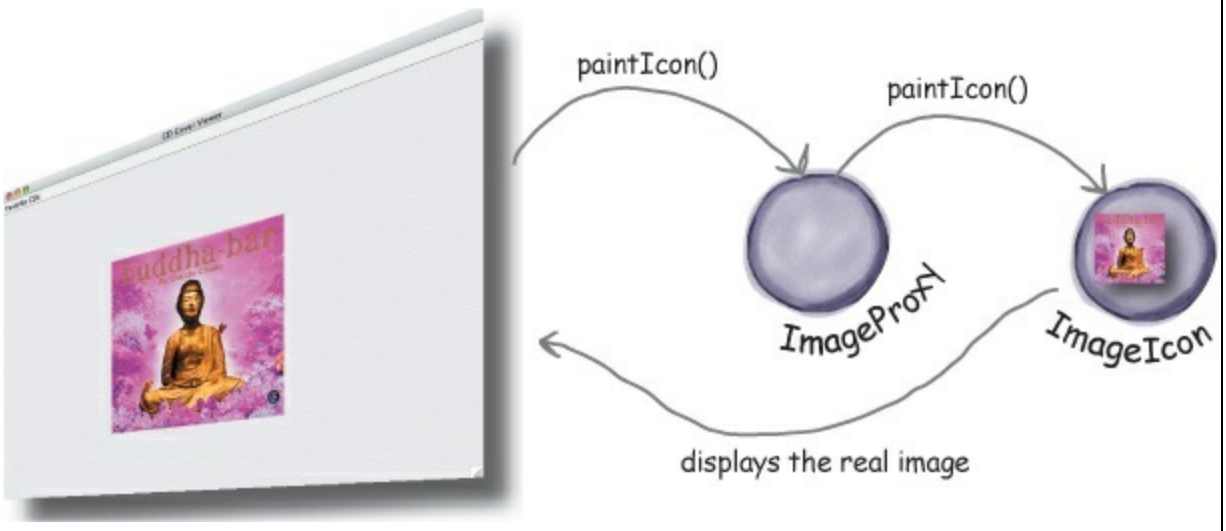
BEHIND THE SCENES

① We created an ImageProxy for the display. The paintIcon() method is called and

ImageProxy fires off a thread to retrieve the image and create the ImageIcon.



② At some point the image is returned and the ImageIcon fully instantiated.
③ After the ImageIcon is created, the next time paintIcon() is called, the proxy delegates to the ImageIcon.



THERE ARE NO DUMB QUESTIONS

Q: **Q: The Remote Proxy and Virtual Proxy seem so different to me; are they really ONE pattern?**

A: **A:** You'll find a lot of variants of the Proxy Pattern in the real world; what they all have in common is that they intercept a method invocation that the client is making on the subject. This level of indirection allows us to do many things, including dispatching requests to a remote subject, providing a representative for an expensive

object as it is created, or, as you'll see, providing some level of protection that can determine which clients should be calling which methods. That's just the beginning; the general Proxy Pattern can be applied in many different ways, and we'll cover some of the other ways at the end of the chapter.

Q: **Q: ImageProxy seems just like a Decorator to me. I mean, we are basically wrapping one object with another and then delegating the calls to the ImageIcon. What am I missing?**

A: **A:** Sometimes Proxy and Decorator look very similar, but their purposes are different: a decorator adds behavior to a class, while a proxy controls access to it. You might ask, "Isn't the loading message adding behavior?" In some ways it is; however, more importantly, the ImageProxy is controlling access to an ImageIcon. How does it control access? Well, think about it this way: the proxy is decoupling the client from the ImageIcon. If they were coupled the client would have to wait until each image is retrieved before it could paint its entire interface. The proxy controls access to the ImageIcon so that before it is fully created, the proxy provides another on screen representation. Once the ImageIcon is created the proxy allows access.

Q: **Q: How do I make clients use the Proxy rather than the Real Subject?**

A: **A:** Good question. One common technique is to provide a factory that instantiates and returns the subject. Because this happens in a factory method we can then wrap the subject with a proxy before returning it. The client never knows or cares that it's using a proxy instead of the real thing.

Q: **Q: I noticed in the ImageProxy example, you always create a new ImageIcon to get the image, even if the image has already been retrieved. Could you implement something similar to the ImageProxy that caches past retrievals?**

A: **A:** You are talking about a specialized form of a Virtual Proxy called a Caching Proxy. A caching proxy maintains a cache of previously created objects and when a request is made it returns cached object, if possible. We're going to look at this and at several other variants of the Proxy Pattern at the end of the chapter.

Q: **Q: I see how Decorator and Proxy relate, but what about Adapter? An adapter seems very similar as well.**

A: **A:** Both Proxy and Adapter sit in front of other objects and forward requests to them. Remember that Adapter changes the interface of the objects it adapts, while the Proxy implements the same interface.
There is one additional similarity that relates to the Protection Proxy. A Protection Proxy may allow or disallow a client access to particular methods in an object based on the role of the client. In this way a Protection Proxy may only provide a partial interface to a client, which is quite similar to some Adapters. We are going to take a look at Protection Proxy in a few pages.

FIRESIDE CHATS

Tonight's talk: **Proxy and Decorator get intentional.**

Proxy:	Decorator:
Hello, Decorator. I presume you're here because people sometimes get us confused?	
	Well, I think the reason people get us confused is that you go around pretending to be an entirely different pattern, when in fact, you're just a Decorator in disguise. I really don't think you should be copying all my ideas.
Me copying *your* ideas? Please. I control access to objects. You just decorate them. My job is so much more important than yours it's just not even funny.	

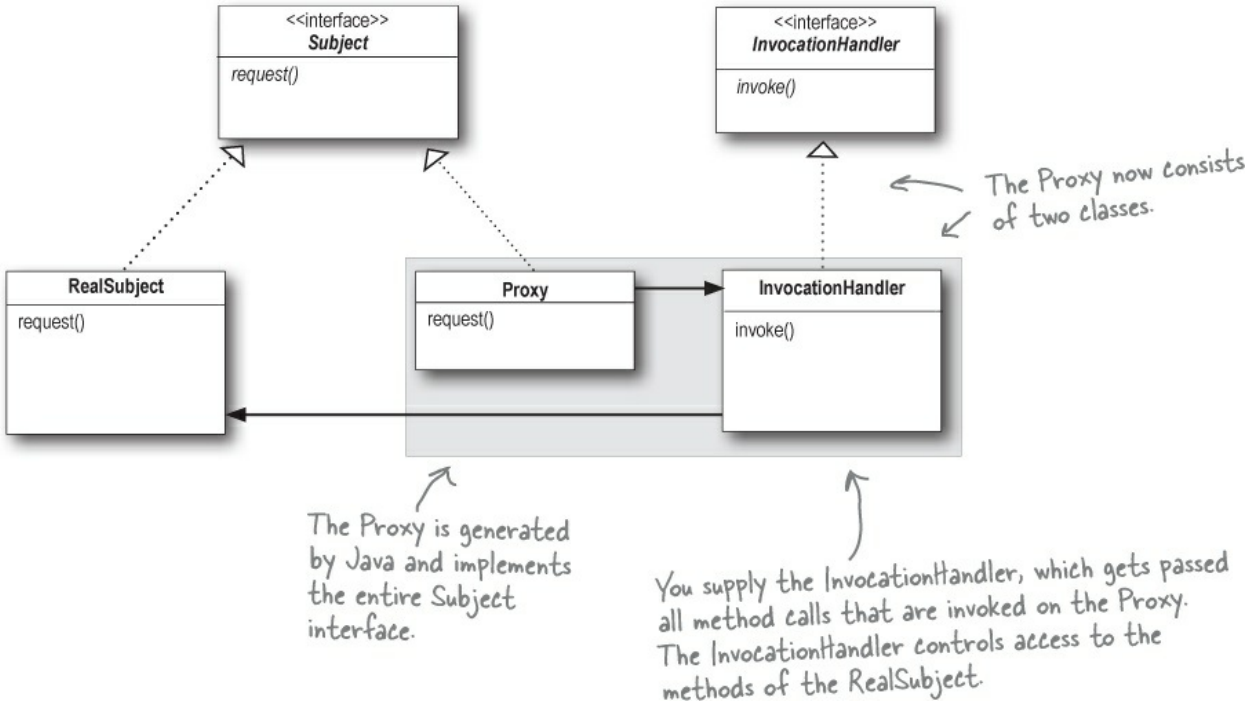
	“Just” decorate? You think decorating is some frivolous, unimportant pattern? Let me tell you buddy, I add *behavior*. That’s the most important thing about objects — what they do!
Fine, so maybe you’re not entirely frivolous... but I still don’t get why you think I’m copying all your ideas. I’m all about representing my subjects, not decorating them.	
	You can call it “representation” but if it looks like a duck and walks like a duck... I mean, just look at your Virtual Proxy; it’s just another way of adding behavior to do something while some big expensive object is loading, and your Remote Proxy is a way of talking to remote objects so your clients don’t have to bother with that themselves. It’s all about behavior, just like I said.
I don’t think you get it, Decorator. I stand in for my Subjects; I don’t just add behavior. Clients use me as a surrogate of a Real Subject, because I can protect them from unwanted access, or keep their GUIs from hanging up while they’re waiting for big objects to load, or hide the fact that their Subjects are running on remote machines. I’d say that’s a very different intent from yours!	
	Call it what you want. I implement the same interface as the objects I wrap; so do you.
Okay, let’s review that statement. You wrap an object. While sometimes we informally say a proxy wraps its Subject, that’s not really an accurate term.	
	Oh yeah? Why not?
Think about a remote proxy... what object am I wrapping? The object I’m representing and controlling access to lives on another machine! Let’s see you do that.	
	Okay, but we all know remote proxies are kinda weird. Got a second example? I doubt it.
Sure, okay, take a virtual proxy... think about the CD viewer example. When the client first uses	

me as a proxy the subject doesn't even exist! So what am I wrapping there?	
	Uh huh, and the next thing you'll be saying is that you actually get to create objects.
I never knew decorators were so dumb! Of course I sometimes create objects. How do you think a virtual proxy gets its subject?! Okay, you just pointed out a big difference between us: we both know decorators only add window dressing; they never get to instantiate anything.	
	Oh yeah? Instantiate this!
Hey, after this conversation I'm convinced you're just a dumb proxy!	
	Dumb proxy? I'd like to see you recursively wrap an object with 10 decorators and keep your head straight at the same time.
Very seldom will you ever see a proxy get into wrapping a subject multiple times; in fact, if you're wrapping something 10 times, you better go back reexamine your design.	
	Just like a proxy, acting all real when in fact you just stand in for the objects doing the real work. You know, I actually feel sorry for you.

Using the Java API's Proxy to create a protection proxy

Java's got its own proxy support right in the java.lang.reflect package. With this package, Java lets you create a proxy class *on the fly* that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this Java technology as a *dynamic proxy*.

We're going to use Java's dynamic proxy to create our next proxy implementation (a protection proxy), but before we do that, let's quickly look at a class diagram that shows how dynamic proxies are put together. Like most things in the real world, it differs slightly from the classic definition of the pattern:



Because Java creates the Proxy class *for you*, you need a way to tell the Proxy class what to do. You can't put that code into the Proxy class like we did before, because you're not implementing one directly. So, if you can't put this code in the Proxy class, where do you put it? In an InvocationHandler. The job of the InvocationHandler is to respond to any method calls on the proxy. Think of the InvocationHandler as the object the Proxy asks to do all the real work after it's received the method calls.

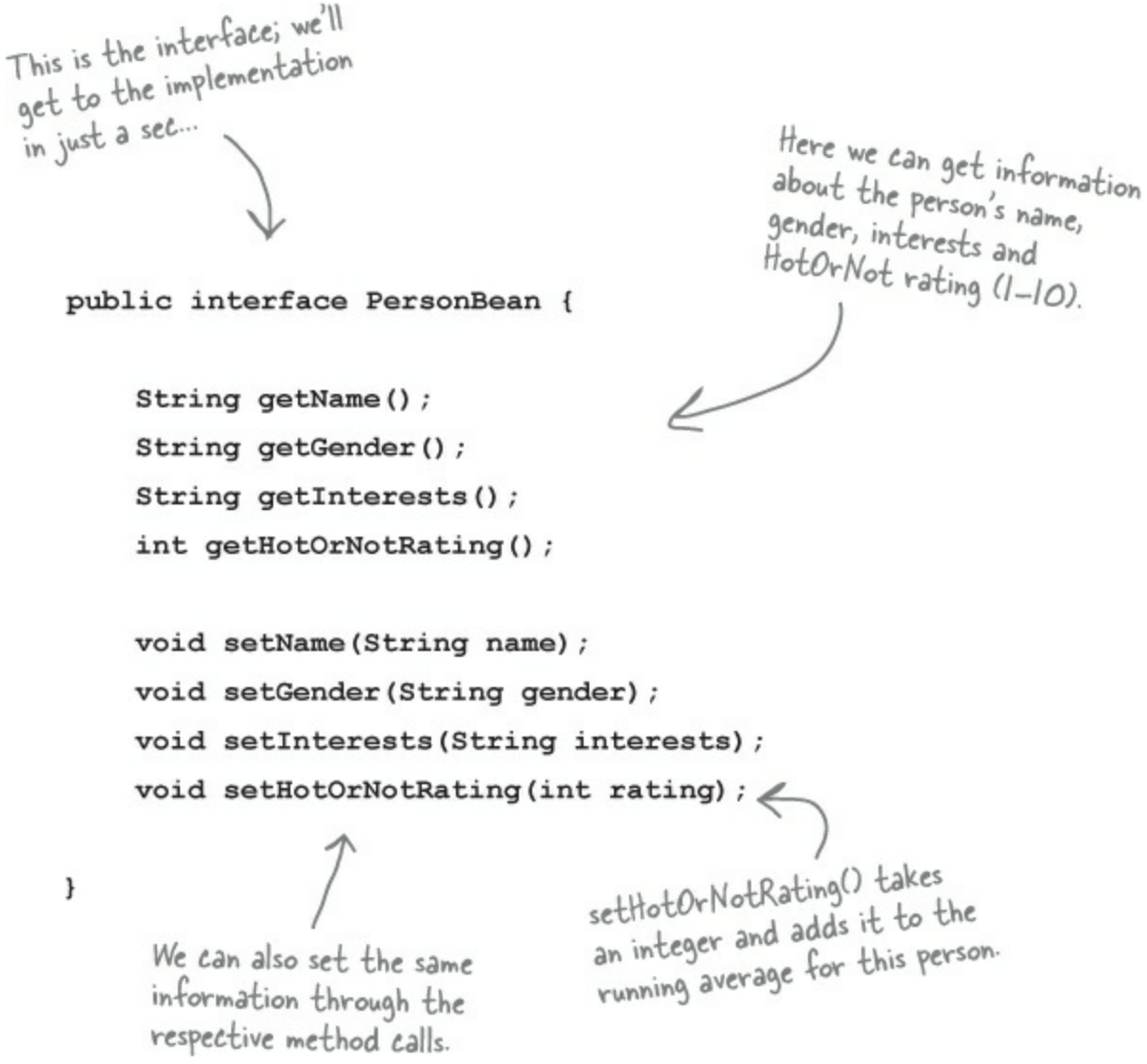
Okay, let's step through how to use the dynamic proxy...

Matchmaking in Objectville

Every town needs a matchmaking service, right? You've risen to the task and

implemented a dating service for Objectville. You've also tried to be innovative by including a "Hot or Not" feature in the service where participants can rate each other — you figure this keeps your customers engaged and looking through possible matches; it also makes things a lot more fun.

Your service revolves around a PersonBean that allows you to set and get information about a person:



Now let's check out the implementation...

The PersonBean implementation

The PersonBeanImpl implements the PersonBean interface.

```
public class PersonBeanImpl implements PersonBean {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setHotOrNotRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

The instance variables.

All the getter methods; they each return the appropriate instance variable...

...except for getHotOrNotRating(), which computes the average of the ratings by dividing the ratings by the ratingCount.

And here's all the setter methods, which set the corresponding instance variable.

Finally, the setHotOrNotRating() method increments the total ratingCount and adds the rating to the running total.

While we suspect other factors may be keeping Elroy from getting dates, he is right: you shouldn't be able to vote for yourself or to change another customer's data. The way our PersonBean is defined, any client can call any of the methods.

This is a perfect example of where we might be able to use a Protection Proxy. What's a Protection Proxy? It's a proxy that controls access to an object based on access rights. For instance, if we had an employee object, a Protection Proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like setSalary()), and a human resources employee to call any method on the object.

In our dating service we want to make sure that a customer can set his own

information while preventing others from altering it. We also want to allow just the opposite with the HotOrNot ratings: we want the other customers to be able to set the rating, but not that particular customer. We also have a number of getter methods in the PersonBean, and because none of these return private information, any customer should be able to call them.

Five-minute drama: protecting subjects

The Internet bubble seems a distant memory; those were the days when all you needed to do to find a better, higher-paying job was to walk across the street. Even agents for software developers were in vogue...