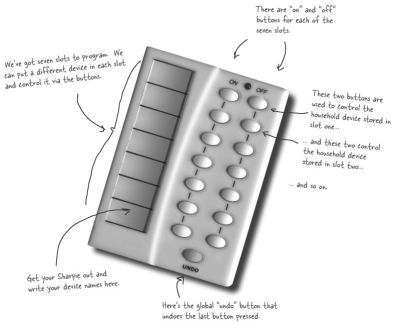
Chapter 6. The Command Pattern: Encapsulating Invocation

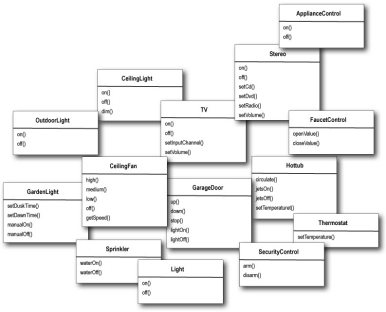
In this chapter, we take encapsulation to a whole new level: we're going to encapsulate method invocation. That's right, by encapsulating method invocation, we can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things, it just uses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

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



Taking a look at the vendor classes

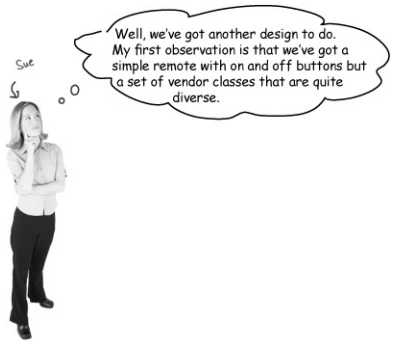
Check out the vendor classes on the CD-R. These should give you some idea of the interfaces of the objects we need to control from the remote.



It looks like we have quite a set of classes here, and not a lot of industry effort to come up with a set of common interfaces. Not only that, it sounds like we can expect more of these classes in the future. Designing a remote control API is going to be interesting. Let's get on to the design.

Cubicle Conversation

Your teammates are already discussing how to design the remote control API...



Mary: Yes, I thought we'd see a bunch of classes with on() and off() methods, but here we've got methods like dim(), setTemperature(), setVolume(), setDirection().

Sue: Not only that, it sounds like we can expect more vendor classes in the future with just as diverse methods.

Mary: I think it's important we view this as a separation of concerns: the remote should know how to interpret button presses and make requests, but it shouldn't know a lot about home automation or how to turn on a hot tub.

Sue: Sounds like good design. But if the remote is dumb and just knows how to make generic requests, how do we design the remote so that it can invoke an action that, say, turns on a light or opens a garage door?

Mary: I'm not sure, but we don't want the remote to have to know the specifics of the vendor classes.

Sue: What do you mean?

Mary: We don't want the remote to consist of a set of if statements, like "if slot1 == Light, then light.on(), else if slot1 = Hottub then hottub.jetsOn()". We know that is a bad design.

Sue: I agree. Whenever a new vendor class comes out, we'd have to go in and modify the code, potentially creating bugs and more work for ourselves!

Mary: Yeah? Tell us more.

Joe: The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action. So, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

Sue: How is that possible? How can we decouple them? After all, when I press a button, the remote has to turn on a light.

Joe: You can do that by introducing "command objects" into your design. A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object). So, if we store a command object for each button, when the button is pressed we ask the command object to do some work. The remote doesn't have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done. So, you see, the remote is decoupled from the light object!

Sue: This certainly sounds like it's going in the right direction.

Mary: Still, I'm having a hard time wrapping my head around the pattern.

Joe: Given that the objects are so decoupled, it's a little difficult to picture how the pattern actually works.

Mary: Let me see if I at least have the right idea: using this pattern we, could create an API in which these command objects can be loaded into button slots, allowing the remote code to stay very simple. And, the command objects encapsulate how to do a home automation task along with the object that needs to do it.

Joe: Yes, I think so. I also think this pattern can help you with that Undo button, but I haven't studied that part yet.

Mary: This sounds really encouraging, but I think I have a bit of work to do to really "get" the pattern.

Sue: Me too.

Meanwhile, back at the Diner..., or, A brief introduction to the Command Pattern

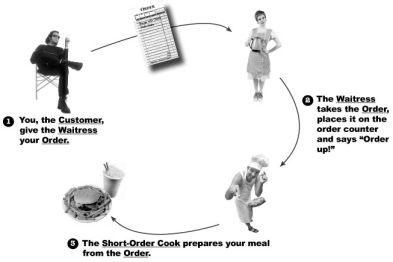
As Joe said, it is a little hard to understand the Command Pattern by just hearing its description. But don't fear, we have some friends ready to help: remember our friendly diner from Chapter 1? It's been a while since we visited Alice, Flo, and the short-order cook, but we've got good reason for returning (well, beyond the food and great conversation): the diner is going to help us

understand the Command Pattern.

So, let's take a short detour back to the diner and study the interactions between the customers, the waitress, the orders and the short-order cook. Through these interactions, you're going to understand the objects involved in the Command Pattern and also get a feel for how the decoupling works. After that, we're going to knock out that remote control API.

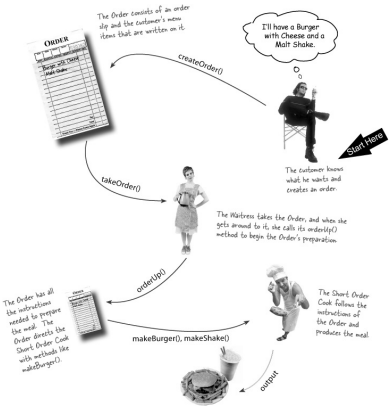
Checking in at the Objectville Diner...

Okay, we all know how the Diner operates:



Let's study the interaction in a little more detail...

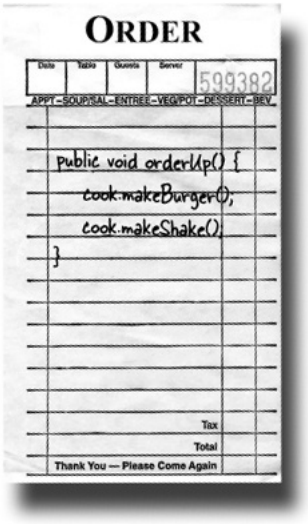
...and given this Diner is in Objectville, let's think about the object and method calls involved, too!



The Objectville Diner roles and responsibilities

An Order Slip encapsulates a request to prepare a meal.

- **Think of the Order Slip as an object, an object that acts as a request to prepare a meal.** Like any object, it can be passed around – from the Waitress to the order counter, or to the next Waitress taking over her shift. It has an interface that consists of only one method, orderUp(), that encapsulates the actions needed to prepare the meal. It also has a reference to the object that needs to prepare it (in our case, the Cook). It's encapsulated in that the Waitress doesn't have to know what's in the order or even who prepares the meal; she only needs to pass the slip through the order window and call "Order up!"



NOTE

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

The Waitress's job is to take Order Slips and invoke the orderUp() method on them.

- **The Waitress has it easy: take an order from the customer, continue helping customers until she makes it back to the order counter, then invoke the orderUp() method to have the meal prepared.** As we've already discussed, in Objectville, the Waitress really isn't worried about what's on the order or who is going to prepare it; she just knows order slips have an orderUp() method she can call to get the job done.

  Now, throughout the day, the Waitress's takeOrder() method gets parameterized with different order slips from different customers, but that doesn't phase her; she knows all Order slips support the orderUp() method and she can call orderUp() any time she needs a meal prepared.

The Short Order Cook has the knowledge required to prepare the meal.

- **The Short Order Cook is the object that really knows how to prepare meals.** Once the Waitress has invoked the orderUp() method; the Short Order Cook takes over and implements all the methods that are needed to create meals. Notice the Waitress and the Cook are totally decoupled: the Waitress has Order Slips that encapsulate the details of the meal; she just calls a method on each order to get it prepared. Likewise, the Cook gets his instructions from the Order Slip; he never needs to directly communicate with the Waitress.

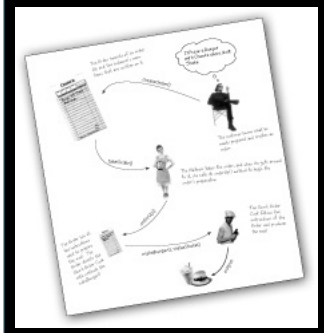
Patience, we're getting there...

Think of the Diner as a model for an OO design pattern that allows us to separate an object making a request from the objects that receive and execute those requests. For instance, in our remote control API, we need to separate the code that gets invoked when we press a button from the objects of the vendor-specific classes that carry out those requests. What if each slot of the remote held an object like the Diner's order slip object? Then, when a button is pressed, we could just call the equivalent of the "orderUp()" method on this object and have the lights turn on without the remote knowing the details of how to make those things happen or what objects are making them happen.

Now, let's switch gears a bit and map all this Diner talk to the Command Pattern...

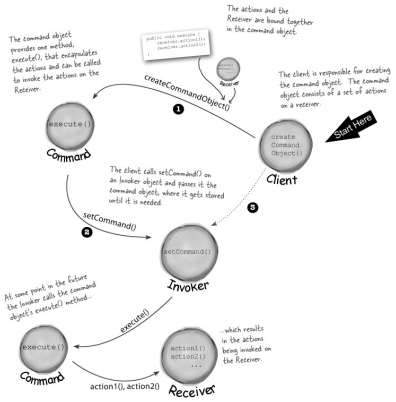
BRAIN POWER

Before we move on, spend some time studying the diagram two pages back along with Diner roles and responsibilities until you think you've got a handle on the Objectville Diner objects and relationships. Once you've done that, get ready to nail the Command Pattern!



From the Diner to the Command Pattern

Okay, we've spent enough time in the Objectville Diner that we know all the personalities and their responsibilities quite well. Now we're going rework the Diner diagram to reflect the Command Pattern. You'll see that all the players are the same; only the names have changed.



LOADING THE INVOKER

1. The client creates a command object.
2. The client does a setCommand() to store the command object in the invoker.
3. **Later**... the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.

WHO DOES WHAT?

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

Waitress	Command
Short Order Cook	execute()
orderUp()	Client
Order	Invoker
Customer	Receiver
takeOrder()	setCommand()

Our first command object

Isn't it about time we build our first command object? Let's go ahead and write some code for the remote control. While we haven't figured out how to design the remote control API yet, building a few things from the bottom up may help us...

Implementing the Command interface

- First things first: all command objects implement the same interface, which consists of one method. In the Diner we called this method orderUp(); however, we typically just use the name execute().

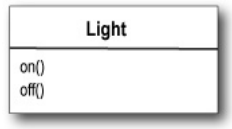
  Here's the Command interface:

```
public interface Command {
    public void execute();
}
```

Simple. All we need is one method called execute().

Implementing a Command to turn a light on

- Now, let's say you want to implement a command for turning a light on. Referring to our set of vendor classes, the Light class has two methods: on() and off(). Here's how you can implement this as a command:




```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

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

This is a command, so we need to implement the Command interface.

The constructor is passed the specific light that this command is going to control – say the living room light – and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

Now that you've got a LightOnCommand class, let's see if we can put it to use...

Using the command object

Okay, let's make things simple: say we've got a remote control with only one button and corresponding slot to hold a device to control:

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control

Here's just a bit of code to test out the simple remote control. Let's take a look and we'll point out how the pieces match the Command Pattern diagram:

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

This is our Client in Command Pattern-speak.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

Now we create a Light object; this will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

Here, pass the command to the Invoker.

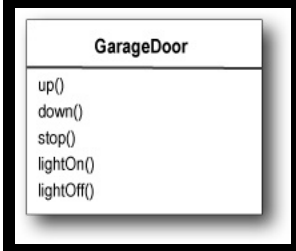
And then we simulate the button being pressed.

```
% java RemoteControlTest
Light is On
%
```

Here's the output of running this test code!

SHARPEN YOUR PENCIL

Okay, it's time for you to implement the GarageDoorOpenCommand class. First, supply the code for the class below. You'll need the GarageDoor class diagram.




```
public class GarageDoorOpenCommand
        implements Command {

}
```

Your code here

Now that you've got your class, what is the output of the following code? (Hint: the GarageDoor up() method prints out "Garage Door is Open" when it is complete.)

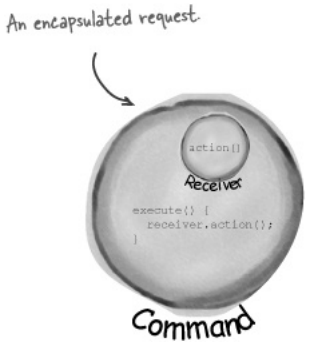
```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);


        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

Your output here

```
%java RemoteControlTest
```

The Command Pattern defined

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



Let's start with its official definition:

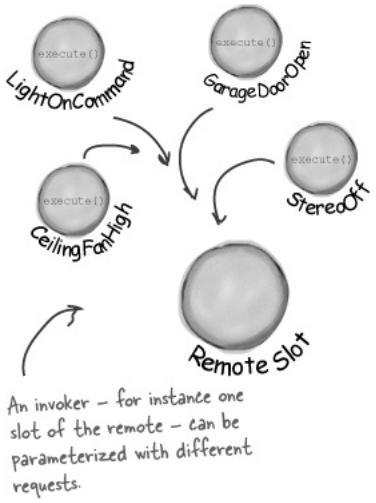
NOTE

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

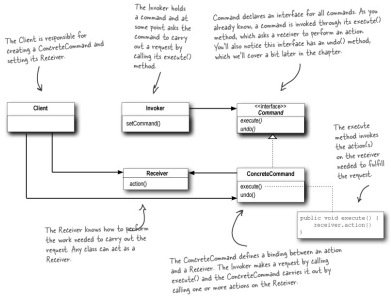
Let's step through this. We know that a command object *encapsulates a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute(). When called, execute() causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced.

We've also seen a couple examples of *parameterizing an object* with a command. Back at the diner, the Waitress was parameterized with multiple orders throughout the day. In the simple remote control, we first loaded the button slot with a "light on" command and then later replaced it with a "garage door open" command. Like the Waitress, your remote slot didn't care what command object it had, as long as it implemented the Command interface.

What we haven't encountered yet is using commands to implement *queues and logs and support undo operations*. Don't worry, those are pretty straightforward extensions of the basic Command Pattern and we will get to them soon. We can also easily support what's known as the Meta Command Pattern once we have the basics in place. The Meta Command Pattern allows you to create macros of commands so that you can execute multiple commands at once.



The Command Pattern defined: the class diagram



BRAIN POWER

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

Mary: Me too. So where do we begin?

Sue: Like we did in the SimpleRemote, we need to provide a way to assign commands to slots. In our case we have seven slots, each with an "on" and "off" button. So we might assign commands to the remote something like this:

```
onCommands[0] = onCommand;
offCommands[0] = offCommand;
```

and so on for each of the seven command slots.

Mary: That makes sense, except for the Light objects. How does the remote know the living room from the kitchen light?

Sue: Ah, that's just it, it doesn't! The remote doesn't know anything but how to call execute() on the corresponding command object when a button is pressed.

Mary: Yeah, I sorta got that, but in the implementation, how do we make sure the right objects are turning on and off the right devices?

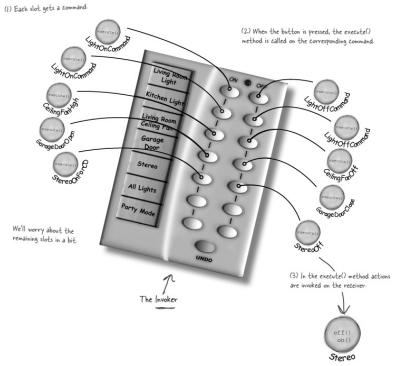
Sue: When we create the commands to be loaded into the remote, we create one LightCommand that is bound to the living room light object and another that is bound to the kitchen light object. Remember, the receiver of the request gets bound to the command it's encapsulated in. So, by the time the button is pressed, no one cares which light is which, the right thing just happens when the execute() method is called.

Mary: I think I've got it. Let's implement the remote and I think this will get clearer!

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

Assigning Commands to slots

So we have a plan: We're going to assign each slot to a command in the remote control. This makes the remote control our *invoker*. When a button is pressed the execute() method is going to be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans, stereos).



Implementing the Remote Control

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

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

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

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

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

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


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

This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the on and off arrays for later use.

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

We've overwritten toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

Implementing the Commands

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

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

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

The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Let's try something a little more challenging; how about writing on and off commands for the Stereo? Okay, off is easy, we just bind the Stereo to the off() method in the StereoOffCommand. On is a little more complicated; let's say we want to write a StereoOnWithCDCommand...

Stereo
on() off() setCd() setDvd() setRadio() setVolume()

```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

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

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

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

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

Not too bad. Take a look at the rest of the vendor classes; by now, you can definitely knock out the rest of the Command classes we need for those.

Putting the Remote Control through its paces

Our job with the remote is pretty much done; all we need to do is run some tests and get some documentation together to describe the API. Home Automation or Bust, Inc. sure is going to be impressed, don't ya think? We've managed to come up with a design that is going to allow them to produce a remote that is easy to maintain and they're going to have no trouble convincing the vendors to write some simple command classes in the future since they are so easy to write.

Let's get to testing this code!

```
public class RemoteLoader {

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

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

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

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

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

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

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        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);
    }
}
```

Create all the devices in their proper locations.

Create all the Light Command objects.

Create the On and Off for the ceiling fan.

Create the Up and Down commands for the garage.

Create the stereo On and Off commands.

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

Here's where we use our toString() method to print each remote slot and the command that it is assigned to.

All right, we are ready to roll! Now, we step through each slot and push its On and Off button.

Now, let's check out the execution of our remote control test...

```
File Edit Window Help CommandsGetThingsDone

% java RemoteLoader
------ Remote Control -------
[slot 0] headfirst.command.remote.LightOnCommand       headfirst.command.remote.LightOffCommand
[slot 1] headfirst.command.remote.LightOnCommand       headfirst.command.remote.LightOffCommand
[slot 2] headfirst.command.remote.CeilingFanOnCommand  headfirst.command.remote.CeilingFanOffCommand
[slot 3] headfirst.command.remote.StereoOnWithCDCommand headfirst.command.remote.StereoOffCommand
[slot 4] headfirst.command.remote.NoCommand            headfirst.command.remote.NoCommand
[slot 5] headfirst.command.remote.NoCommand            headfirst.command.remote.NoCommand
[slot 6] headfirst.command.remote.NoCommand            headfirst.command.remote.NoCommand

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room stereo volume set to 11
Living Room stereo is off
%
```

On slots

Off Slots

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living Room light is on"

Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the onButtonWasPushed() method, we would need code like this:

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

So, how do we get around that? Implement a command that does nothing!

```
public class NoCommand implements Command {
    public void execute() { }
}
```

Then, in our RemoteControl constructor, we assign every slot a NoCommand object by default and we know we'll always have some command to call in each slot.

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

So in the output of our test run, you are seeing slots that haven't been assigned to a command, other than the default NoCommand object which we assigned

when we created the RemoteControl.

PATTERN HONORABLE MENTION

The NoCommand object is an example of a *null object*. A null object is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling `null` from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a NoCommand object that acts as a surrogate and does nothing when its execute method is called.

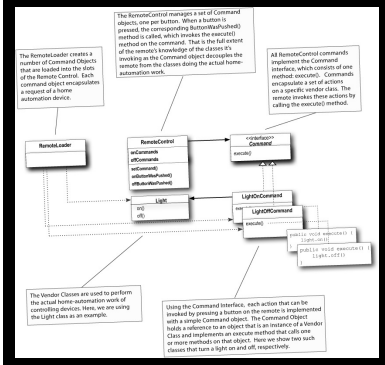
You'll find uses for Null Objects in conjunction with many Design Patterns and sometimes you'll even see Null Object listed as a Design Pattern.

Time to write that documentation...

Remote Control API Design for Home Automation or Bust, Inc.,

We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the Command Pattern to logically decouple the RemoteControl class from the Vendor Classes. We believe this will reduce the cost of producing the remote as well as drastically reduce your ongoing maintenance costs.

The following class diagram provides an overview of our design:



Whoops! We almost forgot... luckily, once we have our basic Command classes, undo is easy to add. Let's step through adding undo to our commands and to the remote control...

What are we doing?

Okay, we need to add functionality to support the undo button on the remote. It works like this: say the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button then the last action will be reversed – in this case the light will turn off. Before we get into more complex examples, let's get the light working with the undo button:

1. When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses. So, before we can add undo to our commands, we need to add an undo() method to the Command interface:

```
public interface Command {
    public void execute();
    public void undo();
}
```

Here's the new undo() method.

That was simple enough.

Now, let's dive into the Light command and implement the undo() method.

2. Let's start with the LightOnCommand: if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() method.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

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

    public void undo() {
        light.off();
    }
}
```

execute() turns the light on, so undo() simply turns the light back off.

Piece of cake! Now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

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

    public void undo() {
        light.on();
    }
}
```

And here, undo() turns the light back on!

Could this be any easier? Okay, we aren't done yet; we need to work a little support into the Remote Control to handle tracking the last button pressed and the undo button press.

3. To add support for the undo button we only have to make a few small changes to the Remote Control class. Here's how we're going to do it: we'll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

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

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

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

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

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

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // toString code here...
    }
}
```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

When a button is pressed, we take the command and first execute it, then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

Time to QA that Undo button!

Okay, let's rework the test harness a bit to test the undo button:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

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

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

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

Add the light Commands to the remote in slot 0.

Turn the light on, then off and then undo.

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

And here's the test results...

Using state to implement Undo

Okay, implementing undo on the Light was instructive but a little too easy. Typically, we need to manage a bit of state to implement undo. Let's try something a little more interesting, like the CeilingFan from the vendor classes. The ceiling fan allows a number of speeds to be set along with an off method.

CeilingFan
high() medium() low() off() getSpeed()

Here's the source code for the CeilingFan:

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

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

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

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

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

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

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

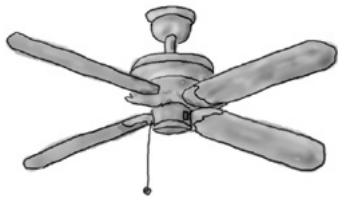
Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

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 ceiling fan commands

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



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

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

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

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

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

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

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

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 zero's on button with the medium setting for the fan and slot one with the high setting. Both corresponding off buttons will hold the ceiling fan off command.

Here's our test script:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
                new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
                new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

        remoteControl.onButtonWasPushed(0);
        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 zero and high in slot one. We also load up the off commands

First, turn the fan on medium

Then turn it off

Undo! It should go back to medium

Turn it on to high this time

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

Testing the ceiling fan...

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

```
File Edit Window Help UndoThis!

% java RemoteLoader

Living Room ceiling fan is on medium
Living Room ceiling fan is off

------ Remote Control -------
[slot 0] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 1] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanOffCommand

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

------ Remote Control -------
[slot 0] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 1] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanHighCommand

Living Room ceiling fan is on medium
%
```

Here are the commands in the remote control.

...and undo has the last command executed, the CeilingFanOffCommand

Undo the last command, and it goes back to medium

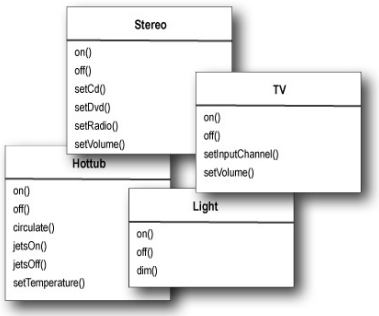
Now, turn it on high

Now, high is the last command executed.

One more undo, and the ceiling fan goes back to medium speed.

Every remote needs a Party Mode!

What's the point of having a remote if you can't push one button and have the lights dimmed, the stereo and TV turned on and set to a DVD and the hot tub fired up?



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:

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

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

Now create all the On commands to control them.

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

SHARPEN YOUR PENCIL

We will also need commands for the off buttons, write the code to create those here:

2. Next we create two arrays, one for the On commands and one for the Off commands, and load them with the corresponding commands:

Create an array for On and an array for Off commands...

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

...and create two corresponding macros to hold them.

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

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

Assign the macro command to a button as we would any command.

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

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 YouCanNeedABanks
% java RemoteLoader
------ Remote Control -------
[slot 0] headfirst.command.party.MacroCommand    headfirst.command.party.MacroCommand
[slot 1] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 2] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 3] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 4] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 5] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 6] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[undo] headfirst.command.party.NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees

%
```

Here are the two macro commands.

All the Commands in the macro are executed when we invoke the on macro.

And when we invoke the off macro. Looks like it works.

EXERCISE

The only thing our MacroCommand is missing its undo functionality. When the undo button is pressed after a macro command, all the commands that were invoked in the macro must undo their previous actions. Here's the code for MacroCommand; go ahead and implement the undo() method:

```
public class MacroCommand implements Command {
    Command[] commands;

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

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

    public void undo() {
        [                                    ]
    }
```

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.*

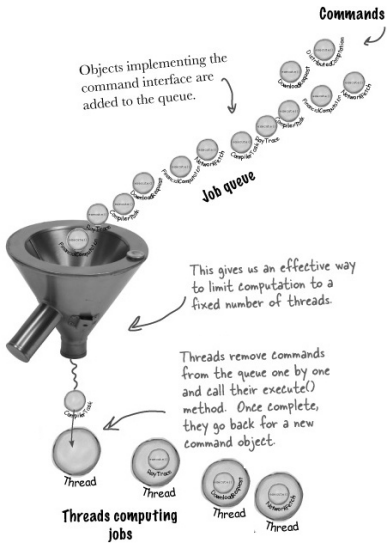
Q: **Q: Could I have just implemented Party Mode as a Command by creating a PartyCommand and putting the calls to execute the other Commands in the PartyCommand's execute() method?**

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.*

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 sit a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.



Note that the job queue classes are totally decoupled from the objects that are doing the computation. One minute a thread may be computing a financial computation, and the next it may be retrieving something from the network. The job queue objects don't care; they just retrieve commands and call execute(). Likewise, as long as you put objects into the queue that implement the Command Pattern, your execute() method will be invoked when a thread is available.

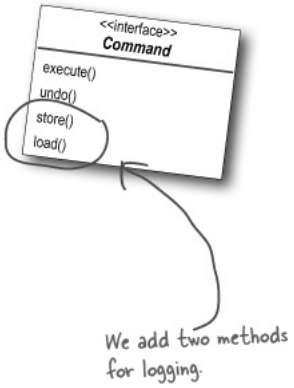
BRAIN POWER

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

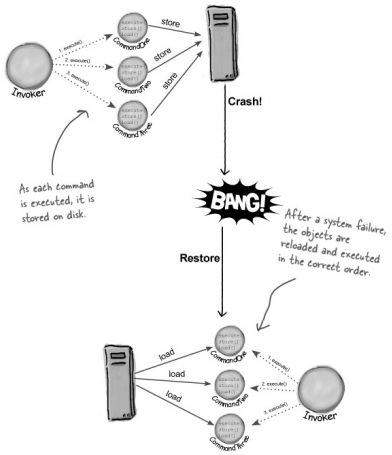
More uses of the Command Pattern: logging requests

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

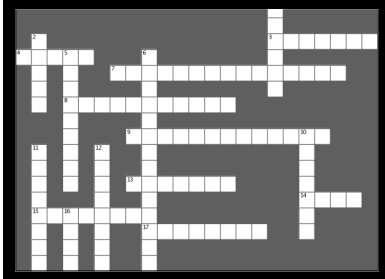
Your toolbox is starting to get heavy! In this chapter we've added a pattern that allows us to encapsulate methods into Command objects: store them, pass them around, and invoke them when you need them.

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.

Time to take a breather and let it all sink in.

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

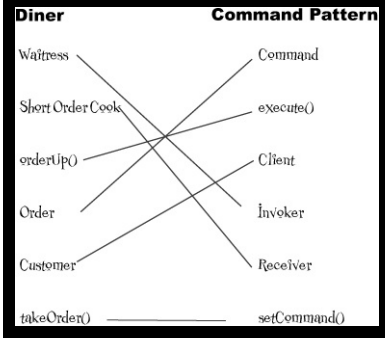


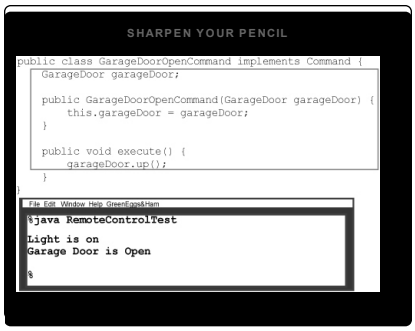
Across	Down
3. The Waitress was one 4. A command _____ a set of actions and a receiver 7. Dr. Seuss diner food 8. Our favorite city 9. Act as the receivers in the remote control 13. Object that knows the actions and the receiver 14. Another thing Command can do 15. Object that knows how to get things 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

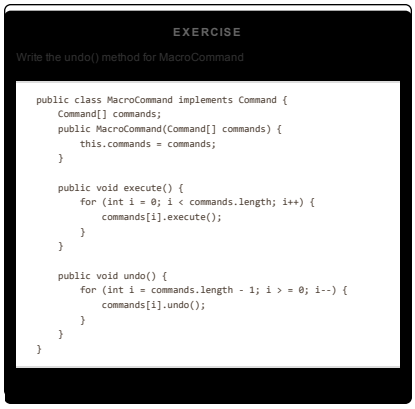
Exercise Solutions

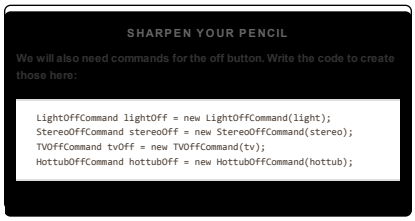
WHO DOES WHAT?

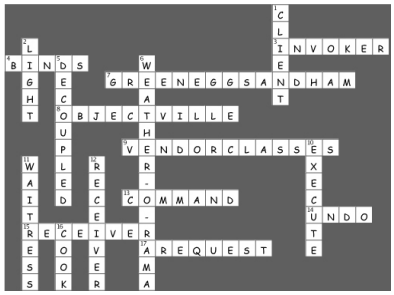
Match the diner objects and methods with the corresponding names from the Command Pattern











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

NEXT
7. The Adapter and Facade Patterns: Being Adaptive

Terms of Service / Privacy Policy