Command Objects

- We introduce **command objects** into the design
  - A command object encapsulates a request to do something (e.g., turn on a light) on a specific object (e.g., the living room lamp)
  - We can then just store a command object for each button such that when the button is pressed, the command is invoked
  - The button doesn't have to know *anything* about the command

An Analogy

An Analogy

An order slip encapsulates a request to prepare a meal. It's method `orderUp()` encapsulates the actions needed to prepare the meal; it also carries its own reference to the appropriate `Cook`

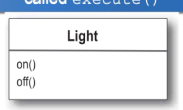
The waitress just creates order slips and invokes the `orderUp()` method.

The Cook knows how to prepare the meals; but he's completely decoupled from the waitress (they need never directly communicate)

A First Command Object

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

Simple. Just one method called `execute()`




```
public class LightOnCommand implements Command {
  Light light;
  public LightOnCommand (Light light) {
    this.light = light;
  }
  public void execute () {
    light.on();
  }
}
```

The constructor is passed the specific light that this command controls. When `execute()` gets called, this is the light that will be the receiver of the request

Using the Command Object

```
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 a command, which will control one device

We have a method to set the command the slot will control; could be called multiple times to change the behavior of the 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

A Simple Test of the Remote

```
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();
  }
}
```

The `RemoteControlTest` class is the Client

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

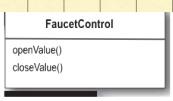
The `Light` object is the Receiver of the request

We create a command and pass it the Receiver

Then we pass the command to the Invoker.

Example

- Implement the `FaucetOffCommand` class
- Here's the new "test" code:



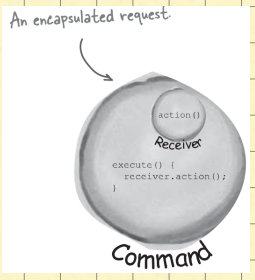

```
public class RemoteControlTest {
  public static void main(String[] args) {
    SimpleRemoteControl remote = new SimpleRemoteControl();
    Faucet faucet = new FaucetControl();
    FaucetOffCommand faucetOff = new FaucetOffCommand(faucet);
    remote.setCommand(faucetOff);
    remote.buttonWasPressed();
  }
}
```

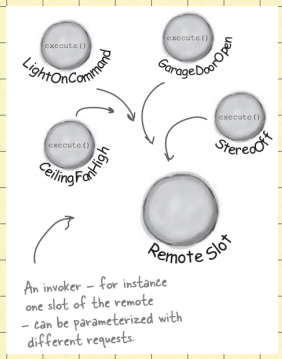
Solution

```
public class FaucetOffCommand implements Command {
  Faucet faucet;
  public FaucetOffCommand(Faucet faucet) {
    this.faucet = faucet;
  }
  public void execute () {
    faucet.closeValve();
  }
}
```

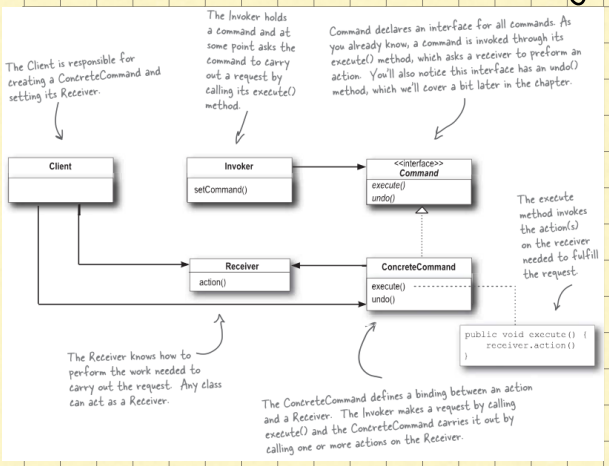
The Command Pattern

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.





The Command Pattern Class Diagram



Implement Remote Control

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

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

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

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

    ...;
}
```

Add Support for the Undo Button

- First, let's expand the `Command` interface:

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

- Now, every `Command` should be undoable
- So we add an implementation for `undo()` for every command that we implement
  - E.g., for `LightOnCommand`, `undo()` simply calls `light.off()`
  - Some `undo()` method implementations are more complicated; e.g., undoing a change in speed of a ceiling fan
- All that's left is to add support to the remote control to handle tracking which `undo()` method to call
  - Store the last command executed; if the undo button is pressed, we can just invoke the `undo()` method on that command