The Command Pattern
Example: Remote Control

• Given remote control with seven programmable slots.
• A different device can be put into each slot.
• There is an On and Off switch for each device slot.
• Global Undo button undoes the last button pressed.
• Also given a CD with different vendor classes that have already been written (for the different devices, TV, Light, Sprinkler, etc)
First Thoughts

• We know that there are seven programmable slots for different devices...so each device may possibly adhere to some common interface.
• We know that we need to turn each device “on” or “off”..so that needs to be commonly done for any device.
• Undo needs to work for any device as well.
What Varies? What stays the same?

• What Varies
  – The actual device assigned to a slot
  – The instruction for “On” on a specific device
  – The instruction for “Off” on a specific device

• What Stays the Same
  – Device with seven slots
  – Capability to assign a slot to a device
  – Capability to request that a device turn On or Off
  – Capability to undo the last action requested against the device
The Vendor Classes

• Vendor classes have been provided to us via a CD.
  – Ceiling Light
  – TV
  – Hottub
• We know that each device needs an “On” or “Off” state.
  – Since the capability to turn a device “On” or “Off” is something that “stays the same”.
  – However, each vendor class has their own unique way of doing “On” or “Off”.
One possible solution...

```java
if (slot1 == Light)
    light.on();

else if (slot1 == Hottub) {
    hottub.prepareJets();
    hottub.jetsOn();
}

Else if (slot1 == TV)
    tv.on();

//...

Also... what about undo???
```

**Problems:**

The Remote needs to be aware of all the details about turning a device on (or off).

If device On/Off mechanism changes, the Remote code will need to be changed.

If a new device is added, this code would need to be changed.

Is this Open for Extension??
Separation of Concerns

• The Vendor Class
  – One (or more) methods that define “On”
  – One (or more) methods that define “Off”

• The Command
  – Sends a message to a device (On or Off)
  – Handle the undo of a message
  – Possible future enhancements
    • Logging request
    • Queue request

• The Remote – handles one or more Commands. The Remote doesn’t know anything about the actual vendor class specifics.
The Command Pattern

• “Allows you to decouple the requestor of the action from the object that performs the action.”

• “A Command object encapsulates a request to do something.”
  – Note: A Command object handles a single request.

• The Command interface (in this example) just does one thing... executes a command.
public class LightOnCommand implements Command {

    Light light;

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

    public void execute() {
        light.on();
    }
}
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

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

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

An encapsulated Request

Command

Invoker

Command encapsulates a Receiver object, use delegation to the corresponding device,
Different commands can fit into a Remote Slot in the remote control.
RemoteControlTest – A client

```java
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();
```

Create two vendor classes.

Create two commands based on these vendor classes.

Set the command and press button
The Command Pattern

• GoF **Intent**: “Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.”

• **Participants:**
  – **Client** (RemoteControlTest) – creates command and associates command with receiver.
  – **Receiver** (TV, HotTub, ec) – knows how to perform the work.
  – **Concrete Command** (LightOnCommand) - implementation of Command interface
  – **Command Interface** – defines interface for all commands.
  – **Invoker** (Remote Control) – holds reference to a command and calls execute() method
- **Client** creates a **ConcreteCommand** and binds it with a **Receiver**.
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
The Sequence Diagram

```
new Command(aReceiver)
StoreCommand(aCommand)
Action()
Execute()
```
Class Diagram for Home automation

**RemoteLoader**
- Creates command objects, binds with devices

**RemoteControl**
- onCommands
- offCommands
- setCommand()
- onButtonPushed()
- offButtonPushed()

**Light**
- on()
- off()

**LightOnCommand**
- execute()
- undo()

**LightOffCommand**
- execute()
- undo()

**Command**
- execute()
- undo()

Invokes execute() method of the button command object

1. Invokes execute() method of the button command object
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 = 0; i < commands.length; i++) {
            commands[i].undo();
        }
    }
}
Adding Undo

Easy for some objects

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

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

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

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

Implementing the remote control with undo.

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

    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;
        }
        undoCommand = noCommand;
    }

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

    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 off() {
        speed = OFF;
        // code to turn fan off
    }
}
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();
        }
    }
}
History of Undo Operations

• If the Undo button is pressed multiple times, we want to undo each command that had been previously applied.
• Which object should we enhance to store a history of each command applied? Why?
  – Client (RemoteControlTest)
  – Receiver (TV, DVD, etc)
  – ConcreteCommand (LightOnCommand, LightOffCommand, etc)
  – Invoker (RemoteControl)
• We need to be able to add commands to a list and then later get the most recent one. What kind of object can we use?
History of Undo Operations

public class RemoteControl {

    Stack<Command> undoStack;  //this gets initialized in
    //constructor.

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

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

    public void undoButtonWasPushed() {
        Command c = undoStack.pop();
        c.undo();
    }
}
Simple Logging

• We want to enhance the Remote Control again to log every time a Command is executed (on or off).

• Which object should we enhance??
Simple Logging

Changes to RemoteControl...

```java
public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
    //Log here
}
```

```java
public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
    //Log here
}
```

Advantage: We can add logging in the Invoker. No change is needed in any of the Command or Receiver objects!
Complex Logging

Let’s say we had a spreadsheet application and we know it may crash often.

For failure recovery, we could periodically store a backup of the spreadsheet every 5 minutes...or we could periodically persist the list of commands executed on the spreadsheet since the last save point.

When a crash occurs, we load the last saved document and re-apply the persisted commands.
Complex Logging

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

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

public void SaveButtonPressed() {
    //Delete stored commands from disk.
}

public void RestoreCommands() {
    //load last saved state
    Commands[] storedCommands = GetCommandsFromDisk();
    //for each Command, call execute()
}
```

Once again, we can make these changes in one place (the Invoker)
Summary so far..

- **OO Basics**
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

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