# Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

# Gang of Four's Pattern Catalog

| Creational | Structural | Behavioral |
|---|---|---|
| Abstract Factory | Adapter | Chain of Responsibility |
| Builder | Bridge | Command |
| Factory Method | Composite | Interpreter |
| Prototype | Decorator | Iterator |
| Singleton | Facade | Mediator |
| | Flyweight | Memento |
| | Proxy | Observer |
| | | State |
| | | Strategy |
| | | Template Method |
| | | Visitor |

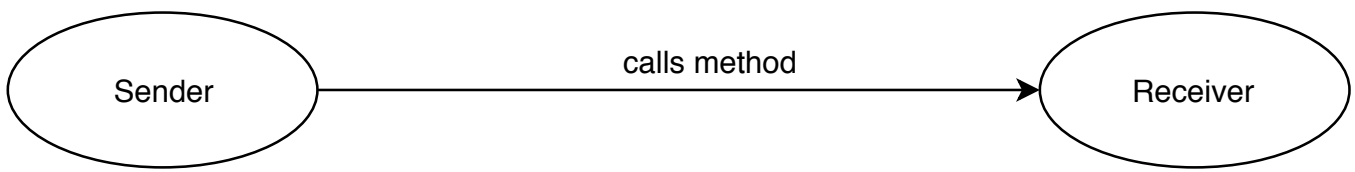
## Command Pattern: Intent

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

- Also known as
  - Action, Transaction

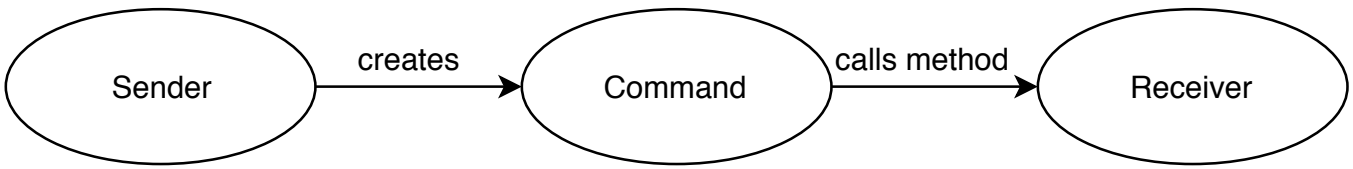
## Command Pattern

- The command pattern encapsulates a request as an object of its own.
- In general, when an object makes a request for a second object to do an action, the first object would call a method of the second object and the second object would complete the task.
- There is direct communication between the sender and receiver object.



## Command Pattern (contd.)

- The command pattern creates a command object in between the sender and receiver.
- This way, the sender does not have to know about the receiver and the methods to call.



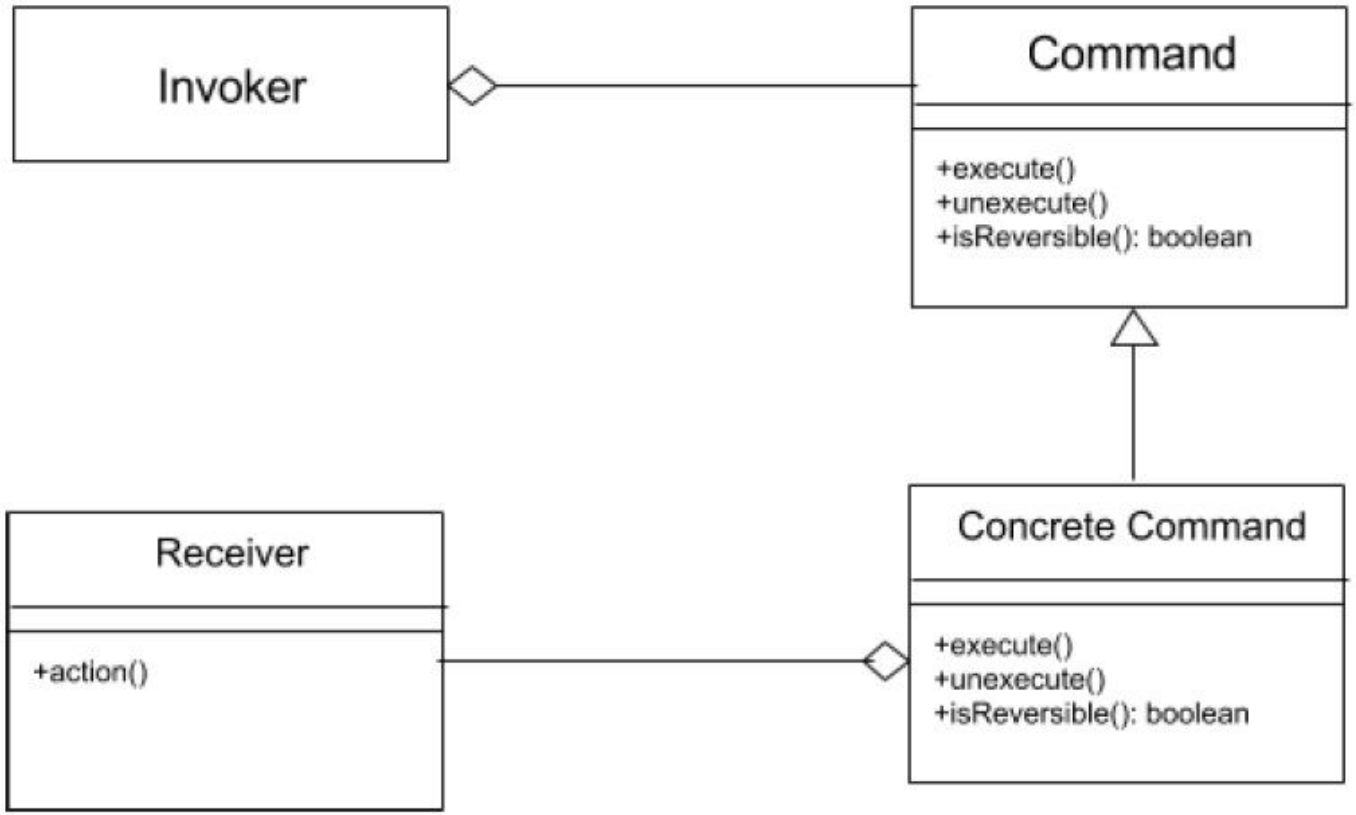
## Command Pattern (contd.)

- In a command pattern, a sender object can create a command object.
- However, an invoker is required to make the command object do what it's supposed to do, and get the specific receiver object to complete the task.
- An invoker is therefore an object that invokes the command objects to complete whatever task it is supposed to do.
- A command manager can also be used which basically keeps track of the commands, manipulates them, and invokes them.

## Command Pattern: Purpose

- One is to store and schedule different requests.
- Another purpose for the command pattern is to allow command to be undone or redone.
- The command pattern lets you do things to requests that you wouldn't be able to do if they were simple method calls from one object to the other.
- Commands can also be stored in a log list, so that if the software crashes unexpectedly, users can redo all the recent commands.

# Command Pattern: Structure



## Command Pattern: Structure (contd.)

- In this diagram, there is a command superclass, and all command are instances of subclasses of this command superclass.
- The superclass defines the common behaviours of your commands. Each command will have the methods execute(), unexecute(), and isReversible().
  - The execute() method will do the work the command is supposed to do.
  - The unexecute() method will do the work of undoing the command.
  - The isReversible() method will determine if the command is reversible, returning true if the command can be undone.

## Command Pattern (contd.)

- Some commands may not be able to be undone, such as a save command.
- The concrete command classes call on specific receiver classes to deal with the work of complete the command.

## Command Pattern: Example

- Let us examine how a command object should be written in Java code, with the example of "simple remote control".

## Command Pattern: Example (contd.)

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

## Command Pattern: Example (contd.)

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

# Command Pattern: Example (contd.)

```
public class Light {
    String location = "";

    public Light(String location) {
        this.location = location;
    }

    public void on() {
        System.out.println(location + " light is on");
    }

    public void off() {
        System.out.println(location + " light is off");
    }
}
```

# Command Pattern: Example (contd.)

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

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

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

# Command Pattern: Example (contd.)

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

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

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

# Command Pattern: Example (contd.)

```
public class GarageDoor {
    String location;
    public GarageDoor(String location) {
        this.location = location;
    }
    public void up() {
        this.lightOn();
        System.out.println(location + " garage Door is open");
    }
    public void down() {
        this.lightOff();
        System.out.println(location + " garage Door is close");
    }
    public void lightOn() {
        System.out.println(location + " garage light is on");
    }
    public void lightOff() {
        System.out.println(location + " garage light is off");
    }
}
```

# Command Pattern: Example (contd.)

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

    public GarageDoorUpCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }

    public void execute() {
        garageDoor.up();
    }
}
```

# Command Pattern: Example (contd.)

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

    public GarageDoorDownCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }

    public void execute() {
        garageDoor.down();
    }
}
```

# Command Pattern: Example (contd.)

```
public class Stereo {
    String location;

    public Stereo(String location) {
        this.location = location;
    }

    public void on() {
        System.out.println(location + " stereo is on");
    }

    public void off() {
        System.out.println(location + " stereo is off");
    }

    public void setCD() {
        System.out.println(location + " stereo is set for CD
            input");
    }
```

# Command Pattern: Example (contd.)

```
//continued from previous slide

    public void setDVD() {
        System.out.println(location + " stereo is set for DVD
            input");
    }

    public void setRadio() {
        System.out.println(location + " stereo is set for
            Radio");
    }

    public void setVolume(int volume){
       // code to set the volume
       System.out.println(location + " stereo volume set to " +
           volume);
    }
}
```

# Command Pattern: Example (contd.)

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

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

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(10);
    }
}
```

# Command Pattern: Example (contd.)

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

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

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

# Command Pattern: Example (contd.)

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

    public SimpleRemoteControl() {
        onCommands = new Command[4];
        offCommands = new Command[4];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 4; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }
```

# Command Pattern: Example (contd.)

```
//continued from previous slide

    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();
    }
}
```

# Command Pattern: Example (contd.)

```
public class RemoteControlTest {

    public static void main(String[] args) {
        SimpleRemoteControl simpleRemoteControl = new
            SimpleRemoteControl();

        Light livingRoomLight = new Light("Living Room");
        LightOnCommand livingRoomLightOn = new
            LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff = new
            LightOffCommand(livingRoomLight);

        Light kitchenLight = new Light("Kitchen");
        LightOnCommand kitchenLightOn = new
            LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff = new
            LightOffCommand(kitchenLight);
```

# Command Pattern: Example (contd.)

```
//continued from previous slide

        GarageDoor garageDoor = new GarageDoor("Garage");
        GarageDoorUpCommand garageDoorUp = new
            GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown = new
            GarageDoorDownCommand(garageDoor);

        Stereo stereo = new Stereo("Living Room");
        StereoOnWithCDCommand stereoOnWithCD = new
            StereoOnWithCDCommand(stereo);
        StereoOffCommand stereoOff = new
            StereoOffCommand(stereo);
```

# Command Pattern: Example (contd.)

```
//continued from previous slide

        simpleRemoteControl.setCommand(0, livingRoomLightOn,
            livingRoomLightOff);
        simpleRemoteControl.setCommand(1, kitchenLightOn,
            kitchenLightOff);
        simpleRemoteControl.setCommand(2, stereoOnWithCD,
            stereoOff);
        simpleRemoteControl.setCommand(3, garageDoorUp,
            garageDoorDown);

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

## Command Pattern: Benefits

- The command pattern allows commands to be manipulated as objects.
- Functionalities can be added to the command objects, such as putting them into queues, and adding undo/redo functions.

## Command Pattern: Benefits (contd.)

- Command patterns also decouple the objects of your software program, as classes do not need to know about other objects in the software system.

- The command object deals with the work by invoking receiver objects, and the original object does not need to know what other objects are involved in the request.

## Command Pattern: Benefits (contd.)

- The command pattern also allows logic to be pulled from user interfaces.
- User interface classes should only be dealing with issues like getting information to and from the user, and application logic should not be in user interface classes.
- The command pattern creates a layer where command objects go, so that every time a button is clicked on the interface, a command object is created. This is where application logic will sit instead.
- The command objects are independent of the user interface, so that adding changes like new buttons to the interface is easier and faster.

## Command Pattern: Benefits (contd.)

- Each and every service in a system can be an object of its own, allowing for more flexible functionality.
- This pattern can be a great asset to making versatile and easy-to-maintain software programs.