Command

com·mand

/kəˈmand/

noun

1. an instruction or signal that causes a computer to perform one of its basic function

Command Intent

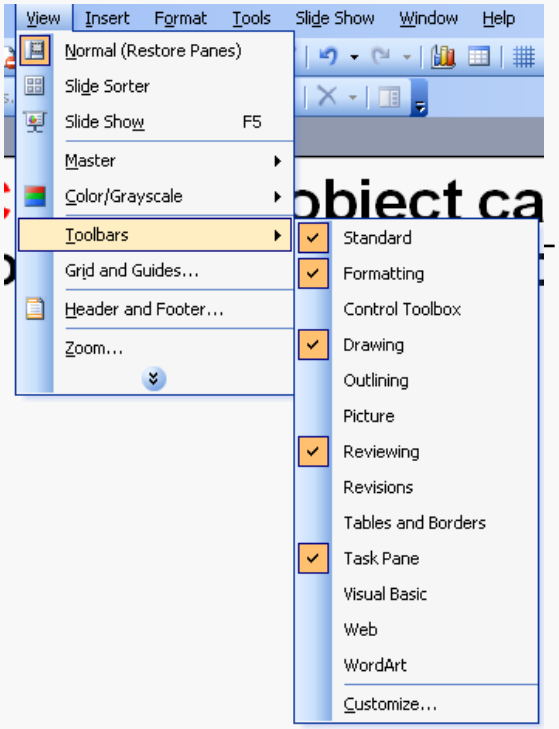
Encapsulate a request as an **object**, thereby letting you **parameterize** clients with different request, queue or log requests, and support undoable operations.

(Behavioral)

In a typical application, how many different ways are available to the user to invoke an operation?

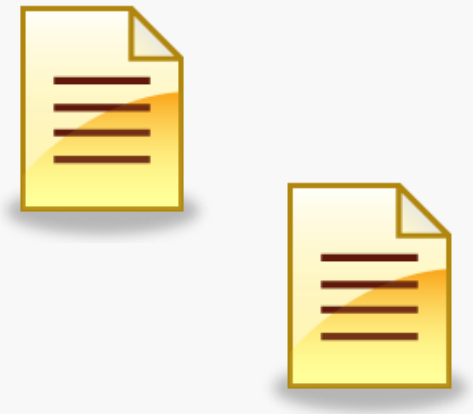
A Command object can decouple invocation from knowledge of execution of the operation.

Invokers



Request

Targets



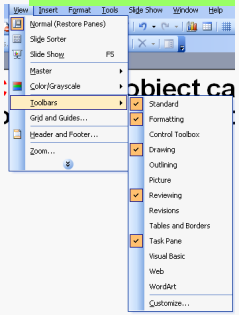
Specifics of what to do when button “clicked” are known to the application using the button (**receiver**)

A Command object can decouple invocation from knowledge of execution of the operation.

Commands know the targets and how to do the operation.

have Command objects

Invokers



Execute

Command Object

Execute

Command Object

Execute

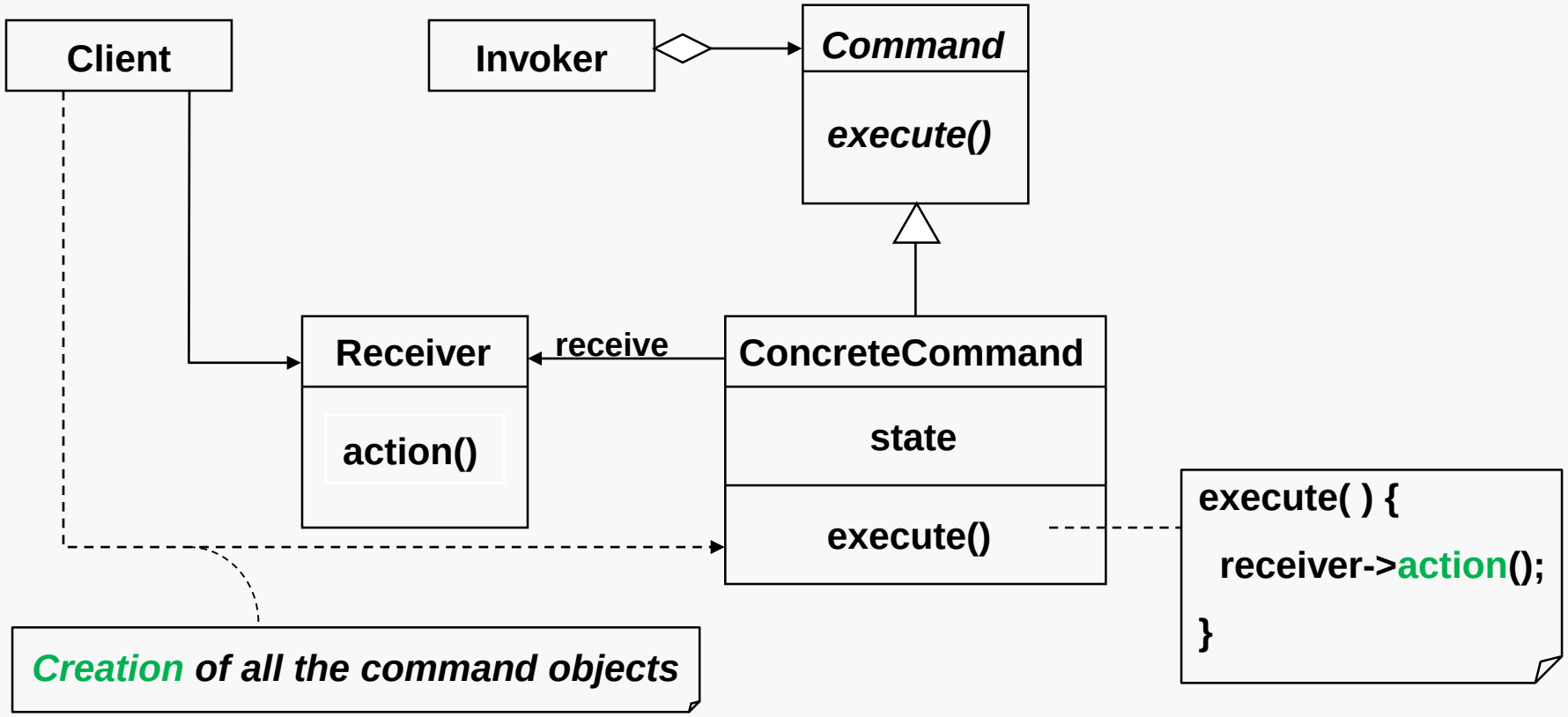
Command Object

Receivers

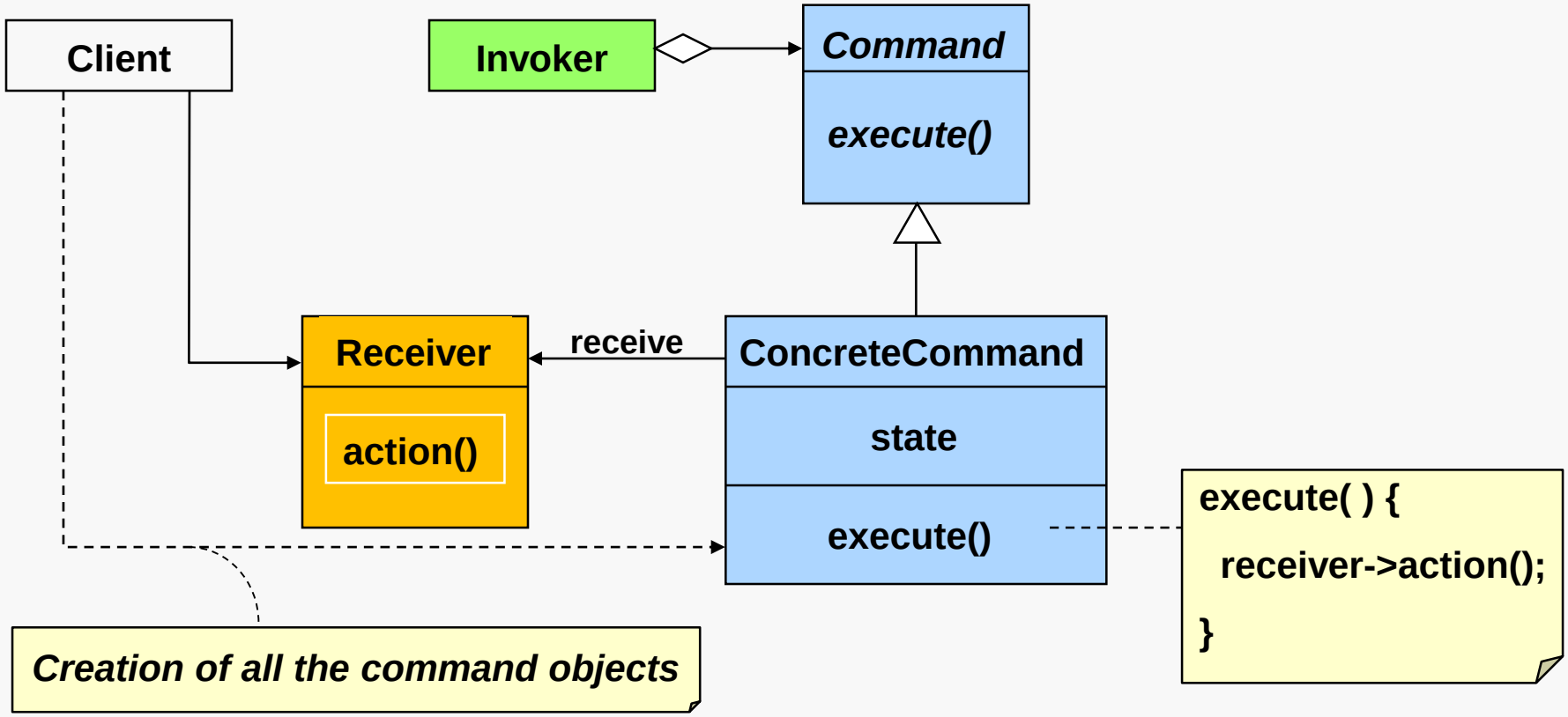
Participants

- Command
  - ***Interface for executing every operation***
- Concrete Command
  - ***Implements operation***
  - ***Binds receiver and action***
- Client
  - ***Creates Concrete Command***
  - ***Determines Receiver***
- Invoker
  - ***Requests command to execute operation***
- Receiver
  - ***Performs the operations needed***

Each command knows how to execute the operation and where to execute it.



Each command knows how to execute the operation and where to execute it.



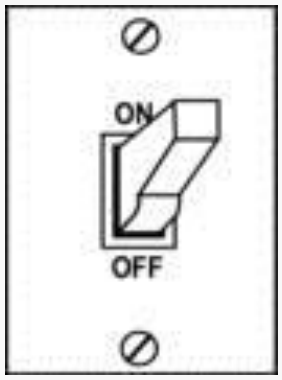
Encapsulating how to perform an operation allows separation of concerns in space and time.

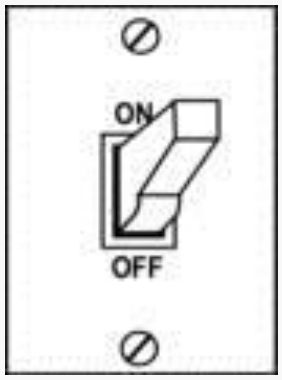
- Invocation (view) is decoupled from execution (control/model).
- Execution can happen at a different time than invocation.
  - ***How can this support undo/redo?***
- You can create sequences of commands for later execution.
  - ***How can this support macro commands?***
  - ***What other design pattern would you use?***

There are several design choices that you have.

- How smart is the command object?
  - ***Only binds command to receiver and action***
  - ***Performs the operation itself***
- When is a command instantiated?
  - ***Prior to invocation***
  - ***Upon invocation***
- When is the receiver bound to the command?
  - ***When command is instantiated***
  - ***When command is invoked***

Command Pattern (Example)

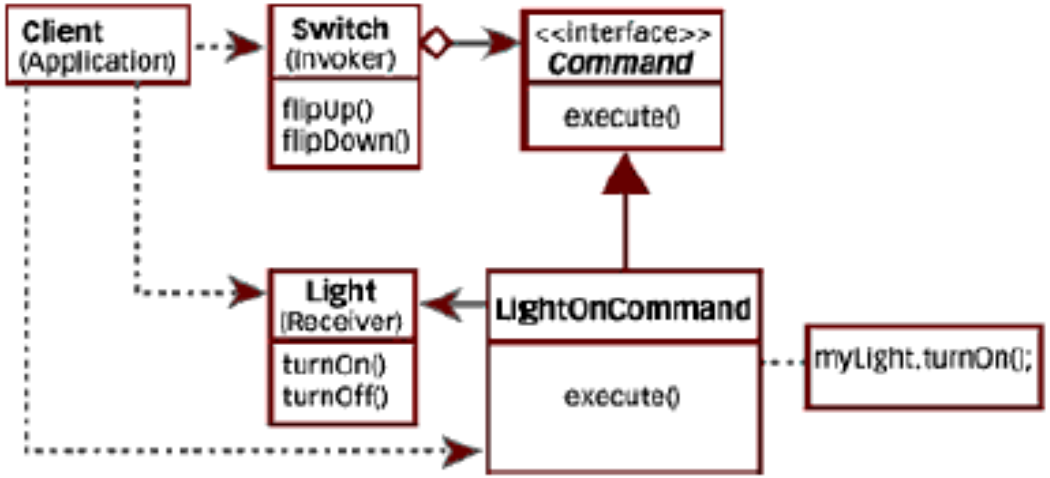




Example – Wiring Electrical Switches

A switch has flipUp() and flipDown() operations in its interface. Switch is called the *invoker* because it invokes the execute operation in the *command interface.*

The *concrete command*, LightOnCommand, implements the execute operation of the command interface. It has the knowledge to call the appropriate *receiver* object's operation

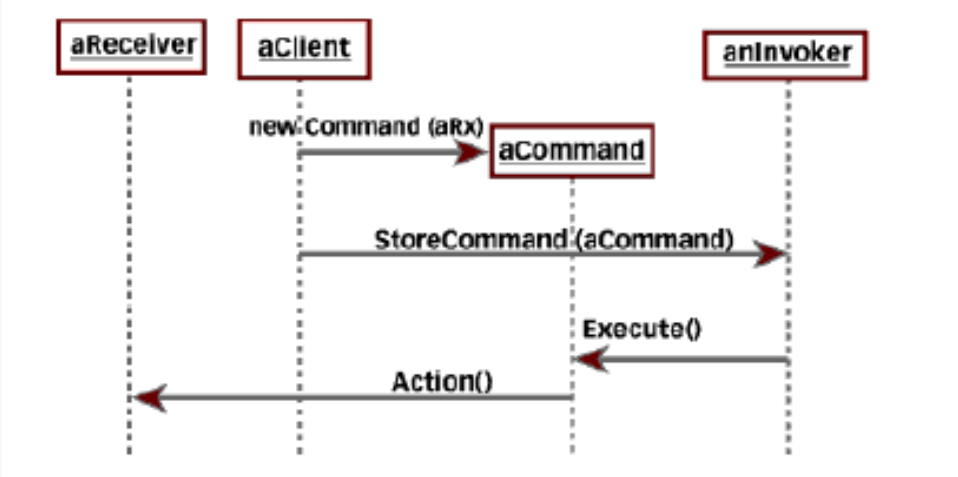


The client creates a command object.

The client does a StoreCommand() to store the command in the Invoker

Later… the invoker will execute the command (i.e. when the switch is flipped in this example)

All the invoker needs to know is that some stored action is executed



Java Implementation – Receiver Classes

```
class Fan {
   public void startRotate() {
     System.out.println("Fan is rotating");
   }
   public void stopRotate() {
     System.out.println("Fan is not rotating");
   }
}

class Light {
   public void turnOn( ) {
     System.out.println("Light is on ");
   }
   public void turnOff( ) {
     System.out.println("Light is off");
   }
}
```

Command Interface & Concrete Commands

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

class LightOnCommand implements Command {
   private Light myLight;
   public LightOnCommand (Light L) {myLight = L;}
   public void execute( ) { myLight.turnOn( ); }
}

class LightOffCommand implements Command {
   private Light myLight;
   public LightOffCommand (Light L) {myLight = L;}
   public void execute( ) { myLight.turnOff( ); }
}

class FanOnCommand implements Command {
   private Fan myFan;
   public FanOnCommand ( Fan F) { myFan = F; }
   public void execute( ) { myFan.startRotate( ); }
}

class FanOffCommand implements Command {
   private Fan myFan;
   public FanOffCommand ( Fan F) { myFan = F; }
   public void execute( ) { myFan.stopRotate( ); }
}
```

Invoker Class

```
class Switch {
   // concrete Commands registered with this invoker during
  // instantiation
  private Command UpCommand, DownCommand;

  public Switch( Command Up, Command Down) {
   // wired at instantiation
     UpCommand = Up;
     DownCommand = Down;
     }

  // invoker calls back concrete Command, which executes
  // the Command on the receiver

  void flipUp( ) {
     UpCommand.execute ( ) ;
     }
  void flipDown( ) {
     DownCommand.execute ( );
     }

  }
```

Simple Client does the wiring and testing

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

   // Create receivers
   Light testLight = new Light( );
   Fan testFan = new Fan( );

   // Create commands
   LightOnCommand testLOC = new LightOnCommand(testLight);
   LightOffCommand testLFC = new LightOffCommand(testLight);

   FanOnCommand foc = new FanOnCommand(testFan);
   FanOffCommand ffc = new FanOffCommand(testFan);

   // Create invokers and store commands
   Switch testSwitch1 = new Switch( testLOC,testLFC);
   Switch testSwitch2 = new Switch( foc,ffc);

   // Have invokers execute commands
   testSwitch1.flipUp( );       // light on
   testSwitch1.flipDown( );     // light off

   testSwitch2.flipUp( );       // fan on
   testSwitch2.flipDown( );     // fan off
   }
}
```

Only the concrete command objects knows of the receiver objects

Wiring at instantiation. The invoker only knows about the command objects and running their execute() method