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

Command Pattern is Behavioral

- Structural
  - Strategy
- Behavioral
  - Observer
  - Decorator
  - Command
- Creational
  - Factory method
  - Abstract factory
  - Singleton

Problem

Your program has many different actions it can perform. Implementing these actions would lead to huge if-elsif or switch blocks.

Command Pattern

- Move the code for each individual action into its own class
- Each of these classes implement the same interface, allowing the code that uses them to interact solely with the interface and not know or care about the individual classes
- This increases cohesion because each class is responsible for one discrete set of logic
- This decreases coupling because the code calling the command only deals with one type – the interface

Uses

- Thread pools
- GUI buttons and menu items (Java Swing – Action is a command object)
- Progress bars: each command object has an estimatedDuration method that can be called
- Multi-level undo: user actions are implemented as command objects and program keeps a stack of most recently executed commands – most recent object is popped and its undo method is executed

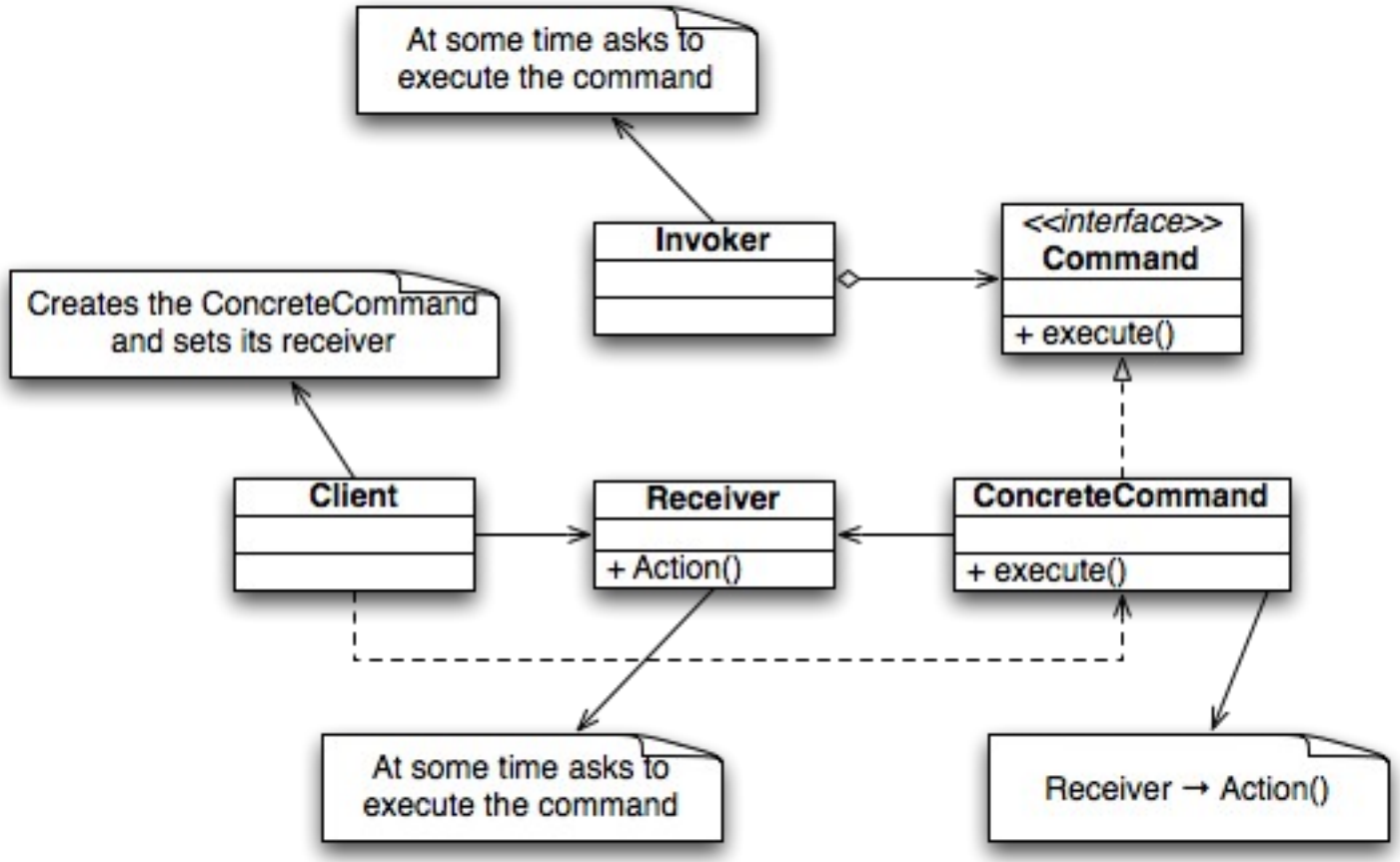
General Use of Command Pattern

Use the Command Pattern when you need to decouple an object making requests from the objects that know how to perform the requests

Command Pattern Main Concepts

- It decouples an object, making a request from the one that knows how to perform it
- Command object is at the center of this decoupling and encapsulates a receiver with an action
- 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 at runtime
- Macro Commands: an extension of Command that allow multiple commands to be invoked

Class Diagram (from Wikipedia)



Command Pattern

- An object is used to represent and encapsulate all the information needed to call a method at a later time
- This information includes the method name, the object that owns the method, and the values for the method parameters
- Three fundamental Command pattern terms:
  - Client: instantiates command object and provides information to call the method at a later time
  - Invoker: decides which method should be called
  - Receiver: an instance of the class that contains the method's code

Command Pattern

- Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the owner of the method or the method parameters
- This is loose coupling at its finest

Simple Switch Example

```
/* The Invoker class */
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
        this.flipUpCommand = flipUpCmd;
        this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
        flipUpCommand.execute();
    }

    public void flipDown() {
        flipDownCommand.execute();
    }
}
```

Simple Switch Example

```
/* The Receiver class */
public class Light {

    public Light() {  }

    public void turnOn() {
       System.out.println("The light is on");
    }

    public void turnOff() {
       System.out.println("The light is off");
    }
}

/* The Command interface */
public interface Command {
   void execute();
}
```

Simple Switch Example

```
/* The Command for turning the light on in North America, or turning the light
off in most other places */
public class FlipUpCommand implements Command {

   private Light theLight;

   public FlipUpCommand(Light light) {
        this.theLight=light;
   }

   public void execute(){
      theLight.turnOn();
   }
}
```

Simple Switch Example

```
/* The Command for turning the light off in North America, or turning the light
on in most other places */
public class FlipDownCommand implements Command {

   private Light theLight;

   public FlipDownCommand(Light light) {
        this.theLight=light;
   }

   public void execute() {
      theLight.turnOff();
   }
}
```

Simple Switch

```
/* The test class or client */
public class PressSwitch {

   public static void main(String[] args) {
      Light lamp = new Light();
      Command switchUp = new FlipUpCommand(lamp);
      Command switchDown = new FlipDownCommand(lamp);

      // See criticism of this model above:
      // The switch itself should not be aware of lamp details (switchUp, switchDown)
      // either directly or indirectly
      Switch s = new Switch(switchUp,switchDown);

      try {
         if (args[0].equalsIgnoreCase("ON")) {
             s.flipUp();
         } else if (args[0].equalsIgnoreCase("OFF")) {
            s.flipDown();
         } else {
            System.out.println("Argument \"ON\" or \"OFF\" is required.");
         }
      } catch (Exception e){
         System.out.println("Arguments required.");
      }
   }
}
```

Benefits of Switch

- Switch can be used with any device (not just a light)
  - Switch constructor can take any sub-class of Command
  - Could configure to start an engine

In Class Team Exercise

- Each team must:
  - Examine the code in the zip file editor.zip
  - Produce a class diagram that represents the code in the zip as it relates to the Command Pattern
  - Submit your team's solution by end of class. This solution can be hand-written if you wish, just make sure it is legible :-)