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.

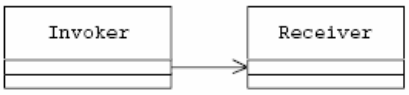
Toni Sellarès
Universitat de Girona

The Command Pattern: Motivation

In general, an object-oriented application makes use of the services of different objects for the processing requirements.

In terms of implementation, the application may depend on *Invoker* objects that invokes operations on different *Receiver* objects.

In this design Invokers and Receivers are closely tied to each other in that they interact with each other directly.



This could result in a set of conditional *if* statements in the implementation of the invoker:

```
…
if (RequestType=TypeA){
//do something
}
```

When a new type of feature is to be added to the application, the existing code needs to be modified and it violates the basic object-oriented open-closed principle.

Command Pattern

Using the Command pattern, the Invoker that issues a request on behalf of the client and the set of service-rendering Receiver objects can be decoupled.

The Command pattern suggests creating an abstraction for the processing to be carried out or the action to be taken in response to client requests.

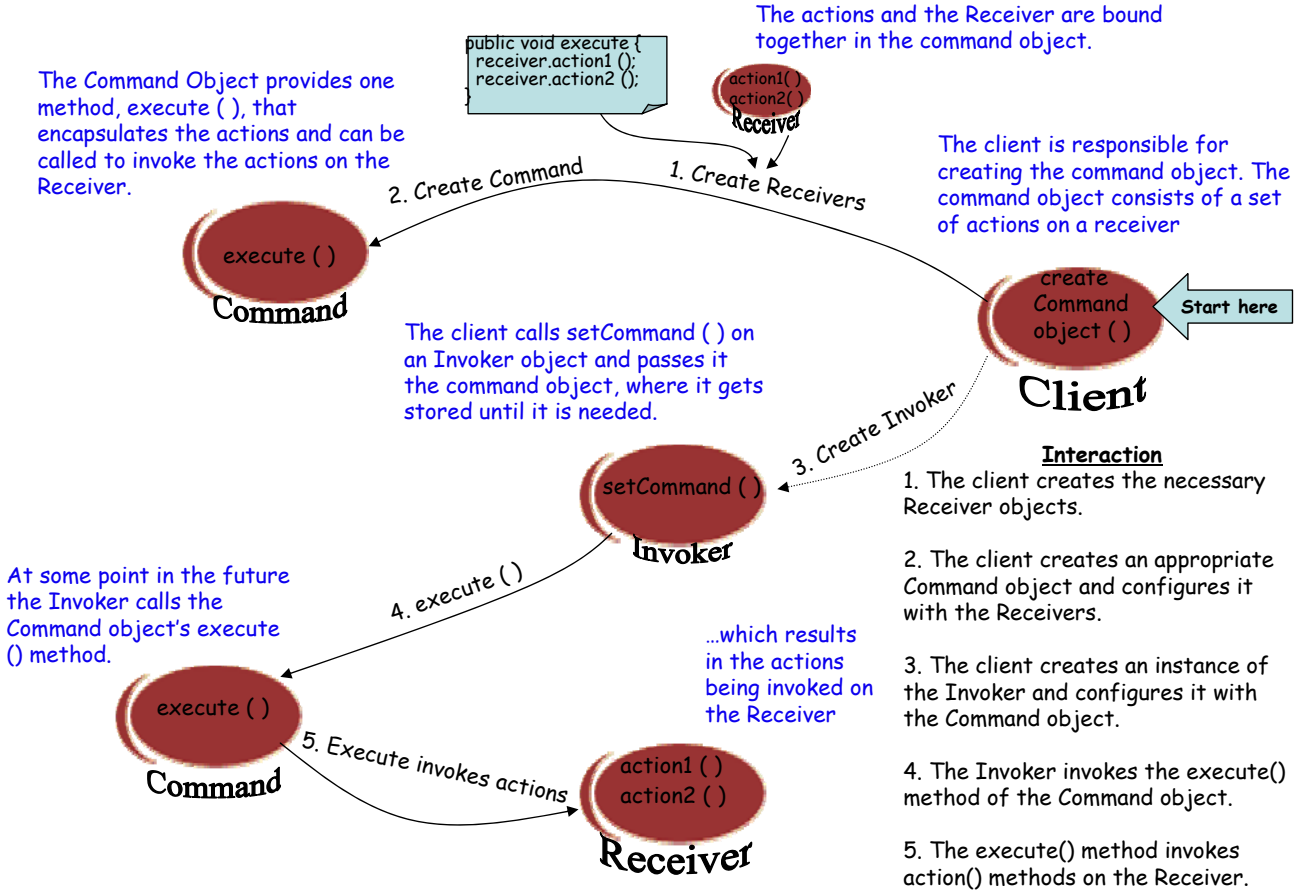
This abstraction can be designed to declare a common interface to be implemented by different concrete implementers referred to as *Command objects*.

Each Command object represents a different type of client request and the corresponding processing.

A given Command object is responsible for offering the functionality required to process the request it represents, but it does not contain the actual implementation of the functionality.

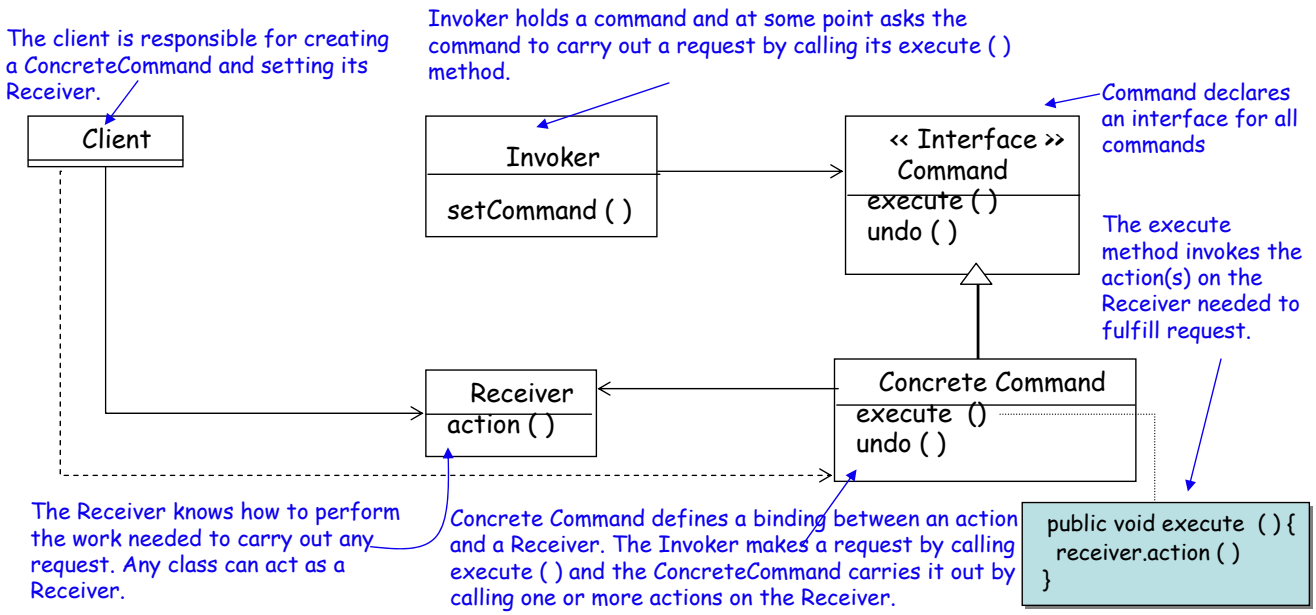
Command objects make use of Receiver objects in offering this functionality.

The Command Pattern



The Command Pattern Defined

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.



Command Pattern: participants

Command:

- Declares an interface for executing an operation.

ConcreteCommand :

- Implements execute( ) by invoking the corresponding operation(s) on Receiver(s).

Client:

- Creates a ConcreteCommand object and sets its Receiver.

Invoker:

- Asks its Command to carry out the request.

Receiver:

- Any classes that know how to perform the operation(s) associated with carrying out a request.

Command Pattern: Structural Code

```
/**
 * Test driver for the pattern.
 */
public class Test {
        public static void main( String arg[] ){
                Client client = new Client();
                Command command = client.setup();
                 command.execute();
                 }
}

/**
 * Knows how to perform the operations associated with
 * carrying out a request. Any class may serve as a Receiver.
 */
public interface Receiver {
     void action();
}
```

```
/**
 * Implementation of the Receiver interface.
 */
public class ConcreteReceiver implements Receiver {
         public void action()
         { … }
}

/**
 * Defines a binding between a Receiver object and an action. Implements
 * Execute by invoking the corresponding operation on Receiver.
 */
public class ConcreteCommand implements Command {
         private Receiver receiver;
         public void setReceiver( Receiver receiver ) {
                  this.receiver = receiver;
         }
         public Receiver getReceiver() {
                  return receiver;
         }
         public void execute() {
                  receiver.action();
         }
}
```

```
/**
 * Declares an interface for executing an operation.
 */
public interface Command {
        void setReceiver( Receiver receiver );
        Receiver getReceiver();
        void execute();
}


/**
 * Creates a ConcreteCommand object and specifies its
receiver.
 */
public class Client {
        public Command setup() {
                Command command = new ConcreteCommand();
                Receiver receiver = new ConcreteReceiver();
                command.setReceiver( receiver );
                // We return the command so that the Invoker may use it.
                return command;
        }
}
```

Command Pattern

The client/invoker does not directly interact with Receiver objects and therefore, they are completely decoupled from each other.

You can undo/redo any Command:

- Each Command stores what it needs to restore state.

You can store Commands in a stack or queue:

- Command processor pattern maintains a history.

It is easy to add new Commands, because you do not have to changeexisting classes:

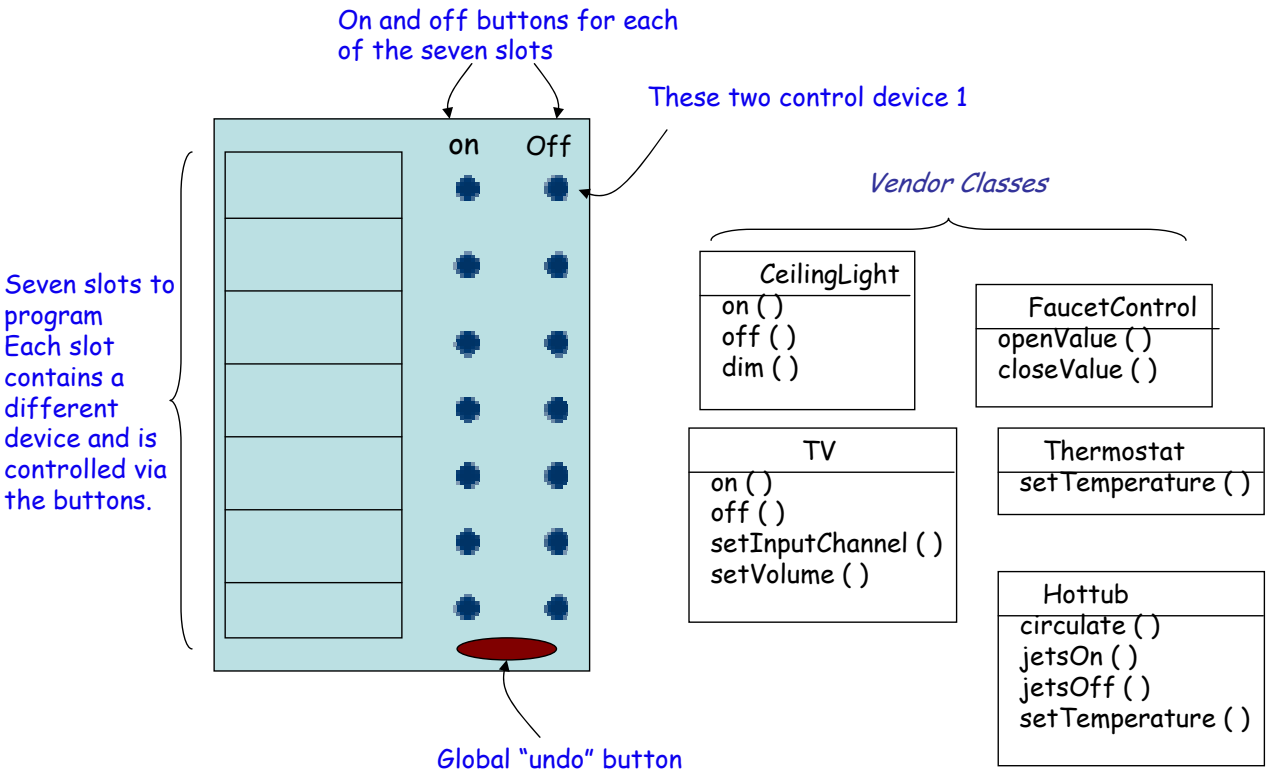
- Command is an abstract class, from which you derive new classes.
- execute(), undo() and redo() are polymorphic functions.

Command Pattern: Remote Control Example

- Goal: Program a Remote Control that can be used to control the devices in the house.

- Requirements:
  - Remote control features seven programmable slots
  - Each slot can be assigned to a different household device
  - Each slot has a corresponding on/off button
  - Remote has a global undo button that undoes the last button pressed.
  - A set of vendors classes are provided to give you an idea of the interfaces of the objects that need to be controlled from the remote.

A Pictorial View



Observations

- *Expectation*: see a bunch of classes with `on ( )` and `off ( )` methods --> common interface.
- *Instead*: each vendor class has its own API: `dim ( )`, `setTemperature ( )`, `setVolume ( )`, `setDirection ( )`.
- *Comment*:
  - Separation of concerns: remote should know how to interpret button presses and make requests, but it really should not know anything about home automation or how to turn on a hot tub.
  - If the remote is dumb, how do we design a remote so that it can invoke an action: turn on a light?
- *Requirement*: There will be more vendor classes in the future - design must accommodate for that.

Solution: the Command Pattern!

- Command Pattern allows you to decouple the *requester* of an action from the *object* that actually performs the action:
  - Requester == remote control, Object == vendor classes
- How does it work?
  - Introduce “command objects” into your design. A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light).
  - Every time a remote button is pressed, its corresponding command object goes to work!
  - Remote does not have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done.
- Here the *remote is decoupled from the object*!

- Implementing the Command Interface:

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

Simple. All we need is one method called execute ( ).

- Implementing a Command to turn the light on:

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

This is a command, so we need to implement the Command interface.

The constructor is passed the specific light that this command is going to control - say the living room light - and stashes it in the light instance variable. When execute ( ) gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on ( ) method on the receiving object, which is the light that we are controlling.

- 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 our command, which will control one device.

We have a method for setting the command the slot it's going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote 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.

- Creating a Simple test to use the Remote Control:

```
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 ( );
    }
}
```

This is our client in Command Pattern-speak

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

Now we create a Light Object - this will be the Receiver of the request.

Here, create a command and pass it to the Receiver.

Here pass the command to the Invoker.

Then, we simulate the button being pressed.

- Implementing the RemoteControl:

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

    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;
        }
    }

    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();
    }

    public String toString() {
        // code here  }
```

This time around the remote is going to handle seven On and Off Commands which we will hold in the corresponding array.

In the constructor all we need to do is instantiate the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the On and Off array for later use.

When an On or a Off button is pressed, the hardware takes care of calling the corresponding methods OnButtonWasPushed ( ) or OffButtonWasPushed ( ).

Used for testing.

- Implementing the Commands:

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

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

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

The LightOffCommand works exactly like the LightOnCommand, except that we are binding the receiver to a different action: the off() command.

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

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

      public void execute() {
            stereo.on();
            stereo.setCD();
            stereo.setVolume(11);
      }
}
```

An instance of the stereo we are going to control is passed to constructor.

To carry out this request we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11 !

- Testing the code:

```
public class RemoteLoader {

      public static void main(String[] args) {
            RemoteControl remoteControl = new RemoteControl();

            Light livingRoomLight = new Light("Living Room");

            LightOnCommand livingRoomLightOn =
                        new LightOnCommand(livingRoomLight);
            LightOffCommand livingRoomLightOff =
                        new LightOffCommand(livingRoomLight);

           remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

            remoteControl.onButtonWasPushed(0);
            remoteControl.offButtonWasPushed(0);
}
```

Create all the devices

Create all the Command objects

Load the command objects

Ready to roll! Test out the remote!

The NoCommand

- In the remote control, we didn't want to check to see if a command was loaded for every slot. To do this we would need to do:

```
public void onButtonWasPushed (int slot) {
    if (onCommands[slot] != null ){
      onCommands[slot].execute ( );
    }
}
```

- To get around it, we introduce a new "NoCommand" object:

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

- Fixing the RemoteControl constructor:

```
Command noCommand = new NoCommand ( );
for (int j=0; I < 7; j++ ){
   onCommands[I] = noCommand;
   off Commands[I] = noCommand;
}
```

NoCommand is an example of a null object. A null object is useful when you don't have a meaningful object to return, yet you want to remove the responsibility for handling null from the client.

Adding the Undo Feature

- When commands support undo, they have an undo ( ) method that mirrors the execute ( ) method. Whatever execute ( ) last did, undo ( ) reverses it:

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

- Implement the undo method in the LightOnCommand:

```
public class LightOnCommand implements Command {

  public LightOnCommand (Light light){
      this.light = light;
  }
  public void execute ( ) {
      light.on ( );
  }
  public void undo ( ) {
     light.off ( );
  }
}
```

execute ( ) turns the light on so undo turns it off. Reverse for LightOffCommand.

Updating the RemoteControl class

- To add support for the undo button we only have to make a few small changes to RemoteControl class:

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

        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;
        }
```

This is where we will stash the last command executed for the undo button

Just like the other slots this is initialized to a NoCommand.

RemoteControl Class

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

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

        public void offButtonWasPushed(int slot) {
                offCommands[slot].execute();
                undoCommand = onCommands[slot];
        }

        public void undoButtonWasPushed ( ) {
               undoCommand.undo ( );
        }

        public String toString() {
               // code here  }
}
```

When the button is pressed we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" and "off" commands.

When the undo button is pressed, we invoke the undo ( ) method of the command stored in the undoCommand. This reverses the operation that was last executed.

Using State to Implement 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;
        String location;
        int speed;

        public CeilingFan(String location) {
                this.location = location;
                speed = OFF; }

        public void high() {
                speed = HIGH;
                System.out.println(location + " ceiling fan is on high");       }

        public void medium() {
                speed = MEDIUM;
                System.out.println(location + " ceiling fan is on medium");       }

        public void low() {
                speed = LOW;
                System.out.println(location + " ceiling fan is on low");       }

        public void off() {
                speed = OFF;
                System.out.println(location + " ceiling fan is off");       }

         public int getSpeed() {
                return speed;       }
}
```

Holds local state

Set the speed of the ceiling fan.

Can get the current speed.

Adding Undo to Ceiling Fan Commands

```
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();
                }
        }
}
```

Added local state to keep track of the previous speed of the fan.

In execute ( ), before we change the speed we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous state.

The Party Mode

- Executing multiple commands with one:
  - Push one button to dim the lights, turn on the stereo and the TV and set the to DVD.

```
public class MacroCommand implements Command {
        Command[] commands;
         public MacroCommand (Command[ ] commands){
              this.commands = commands;
         }
         public void execute  ( ) {
                  for (int j = 0; j < commands.length; j++){
                        commands[I].execute ( );
                   }
         }
}
```

Take an array of Commands and store them in the MacroCommand

When the macro gets executed by the remote, execute those commands one at a time.

Using the Macro Command

1. Create the set of commands we want to go into the macro,

2. Create two arrays, one for the On Commands and the other for the Off Commands and load them with the corresponding commands,

3. Assign the macroCommand to the button as before,

4. Then push some buttons to test it out!

Command Pattern: Queuing Requests

- Commands provide a nice mechanism to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object.
- Imagine a job queue:
  - Add commands to the queue on one end, and on the other end sits a group of threads
  - Threads run the following script:
    - Remove command from the queue
    - Call its execute method
    - Wait for the call to finish
    - Discard the command object and retrieve the new one.
- Here job queues are totally decoupled from the objects that are doing the computation - one minute the thread may be doing a financial computation and the next retrieving something from the network.

Command Pattern: Logging Requests

- Semantics of some applications require that we log all actions and be able to recover after a crash by re-invoking those actions. The Command Pattern can store these semantics with the addition of two new methods: `store ( )` and `load ( )`.
- How does it work?
  - As we execute commands, we store a history of them on disk
  - When a crash occurs, we reload the command objects and invoke their `execute` () methods in batch and in order.

Summary

- The Command pattern decouples an object making a request from one that knows how to perform it.
- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- 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 the Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute ( ) method was last called.
- *Macro commands* are a simple extension of Command that allow multiple commands to be invoked.
- Commands may also be used to implement *logging* and *transactional* systems.