

# Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood\_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

# Command Pattern

# Command Pattern

- Lets you encapsulate actions within Java classes, where each class has an "execute()" method which is declared in the Command interface the class implements.
- Builds your Command objects into a "ready to run" state.
- Pass those objects to an "invoker" class which makes a callback to the execute() method of each class at the appropriate time.
- Invoker class doesn't know anything about the logic each Command object contains; it simply knows that it should call the execute() method on an object when an event occurs.
- Lets you create macros in your applications by building a series of Command objects that have execute() -- and optionally undo() -- methods.

# Applications- Command pattern

- Handling actions for Java menu items and buttons.
- Providing support for macros (recording and playback of macros).
- Providing "undo" support.
- Java progress bars.
- Java wizards.

# Terminology

- Three terms always associated with the command pattern are client, invoker and receiver.
- The ***client*** instantiates the command object and provides the information required to call the method at a later time.
- The ***invoker*** decides when method should be called.
- The ***receiver*** is an instance of the class that contains the method's code (the execute() method of the class).
- Command pattern is a data driven design pattern and falls under **behavioral pattern** category.

# Implementing Command

- Define a Java Command interface with a method named `execute()`.
- Create Command classes that implement this interface. The `execute()` method of each object will contain the logic specific to the action you are encapsulating. (For instance, you might create a `FileOpenCommand` and a `FileCloseCommand` in a text editor application.)
- In an Model/View/Controller GUI program you will have a controller class that constructs each of your Command objects. (The "client" class.)
- Each of these "ready to run" command objects is passed to an invoker class.
- The invoker class calls the `execute()` method of a Command object at the appropriate time. In a GUI application this may happen when a user selects a menu item or presses a button.

# Example #1

- GUI code written by a new developer, you might run across an actionPerformed() method in a Swing application that looks like this:

```
public void actionPerformed(ActionEvent e)
{
    Object o = e.getSource();
    if (o == fileNewMenuItem)
        doFileNewAction();
    else if (o == fileOpenMenuItem)
        doFileOpenAction();
    else if (o == fileOpenRecentMenuItem)
        doFileOpenRecentAction();
    else if (o == fileSaveMenuItem)
        doFileSaveAction();
    // and more ...
}
```

# Example #1

- **A Java Command Pattern solution**
- 1) We create a simple Java Command interface, like this:

```
// the Command Pattern in Java
public interface Command
{
    public void execute();
}
```

- 2) Next, we modify our JMenuItem and JButton classes to implement our Command interface, as shown here:

```
public class FileOpenMenuItem extends
JMenuItem implements Command
{
    public void execute()
    {
        // your business logic goes here
    }
}
```



# Example #1

- 3) Next, we add our menu items and buttons to our application as we did before. (we won't show that here, because this portion of the application doesn't change)
- 4) Finally, after these changes, our new actionPerformed() method in our ActionListener class looks like this:

```
public void actionPerformed(ActionEvent e)
{
    Command command = (Command)e.getSource();
    command.execute();
}
```

# Example #1 : Advantages

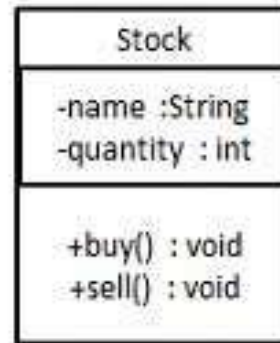
- The actionPerformed() method is much simpler.
- All of the "if" statements in the actionPerformed method have been eliminated.
- We have a new Java Command interface.
- We have a collection of small Java classes (FileOpenMenuItem etc.) that implement the Command interface. The logic for each action is small, and self-contained in each execute() method.
- The actionPerformed method has no idea what actual Command it is running.

# Example #2

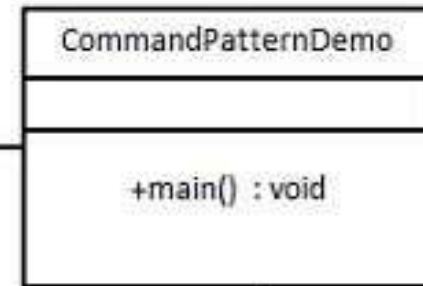
- We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.
- *Broker* object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern.

# Example #2

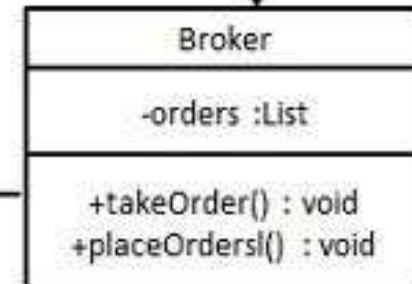
2. request/ receiver class



5. Client : Use Broker class to take and execute commands.



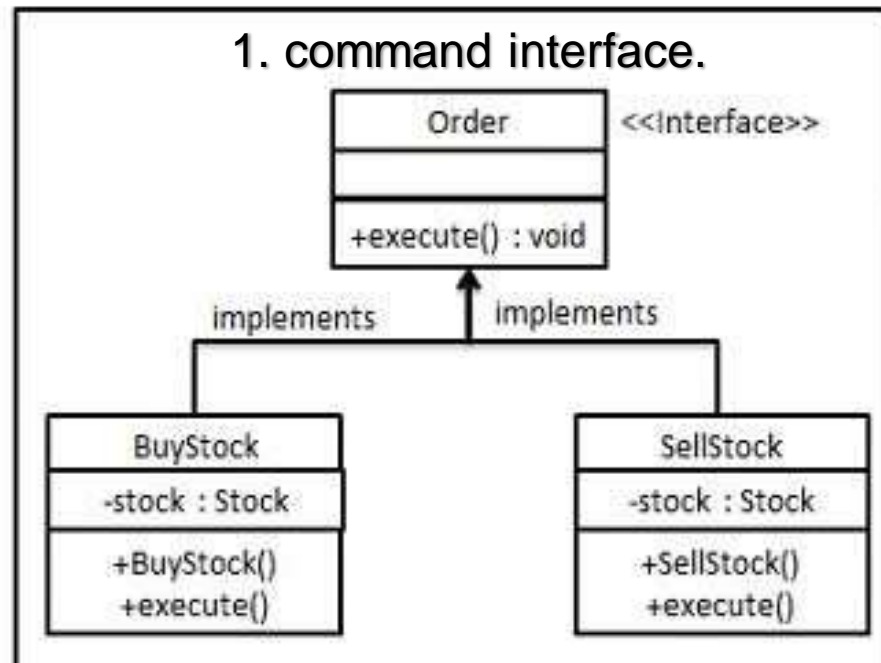
uses



4. command invoker class.

uses

1. command interface.



3. concrete classes implementing the *Order* interface.

# Example #2

- **Step 1:** Create a command interface.

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

- **Step 2:** Create a request class. *Stock.java*

```
public class Stock {  
    private String name = "ABC";  
    private int quantity = 10;  
    public void buy(){  
        System.out.println("Stock [ Name: "+name+  
            ", Quantity: " + quantity + " ] bought");  
    }  
    public void sell(){  
        System.out.println("Stock [ Name: "+name+  
            ", Quantity: " + quantity + " ] sold");  
    }  
}
```

# Example #2

- **Step 3:** Create concrete classes implementing the *Order* interface - *BuyStock.java* - *SellStock.java*

```
public class BuyStock implements Order{  
    private Stock abcStock;  
    public BuyStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
    public void execute() {  
        abcStock.buy();  
    }  
}
```

```
public class SellStock implements Order{  
    private Stock abcStock;  
    public SellStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
    public void execute() {  
        abcStock.sell();  
    }  
}
```

# Example #2

- **Step 4:** Create command invoker class. *Broker.java*

```
import java.util.ArrayList;
import java.util.List;
public class Broker {
    private List<Order> orderList = new ArrayList<Order>();
    public void takeOrder(Order order){
        orderList.add(order);
    }
    public void placeOrders(){
        for (Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}
```

# Example #2

- **Step 5:** Use Broker class to take and execute commands - *CommandPatternDemo.java*

```
public class CommandPatternDemo {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Stock abcStock = new Stock();  
        BuyStock buyStockOrder = new BuyStock(abcStock);  
        SellStock sellStockOrder = new SellStock(abcStock);  
        Broker broker = new Broker();  
        broker.takeOrder(buyStockOrder);  
        broker.takeOrder(sellStockOrder);  
        broker.placeOrders();  
    }  
}
```

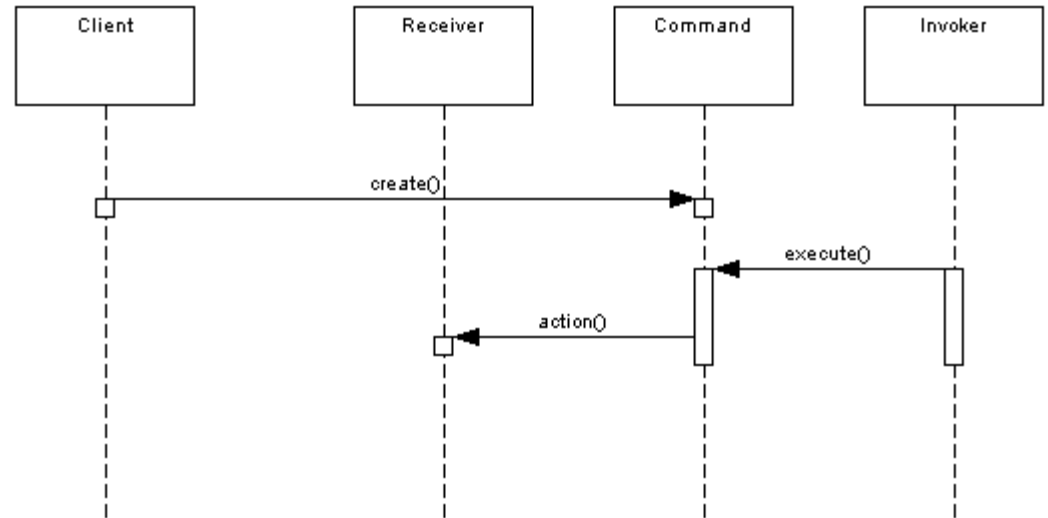
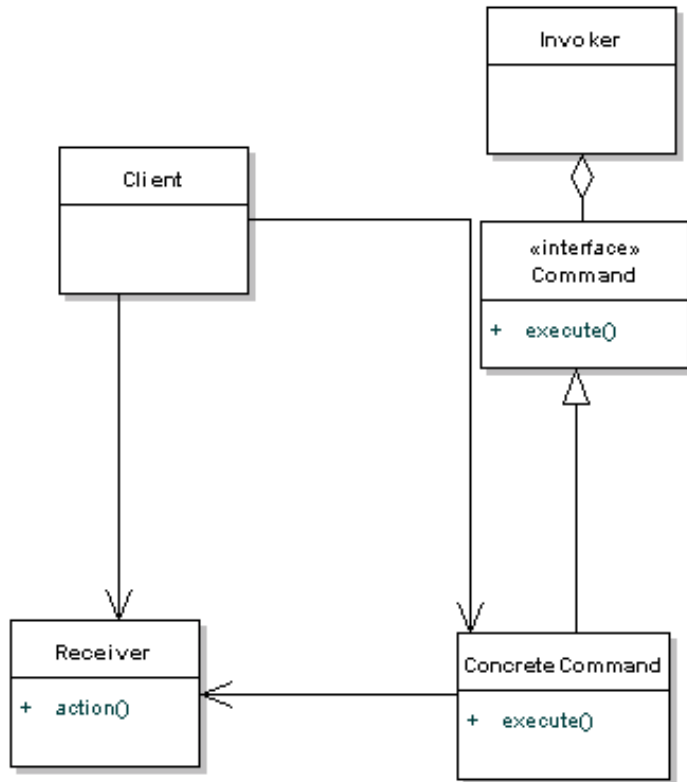
- **Step 6:** Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought  
Stock [ Name: ABC, Quantity: 10 ] sold
```



# Summary - Command

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



**Command** declares an interface for all commands, providing a simple **execute()** method which asks the **Receiver** of the command to carry out an operation. The **Receiver** has the knowledge of what to do to carry out the request. The **Invoker** holds a command and can get the **Command** to execute a request by calling the execute method. The **Client** creates **ConcreteCommands** and sets a **Receiver** for the command. The **ConcreteCommand** defines a binding between the action and the receiver. When the **Invoker** calls execute the **ConcreteCommand** will run one or more actions on the **Receiver**.

# Hollywood Principle

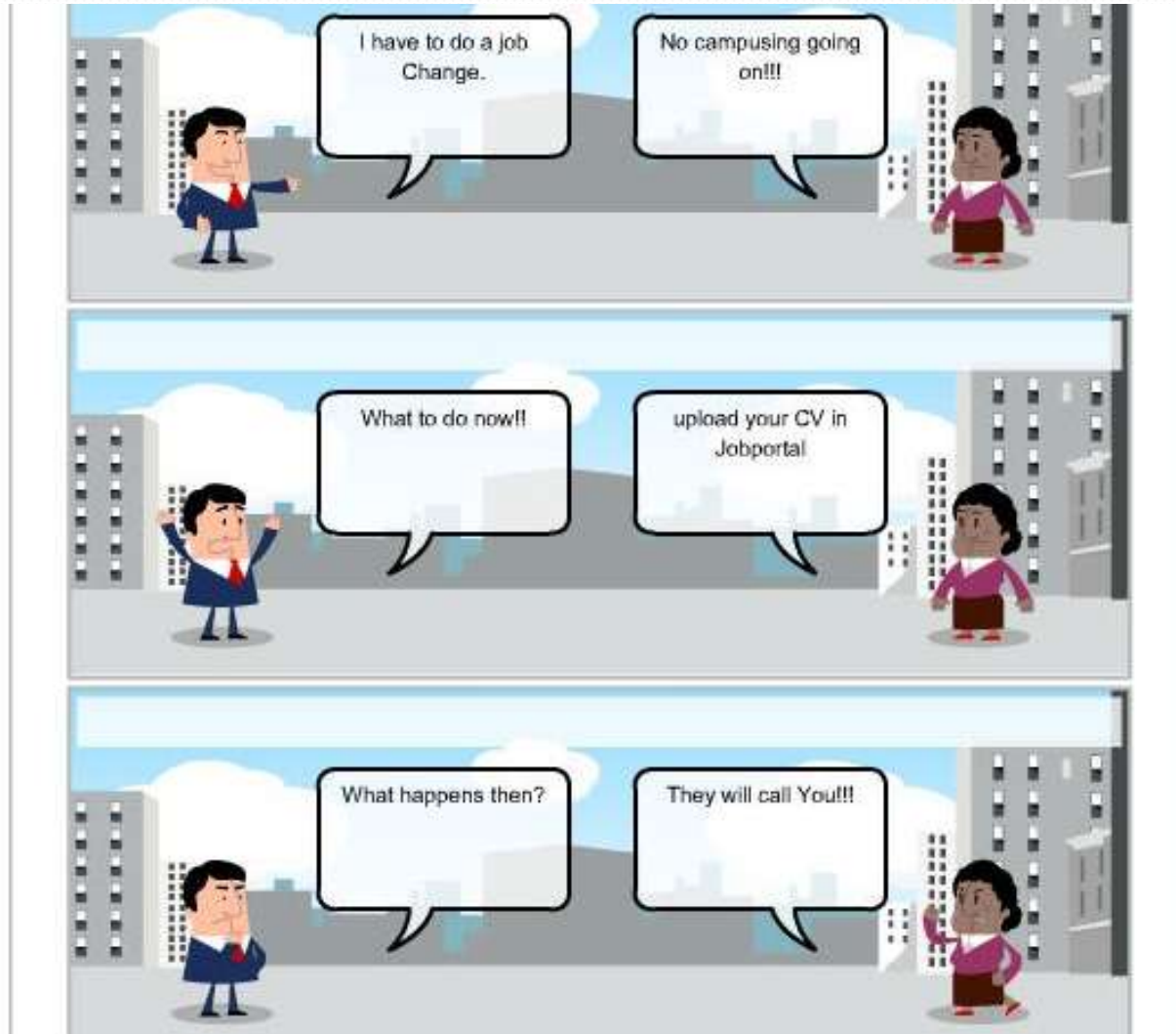
- Hollywood Principle says, "Don't call us, we'll call you."
- closely related to the **Dependency Inversion Principle**, and illustrates a different way of writing software from the more traditional form of programming.
- To explain "you" and "they" in technical terms, first we need to understand how software design works. When we design a software, we try to implement two things.
  - API
  - Framework

# Hollywood Principle

- **API** is used to publish some methods/functions, so the caller/user of the API calls this method to get some useful information. So, the caller does not have any action points to take — only call methods and outputs.
- **Framework** is a little bit more critical than the API. The framework takes the strategy or business implementation from the caller and calls it when required.
- With the Hollywood Principle, we can feed our strategy / business logic, denoting the framework engine/implementation, which calls the fed strategy when required.

# Hollywood Principle

- Use Case



# Hollywood Principle

```
public class Resume {  
    private String email;  
    private String name;  
    private String content;  
    /* getters and setters */  
    @Override  
    public String toString() {  
        return "Resume [email=" + email + ", name=" + name + ", content=" + content + "];"  
    }  
}
```

# Hollywood Principle

```
public class JobPortal {  
    private static JobPortal portal = new JobPortal();  
    private List<Resume> resumeList = new ArrayList<Resume>();  
    public static JobPortal get(){  
        return portal;  
    }  
    private JobPortal() {}  
    public void upload(String mail ,String name,String content)  
    {  
        Resume res = new Resume();  
        res.setName(name);  
        res.setEmail(mail);  
        res.setContent(content);  
        resumeList.add(res);  
    }  
    public void triggerCampusing(){  
        for(Resume res : resumeList){  
            System.out.println("Sending mail to " + res.getName() + " at " + res.getEmail());  
        }  
    }  
}
```

# Hollywood Principle

```
public class HollywoodTest {  
    public static void main(String[] args) {  
        JobPortal.get().upload("shamik@xyz.com", "Shamik Mitra", "A java developer");  
        JobPortal.get().upload("Ajanta@vvv.com", "Ajanta Mitra", "A PHP developer");  
        JobPortal.get().upload("Swastika@vvv.com", "Swastika Mitra", "A Java developer");  
        JobPortal.get().upload("Mayukh@vvv.com", "Mayukh Mitra", "A Network engineer");  
        JobPortal.get().upload("Samir@123.com", "Samir Mitra", "A java Architect");  
        // Now trigger campusing  
        JobPortal.get().triggerCampusing();  
    }  
}
```

## ● Output

Sending mail to Shamik Mitra at shamik@xyz.com  
Sending mail to Ajanta Mitra at Ajanta@vvv.com  
Sending mail to Swastika Mitra at Swastika@vvv.com  
Sending mail to Mayukh Mitra at Mayukh@vvv.com  
Sending mail to Samir Mitra at Samir@123.com



# Hollywood Principle

- Example1: ASP.NET Web Form might have in its codebehind page event handlers to respond to Page\_Load and Button\_Click events. An ASP.NET developer writes code to respond to these external framework events, instead of owning the execution of the web server and making all decisions and method calls accordingly.
- Example2: The observer pattern follows the Hollywood principle. This pattern allows to observe the state of an object in a well defined manner. It is typically implemented by injecting a callback object (observer) into the class to be observed (subject). The subject simply raises an event in all observers when its state changes. How the observer reacts to the event is outside the scope or care of the subject.



# Law of Demeter & Principle of least Knowledge

- A module should not know about the inner details of the objects it manipulates. If a code depends upon internal details of a particular object, there is good chance that it will break as soon as internal of that object changes.
- It's better not to have a chain of methods, originating from unknown object, which may change.
- **Law of Demeter**
- According to Law of Demeter, a method M of object O should only call following types of methods:
  - Methods of Object O itself
  - Methods of Object passed as an argument
  - Method of object, which is held in instance variable
  - Any Object which is created locally in method M

# Law of Demeter - Principle of least Knowledge

- This method shows two violations of "Law of Delimiter" or "Principle of least knowledge"

```
public class LawOfDelimiterDemo {  
    public void process(Order o) {  
        // as per rule 1, this method invocation is fine, because o is a argument of process() me  
        Message msg = o.getMessage();  
        // this method call is a violation, as we are using msg, which we got from Order.  
        // We should ask order to normalize message, e.g. "o.normalizeMessage();"   
        msg.normalize();  
        // this is also a violation, instead using temporary variable it uses method chain.  
        o.getMessage().normalize();  
        // this is OK, a constructor call, not a method call.  
        Instrument symbol = new Instrument();  
        // as per rule 4, this method call is OK, because instance of Instrument is created locally  
        symbol.populate();  
    }  
}
```

# Law of Demeter - Principle of least Knowledge

- Example 2

```
public class XMLUtils {  
    public Country getFirstBookCategoryFromXML(XMLMessage xml) {  
        return  
xml.getXML().getBooks().getBookArray(0).getBookHeader().getBookCategory();  
    }  
}
```

- This code is now dependent upon lot of classes e.g.

XMLMessage

XML

Book

BookHeader

BookCategory