DESIGN PATTERNS ASSIGNMENT – AU JULY 2020

COMMAND

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Command pattern is a data driven design pattern and falls under behavioural pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

It is used when the intent is to:
1. Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
2. Promote "invocation of a method on an object" to full object status
3. An object-oriented callback

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.
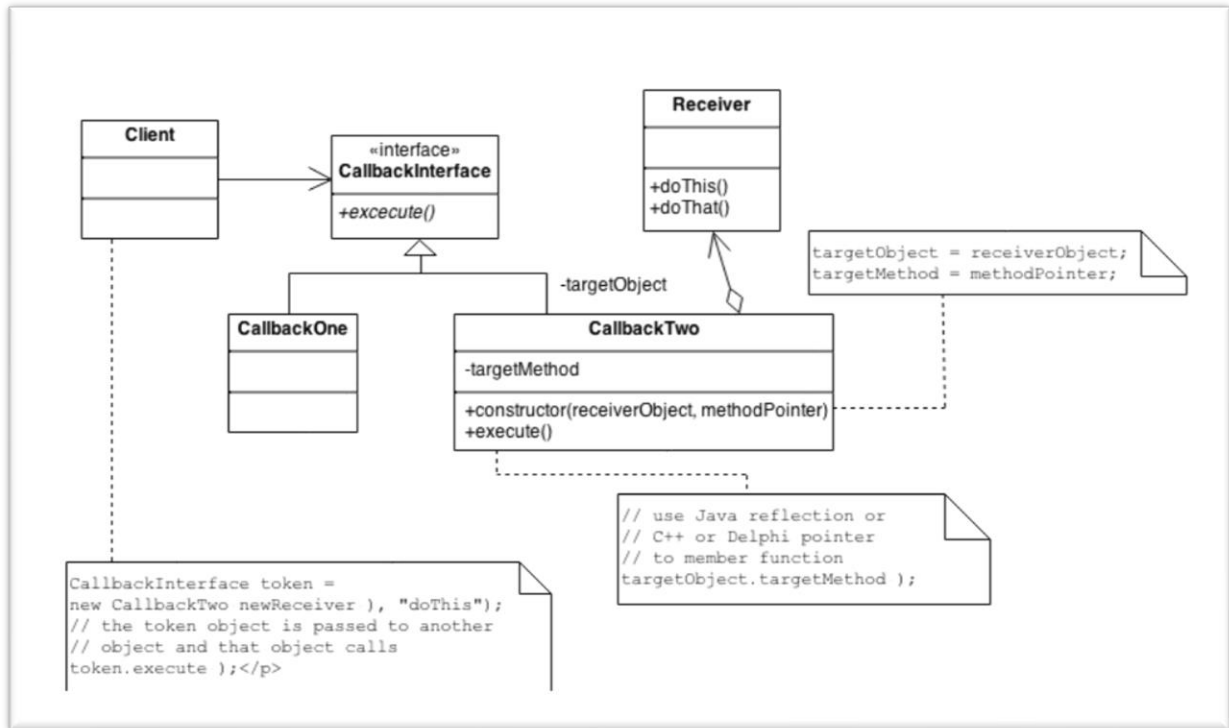
All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command's "execute" method then causes the pieces to come together.

Sequences of Command objects can be assembled into composite (or macro) commands.

Structure
The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object. Command objects can be thought of as "tokens" that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Client

«interface»
CallbackInterface

+excecute()

Receiver

+doThis()
+doThat()

targetObject = receiverObject;
targetMethod = methodPointer;

-targetObject

CallbackOne

CallbackTwo

-targetMethod

+constructor(receiverObject, methodPointer)
+execute()

// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod);

CallbackInterface token =
new CallbackTwo newReceiver), "doThis");
// the token object is passed to another
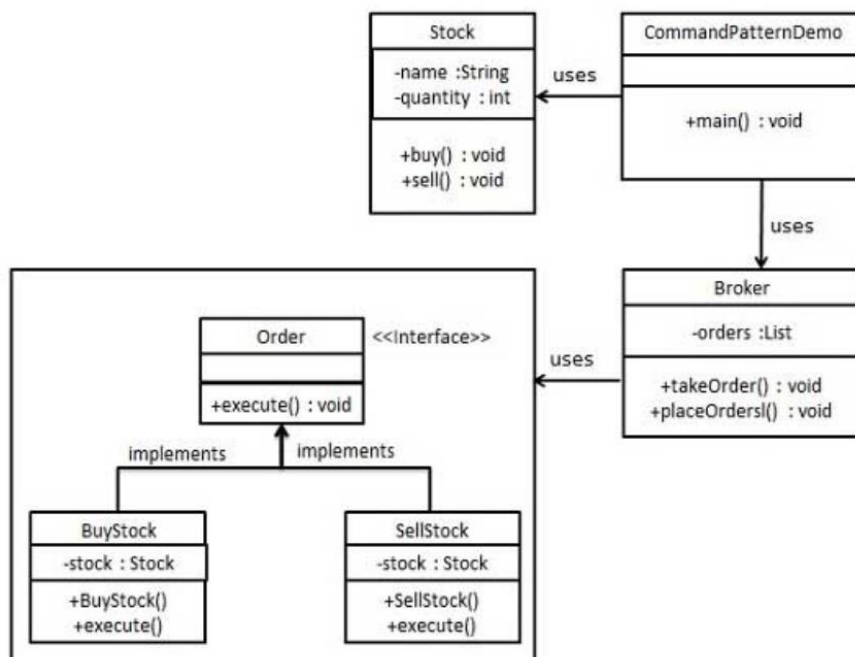// object and that object calls
token.execute);</p>

Implementation

I have created an interface *Order* which is acting as a command. Then created a *Stock* class which acts as a request.

The 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*, the demo class, will use *Broker* class to demonstrate command pattern.

Stock

-name :String
-quantity : int

+buy() : void
+sell() : void

uses

CommandPatternDemo

+main() : void

uses

Order <<Interface>>

+execute() : void

implements   implements

Broker

-orders :List

+takeOrder() : void
+placeOrdersl() : void

uses

BuyStock

-stock : Stock

+BuyStock()
+execute()

SellStock

-stock : Stock

+SellStock()
+execute()

Order.java

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

Create a request class.

Stock.java

```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");
   }
}
```

Create concrete classes implementing the *Order* interface.

BuyStock.java

```java
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```java
public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

Create command invoker class.

Broker.java

```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();
   }
}
```

Use the Broker class to take and execute commands.

CommandPatternDemo.java

```java
public class CommandPatternDemo {
   public static void main(String[] args) {
      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();
   }
}
```