Command Pattern

The Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command design pattern provides the options to queue commands, undo/redo actions and other manipulations.

## Intent

- encapsulate a request in an object  
- allows the parameterization of clients with different requests  
- allows saving the requests in a queue.

Problem

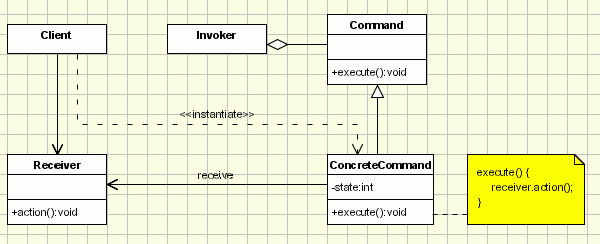
Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

In the command pattern, a Command interface declares a method for executing a particular action. Concrete Command classes implement the execute() method of the Command interface, and this execute() method invokes the appropriate action method of a Receiver class that the Concrete Command class contains. The Receiver class performs a particular action. A Client class is responsible for creating a Concrete Command and setting the Receiver of the Concrete Command. An Invoker class contains a reference to a Command and has a method to execute the Command.

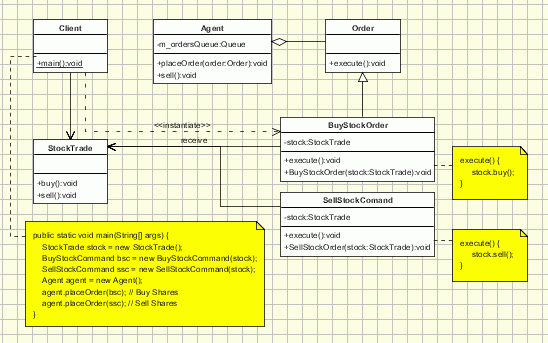
In the command pattern, the invoker is decoupled from the action performed by the receiver. The invoker has no knowledge of the receiver. The invoker invokes a command, and the command executes the appropriate action of the receiver. Thus, the invoker can invoke commands without knowing the details of the action to be performed. In addition, this decoupling means that changes to the receiver's action don't directly affect the invocation of the action.

The command pattern can be used to perform 'undo' functionality. In this case, the Command interface should include an unexecute() method.

## **Implementation**

The idea and implementation of the Command design pattern is quite simple, as we will see in the diagram below, needing only few extra classes implemented.

The classes participating in the pattern are:  
- Command - declares an interface for executing an operation;  
- ConcreteCommand - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.  
- Client - creates a ConcreteCommand object and sets its receiver;  
- Invoker - asks the command to carry out the request;  
- Receiver - knows how to perform the operations;  
  
The Client asks for a command to be executed. The Invoker takes the command, encapsulates it and places it in a queue, in case there is something else to do first, and the ConcreteCommand that is in charge of the requested command, sending its result to the Receiver.  
  
Here is a sample code of a classic implementation of this pattern for placing orders for buying and selling stocks:



The client creates some orders for buying and selling stocks (ConcreteCommands). Then the orders are sent to the agent (Invoker).The agent takes the orders and place them to the StockTrade system (Receiver). The agent keeps an internal queue with the order to be placed. Let's assume that the StockTrade system is closed each Monday, but the agent accepts orders, and queue them to be processed later on.

|  |
| --- |
| public interface Order {     public abstract void execute ( ); }  // Receiver class. class StockTrade {     public void buy() {         System.out.println("You want to buy stocks");     }     public void sell() {         System.out.println("You want to sell stocks ");     } }  // Invoker. class Agent {     private m\_ordersQueue = new ArrayList();      public Agent() {     }          void placeOrder(Order order) {         ordersQueue.addLast(order);         order.execute(ordersQueue.getFirstAndRemove());     }     }  //ConcreteCommand Class. class BuyStockOrder implements Order {     private StockTrade stock;     public BuyStockOrder ( StockTrade st) {         stock = st;     }     public void execute( ) {         stock . buy( );     } }  //ConcreteCommand Class. class SellStockOrder implements Order {      private StockTrade stock;     public SellStockOrder ( StockTrade st) {         stock = st;     }     public void execute( ) {         stock . sell( );     } }  // Client public class Client {     public static void main(String[] args) {         StockTrade stock = new StockTrade();         BuyStockOrder bsc = new BuyStockOrder (stock);         SellStockOrder ssc = new SellStockOrder (stock);         Agent agent = new Agent();          agent.placeOrder(bsc); // Buy Shares         agent.placeOrder(ssc); // Sell Shares     } } |

Applicability & Examples

The applicability of the Command design pattern can be found in these cases below:

- parameterizes objects depending on the action they must perform

- specifies or adds in a queue and executes requests at different moments in time

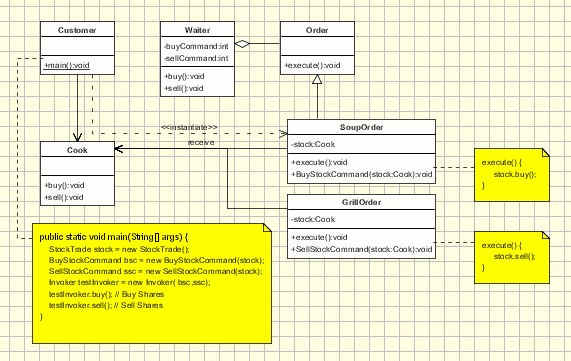
- offers support for undoable actions (the Execute method can memorize the state and allow going back to that state)

- structures the system in high level operations that based on primitive operations

- decouples the object that invokes the action from the object that performs the action. Due to this usage it is also known as Producer - Consumer design pattern.

The example of the meal order at a restaurant is a very good one when trying to explain better how the pattern works:

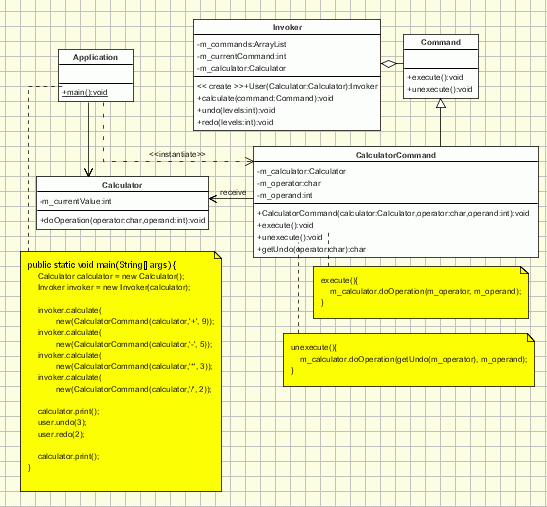
The waiter (Invoker) takes the order from the customer on his pad. The order is then queued for the order cook and gets to the cook (Receiver) where it is processed.



In this case the actors in the scenario are the following: The Client is the customer. He sends his request to the receiver through the waiter, who is the Invoker. The waiter encapsulates the command (the order in this case) by writing it on the check and then places it, creating the ConcreteCommand object which is the command itself. The Receiver will be the cook that, after completing work on all the orders that were sent to him before the command in question, starts work on it. Another noticeable aspect of the example is the fact that the pad for the orders does not support only orders from the menu, so it can support commands to cook many different items.

Just the same way we can consider the example of an auto-repair shop. People come in with different cars that have different problems. The person at the front desk takes their information and places the car in a queue for repair. The information on the order is encapsulated in the paper the car owner will use when he comes back to pick up the fixed car. At some point the car will become the first item in the queue and the mechanic will repair it. Just as in the example above, the Client is the customer. The Invoker is the person at the front desk that takes the information on the car and its problems, the ConcreteCommand is the request for fixing the car and the Receiver is the mechanic.

The most used implementation of the command pattern is the one used to implement the undo options in applications:



Here is an example of the command pattern. We have a Command interface with an execute() method.

### [Command.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/Command.java)

**package** com.cakes;

**public** **interface** Command {

**public** **void** execute();

}

LunchCommand implements Command. It contains a reference to Lunch, a receiver. Its execute() method invokes the appropriate action on the receiver.

### [LunchCommand.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/LunchCommand.java)

**package** com.cakes;

**public** **class** LunchCommand **implements** Command {

Lunch lunch;

**public** LunchCommand(Lunch lunch) {

**this**.lunch = lunch;

}

@Override

**public** **void** execute() {

lunch.makeLunch();

}

}

The DinnerCommand is similar to LunchCommand. It contains a reference to Dinner, a receiver. Its execute() method invokes the makeDinner() action of the Dinner object.

### [DinnerCommand.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/DinnerCommand.java)

**package** com.cakes;

**public** **class** DinnerCommand **implements** Command {

Dinner dinner;

**public** DinnerCommand(Dinner dinner) {

**this**.dinner = dinner;

}

@Override

**public** **void** execute() {

dinner.makeDinner();

}

}

Lunch is a receiver.

### [Lunch.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/Lunch.java)

**package** com.cakes;

**public** **class** Lunch {

**public** **void** makeLunch() {

System.out.println(**"Lunch is being made"**);

}

}

Dinner is also a receiver.

### [Dinner.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/Dinner.java)

**package** com.cakes;

**public** **class** Dinner {

**public** **void** makeDinner() {

System.out.println(**"Dinner is being made"**);

}

}

MealInvoker is the invoker class. It contains a reference to the Command to invoke. Its invoke() method calls the execute() method of the Command.

### [MealInvoker.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/MealInvoker.java)

**package** com.cakes;

**public** **class** MealInvoker {

Command command;

**public** MealInvoker(Command command) {

**this**.command = command;

}

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** invoke() {

command.execute();

}

}

The Demo class demonstrates the command pattern. It instantiates a Lunch (receiver) object and creates a LunchCommand (concrete command) with the Lunch. The LunchCommand is referenced by a Command interface reference. Next, we perform the same procedure on the Dinner and DinnerCommand objects. After this, we create a MealInvoker object with lunchCommand, and we call the invoke() method of mealInvoker. After this, we set mealInvoker's command to dinnerCommand, and once again call invoke() on mealInvoker.

### [Demo.java](http://www.avajava.com/tutorials/design-patterns/command-pattern/Demo.java)

**package** com.cakes;

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

Lunch lunch = **new** Lunch(); // receiver

Command lunchCommand = **new** LunchCommand(lunch); // concrete command

Dinner dinner = **new** Dinner(); // receiver

Command dinnerCommand = **new** DinnerCommand(dinner); // concrete command

MealInvoker mealInvoker = **new** MealInvoker(lunchCommand); // invoker

mealInvoker.invoke();

mealInvoker.setCommand(dinnerCommand);

mealInvoker.invoke();

}

}

The console output of the execution of Demo is shown here.

### [Console Output](http://www.avajava.com/tutorials/design-patterns/command-pattern/console.txt)

Lunch is being made

Dinner is being made

As you can see, the invoker invokes a command, but has no direct knowledge of the action being performed by the receiver.