<JSTORM>

Command패턴의 기초

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

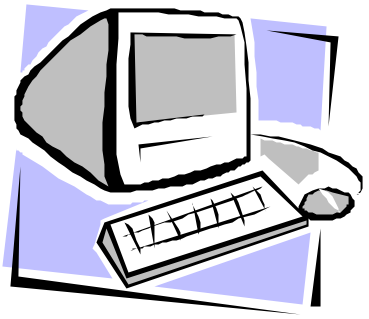
Document Information

Document title:	Command패턴의 기초
Document file name:	FundamentalOfCommandPattern_calmglow_pattern_jstorm_1.0_final.doc
Revision number:	<1.0>
Issued by:	<최진호> csecau@orgio.net
Issue Date:	<2001/07/16>
Status:	Final

Content Information

Audience	자바 초급
Abstract	Command패턴의 기초적인 이해를 돕는 문서
Reference	http://www.javaworld.com/javaworld/javatips/jw-javatip68_p.html 자바월드 문서 번역
Benchmark information	

Table of Contents



Command패턴의 기초

종종 클라이언트로부터의 요청을 받는 수신측이나 요청에 대한 처리방법에 대해 모호하여 그러한 처리를 도맡아 처리하는 방법이 필요할 경우가 있다. 절차식 언어에서는 주로 Callback방식에 의해 상호간 메시지 교류를 하였다. 즉, 필요로 하는 행동을 취하는 메소드의 포인터를 등록시키는 방법이 주로 쓰인 방식이었다. 본 문서에서는 자바에서 Command Pattern을 이용하여 요청에 대한 처리 방식에 대한 객체지향적인 접근을 시도해보겠다.

디자인패턴은 단지 객체지향 프로젝트의 설계 효율을 높일뿐만 아니라 소프트웨어의 품질과 개발팀의 생산성을 향상시켜준다. Command 패턴은 Sender와 Receiver간의 완전한 분리를 제공하여 준다. (Sender는 작업 실행을 요청하는 객체이고 Receiver는 작업 실행을 요청받는 객체이다. 이러한 완전한 분리로 Sender는 Receiver의 인터페이스에 대해 알 필요가 없다.) 여기서 요청(Request)은 실행될 명령을 말한다. Command 패턴은 이 요청(Request)이 언제 어떻게 자기 임무를 완수할 지를 담당하고 있다. 그래서 Command 패턴은 유연함과 확장성을 가져다준다.

C같은 프로그래밍 언어에서는 함수 포인터를 종종 방대해지는 Switch문의 복잡도를 줄이는 데 사용된다. 자바에서는 함수 포인터라는 것이 없지만 콜백을 Command패턴을 이용해서 구현할 수 있다. 아래 TestCommand.java를 통해 그 예를 볼 수 있다.

기타 다른 언어로 개발하는 이들은 습관적으로 함수 포인터를 사용하였기 때문에 같은 방법으로 Reflection API의 Method객체를 사용한다. 본인은 이러한 방법을 아주 싫어한다.
우리가 만드는 프로그램을 좀 더 쉽게 디버그하고 유지보수하고 싶다면 제발 그러한 Method객체를 사용하지 말고 인터페이스를 만들어서 필요로 하는 액션을 취하는 것을 그 안에서 구현하길 바란다.

따라서 본인은 본 문서에서 함수 포인터를 구현하기 위한 바인딩 구조와 자바의 동적 로딩을 같이 묶어 Command패턴을 설명하려 한다.
아마도 이제 우리는 이제까지의 짜증나던 switch 문을 제거하고 Command 패턴을 이용한

어플리케이션을 만듦으로서 훨씬 동적이고 유연한 시스템을 만들 수 있을 것이다. TestTransactionCommand.java라 불리는 두번째 소스코드를 통해서 좀 더 자세한 부분을 볼 수 있을 것이다.

Command 패턴은 요청(Request)자체를 객체화 하였다. 이 요청 객체는 다른 객체들 처럼 저장되거나 전달될 수 있다. 이 패턴의 핵심은 바로 실행하고자하는 행동을 정의하고 선언한 Command 인터페이스이다. 최대한 간단하게 구성하자면 이 인터페이스는 추상 Method execute 메소드를 가질 것이다.
각각의 execute()메소드를 구현한 Command 클래스들은 Receiver가 취할 특수한 행동을 가지고 있다.즉, Receiver는 요청에 대해 실행하기 위해 필요한 정보를 알고 있는 것이다.

이제 전등을 켜고 끄는 Switch을 프로그램화 한 예제를 통해 Command객체의 사용법의 간단한 예를 살펴보겠다. 그림 1은 Command 패턴을 구현한 Switch프로그램의 구조도이다. 이것의 인터페이스 안에는 flipUp()과 flipDown() 메소드를 가지고 있다. Switch클래스는 Command 인터페이스 내의 execute 메소드를 실행하기 때문에 invoker라고 불린다.

Command 인터페이스를 상속받아 구현된 LightOnCommand은 Command 인터페이스의 execute메소드를 구현하여 실제 스위치를 켰을 때 행해지는 행동을 구현한 클래스이다. 이 클래스는 마치 adapter처럼 행동한다.
이 Command 객체는 Invoker와 Receiver간의 서로 다른 인터페이스를 연결시켜주는 Adapter처럼 행동하는 것이다.

클라이언트는 Invoker와 Receiver, 그리고 구현된 Command 객체를 초기화하는 일을 담당한다.

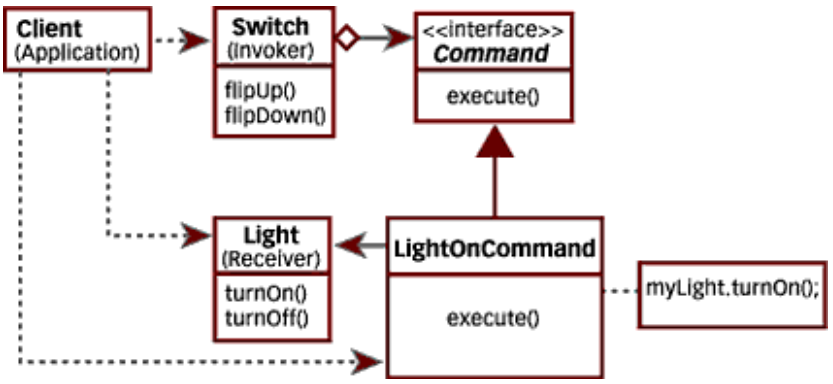


그림 1

그림 2는 객체간의 상호 작용을 보여주는 Sequence diagram이다. 여기서 우리는 Command가 Invoker와 Receiver간의 분리를 돕고 있는 것을 볼 수 있을 것이다.(또한 요청(Request)을 실행하는 것 또한...) 클라이언트는 Receiver객체를 인자로 줌으로써 적절한 Command 객체를 생성한 후, 그 Command 객체를 Invoker에 저장한다. 이제 나중에 Invoker는 어느 시점에서 필요한 Action을 해야할 경우 그 Action을 처리할 방법을 알고 있는

Command 객체를 실행할 것이다.

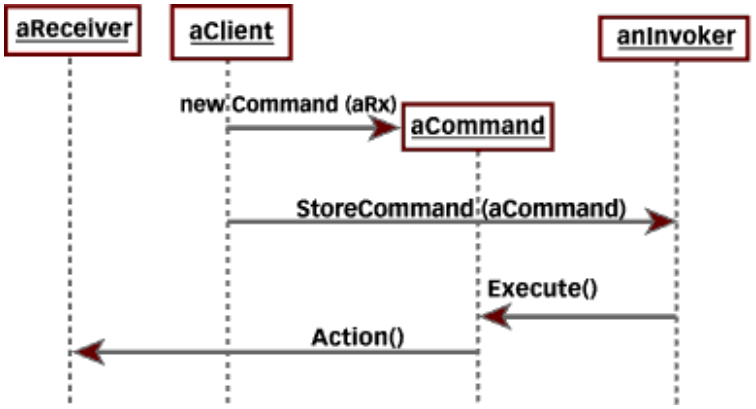


그림 2

클라이언트는 단지 Command 객체를 만들고 Receiver를 설정한다. Invoker객체인 Switch는 Command 객체를 저장한다. 그리고나서 Invoker는 Command객체의 execute 메소드를 호출함으로써 요청(Request)을 해결한다. Command객체는 Invoker의 요청을 처리하기 위해서 자기가 가지고 있는 Receiver의 메소드를 실행한다.

여기서 중요한 점은 구현된 Command 객체가 등록되는 곳은 Invoker이고 Invoker는 Command를 실행하여 Receiver의 명령을 간접적으로 실행하게 된다는 점이다.

Command 패턴 예제 코드

이제 Command패턴을 통해 만들어진 CallBack 구조를 간단한 코드를 통해 살펴보자

자, 선풍기와 전등이 있다. 우리의 목적은 그 둘을 작동시키는 Switch를 개발하는 것이다. 선풍기와 전등은 서로 다른 인터페이스를 가지고 있지만 Switch는 Receiver 인터페이스에 독립적이어야 하며, Switch는 Receiver의 인터페이스에 대한 정보를 전혀 가지고 있지 못하다. 당연히 전등과 연결된 Switch는 선풍기와 연결된 Switch와는 전혀 다르다. 여기서 Command 클래스는 이러한 작업을 위해 추상화와 인터페이스를 제공해준다.

Switch의 생성자가 실행될 때 적절한 Command 객체를 인자로 준다면 Command객체는 Switch의 변수로 저장된다. flipUp()과 flipDown()메소드가 불려지면 Switch는 단지 자기가 가지고 있는 Command의 execute()메소드를 실행할 것이다.

TestCommand.java

```
class Fan {
    public void startRotate() {
        System.out.println("Fan is rotating");
    }
```

```
        public void stopRotate() {
                System.out.println("Fan is not rotating");
        }
}
class Light {

        public void turnOn( ) {
                System.out.println("Light is on ");
        }
        public void turnOff( ) {
                System.out.println("Light is off");
        }
}
class Switch {
        private Command UpCommand, DownCommand;

        public Switch( Command Up, Command Down) {
                UpCommand = Up; // concrete Command registers itself with the
invoker
                DownCommand = Down;
        }
        void flipUp( ) { // invoker calls back concrete Command, which executes
the Command on the receiver
                        UpCommand . execute ( ) ;
        }
        void flipDown( ) {
                        DownCommand . execute ( );
        }
}

class LightOnCommand implements Command {

        private Light myLight;

        public LightOnCommand ( Light L) {
                myLight = L;
        }
        public void execute( ) {
                myLight . turnOn( );
        }

}
class LightOffCommand implements Command {
```

```
        private Light myLight;

        public LightOffCommand ( Light L) {
            myLight = L;
        }
        public void execute( ) {
            myLight . turnOff( );
        }

}
class FanOnCommand implements Command {

        private Fan myFan;

        public FanOnCommand ( Fan F) {
            myFan = F;
        }
        public void execute( ) {
            myFan . startRotate( );
        }

}
class FanOffCommand implements Command {

        private Fan myFan;

        public FanOffCommand ( Fan F) {
            myFan = F;
        }
        public void execute( ) {
            myFan . stopRotate( );
        }

}

public class TestCommand {
        public static void main(String[] args) {
            Light testLight = new Light( );
            LightOnCommand testLOC = new LightOnCommand(testLight);
            LightOffCommand testLFC = new LightOffCommand(testLight);
            Switch testSwitch = new Switch( testLOC,testLFC);
            testSwitch.flipUp( );
```

```
                testSwitch.flipDown( );

                Fan testFan = new Fan( );
                FanOnCommand foc = new FanOnCommand(testFan);
                FanOffCommand ffc = new FanOffCommand(testFan);
                Switch ts = new Switch( foc,ffc);
                ts.flipUp( );
                ts.flipDown( );

        }
}
```

Command.java

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

위 예제에서 명심할 것은 Command 패턴은 객체 자신이 실행해야할 정보를 완전히 분리하여 준다는 점이다. 이것은 상당한 유연함을 제공한다.

Transaction 구현을 위한 Command 패턴

Command패턴은 Action패턴 혹은 Transaction패턴으로도 알려져있다. TCP/IP소켓을 통해 클라이언트들의 요청을 처리하는 서버를 생각해보자. 이러한 Transaction은 하나의 Command로 구성되어있다.

대부분의 개발자들은 각 Command에 대해 switch문을 이용해서 case별로 처리할 것이다. 코드안의 switch문은 객체지향 프로젝트의 아주 나쁜 디자인에 속한다. 각 Command들은 객체지향적인 방법으로 Transaction을 지원할 수 있도록 사용될 수 있다.

아래 TestTransactionCommand.java 프로그램의 클라이언트 부분에서는 모든 요청들이 TransactionCommand객체 안에 보호(Encapsulated)되어있다. 클라이언트에 의해 TransactionCommand 가 생성되고 적절한 CommandManager에 등록된다.
큐에 들어간 요청(Request)들은 runCommands()메소드에 의해서 임의의 시간에 실행되어 유연성을 제공한다.
또한 여러 Command들을 혼합하여 새로운 Command들의 집합을 만들 수 있기도 하다. 본 예제는 CommandArgument, CommandReceiver 그리고 CommandManager클래스 및 AddCommand나 SubtractCommand 같은 TransactionCommand의 자식클래스로 구성되어 있다. 각각의 클래스들은 다음과 같은 역할을 한다.

- CommandArgument는 각 명령들의 인자를 저장하고 있는 helper클래스이다. 크고 다양한 종류의 인자들을 넘겨주는 작업을 간단하게 하기 위해 재작성될 수 있다.
- CommandReceiver는 Singleton패턴으로 구현되었는데 모든 명령들을 처리하는 메소드들을 구현하였다.
- CommandManager는 Invoker이며 처음 예제에서 보았던 Switch의 역할을 한다. myCommand라는 객체에 TransactionCommand객체를 저장하고 있다가 runCommand()메소드가 호출되면 myCommand의 execute()를 실행한다.

자바에서는 클래스의 이름으로 그 클래스를 동적으로 생성할 수 있다. TransactionCommand클래스의 execute()메소드를 보면 클래스이름으로 실행중에 객체를 만드는 부분을 볼 수 있다. 여기서 command 이름은 언제나 뒷부분에 "Command"문자열을 지니고 있다. 따라서 실제 Command클래스를 Command이름과 "Command"문자열을 결합함으로써 객체를 동적으로 만든다.

```
//TestTransactionCommand.java
import java.util.*;

final class CommandReceiver {

  private int[] c;
  private CommandArgument a;

    private CommandReceiver(){
      c = new int[2];
    }

     private static CommandReceiver cr = new CommandReceiver();

    public static CommandReceiver getHandle() {
  return cr;
    }

    public void setCommandArgument(CommandArgument a) {
  this.a = a;
    }

    public void methAdd() {
  c = a.getArguments();
      System.out.println("The result is " + (c[0]+c[1]));
    }

    public void methSubtract() {
```

```
  c = a.getArguments();
       System.out.println("The result is " + (c[0] -c[1]));
     }
}

 class CommandManager {

   private Command myCommand;

   public CommandManager(Command myCommand) {
     this.myCommand = myCommand ;
   }

   public void runCommands( ) {
         myCommand.execute();
   }

 }

class TransactionCommand implements Command {

  private CommandReceiver commandreceiver;
  private Vector commandnamelist,commandargumentlist;
  private String commandname;
  private CommandArgument commandargument;
  private Command command;

  public TransactionCommand () {
 this(null,null);
  }

  public TransactionCommand ( Vector commandnamelist, Vector
commandargumentlist){
    this.commandnamelist = commandnamelist;
    this.commandargumentlist = commandargumentlist;
    commandreceiver = CommandReceiver.getHandle();
  }

  public void execute( ) {

    for (int i = 0; i < commandnamelist.size(); i++) {

      commandname = (String)(commandnamelist.get(i));
```

```
        commandargument = (CommandArgument)((commandargumentlist.get(i)));
        commandreceiver.setCommandArgument(commandargument);
        String classname = commandname + "Command";

          try {
            Class cls = Class.forName(classname);
            command = (Command) cls.newInstance();
          }
          catch (Throwable e) {
                  System.err.println(e);
          }
        command.execute();
      }
    }
  }

 class AddCommand extends TransactionCommand {
   private CommandReceiver cr;

   public AddCommand () {
      cr = CommandReceiver.getHandle();
   }

   public void execute( ) {
     cr.methAdd();
   }
 }

 class SubtractCommand extends TransactionCommand {
   private CommandReceiver cr;

   public SubtractCommand () {
      cr = CommandReceiver.getHandle();
   }

   public void execute( ) {
     cr.methSubtract();
   }
 }


 class CommandA rgument {
   private int[] args;
```

```
  CommandArgument() {
    args = new int[2];
  }
  public int[] getArguments() {
return args;
  }
  public void setArgument(int i1, int i2) {
        args[0] = i1; args[1] = i2;
  }
}

public class TestTransactionCommand {
  private Vector clist,alist;

  public TestTransactionCommand() {
 clist = new Vector();
      alist = new Vector();
  }

  public void clearBuffer(Vector c, Vector a) {
 clist.removeAll(c);
      alist.removeAll(a);
  }

  public Vector getClist() {
 return clist;
  }

  public Vector getAlist() {
 return alist;
  }

   public static void main(String[] args) {
      CommandArgument ca,ca2;

    TestTransactionCommand t = new TestTransactionCommand();

    ca = new CommandArgument();
    ca.setArgument(2,8);
    Vector myclist = t.getClist();
    Vector myalist = t.getAlist();
    myclist.addElement("Add"); myalist.addElement(ca);
```

```
        TransactionCommand tc = new TransactionCommand(myclist,myalist);
        CommandManager cm = new CommandManager(tc);
                cm.runCommands();

        t.clearBuffer(myclist,myalist);
        ca2 = new CommandArgument();
        ca2.setArgument(5,7);
        myclist = t.getClist();
        myalist = t.get Alist();
        myclist.addElement("Subtract"); myalist.addElement(ca2);
        myclist.addElement("Add"); myalist.addElement(ca2);

        TransactionCommand tc2 = new TransactionCommand(myclist,myalist);
        CommandManager cm2 = new CommandManager(tc2);
                cm2.runCommands();

    }
}
```

command와 그것에 해당하는 인자들은 하나의 리스트에 저장되고 TransactionCommand객체에 보호되며 TransactionCommand객체는 또한 CommandManager에 등록된다. 따라서 실행중의 어떤 시점이라도 CommandManager클래스의 runCommands()를 실행함으로써 명령들은 실행된다.

클라이언트의 코드를 볼 것같으면 어떤 고정된 TransactionCommand의 자식클래스에도 의존하지 않음을 알 수 있다. 이것은 구현이 아니라 인터페이스를 통해 프로그래밍을 했기 때문이다. 이러한 방식은 확장성을 가져다 준다. 새로운 command를 만들기 위해 단지 새로운 TransactionCommand 자식클래스를 만들고 CommandReceiver클래스 내부에 새로운 명령 처리를 위한 메소드를 구현하면 되는 것이다.