Command Design Pattern

APRIL 2, 2018 BY PANKAJ — 16 COMMENTS

Command Pattern is one of the Behavioral Design Pattern. Command design pattern is used to implement **loose coupling** in a request-response model.

Table of Contents [hide]

Command Pattern

In command pattern, the request is send to the `invoker` and invoker pass it to the encapsulated `command` object.

Command object passes the request to the appropriate method of `Receiver` to perform the specific action.

The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action.

Now when client program executes the action, it's processed based on the command and receiver object.

Command Design Pattern Example

We will look at a real life scenario where we can implement Command pattern. Let's say we want to provide a File System utility with methods to open, write and close file. This file system utility should support multiple operating systems such as Windows and Unix.

To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work.

Since we code in terms of interface in java, we can have `FileSystemReceiver` interface and it's implementation classes for different operating system flavors such as Windows, Unix, Solaris etc.

Command Pattern Receiver Classes

```
package com.journaldev.design.command;

public interface FileSystemReceiver {

        void openFile();
        void writeFile();
```

```
        void closeFile();
}
```

FileSystemReceiver interface defines the contract for the implementation classes. For simplicity, I am creating two flavors of receiver classes to work with Unix and Windows systems.

```
package com.journaldev.design.command;

public class UnixFileSystemReceiver implements FileSystemReceiver {

        @Override
        public void openFile() {
                System.out.println("Opening file in unix OS");
        }

        @Override
        public void writeFile() {
                System.out.println("Writing file in unix OS");
        }

        @Override
        public void closeFile() {
                System.out.println("Closing file in unix OS");
        }

}
```

```
package com.journaldev.design.command;

public class WindowsFileSystemReceiver implements FileSystemReceiver {

        @Override
        public void openFile() {
                System.out.println("Opening file in Windows OS");

        }

        @Override
        public void writeFile() {
                System.out.println("Writing file in Windows OS");
        }
```

```
        @Override
        public void closeFile() {
                System.out.println("Closing file in Windows OS");
        }

}
```

Did you noticed the Override annotation and if you wonder why it's used, please read java annotations and override annotation benefits.

Now that our receiver classes are ready, we can move to implement our Command classes.

Command Pattern Interface and Implementations

We can use interface or abstract class to create our base Command, it's a design decision and depends on your requirement.

We are going with interface because we don't have any default implementations.

```
package com.journaldev.design.command;

public interface Command {

        void execute();
}
```

Now we need to create implementations for all the different types of action performed by the receiver. Since we have three actions we will create three Command implementations. Each Command implementation will forward the request to the appropriate method of receiver.

```
package com.journaldev.design.command;

public class OpenFileCommand implements Command {

        private FileSystemReceiver fileSystem;

        public OpenFileCommand(FileSystemReceiver fs){
                this.fileSystem=fs;
```

```
        }
        @Override
        public void execute() {
                //open command is forwarding request to openFile method
                this.fileSystem.openFile();
        }

}
```

```
package com.journaldev.design.command;

public class CloseFileCommand implements Command {

        private FileSystemReceiver fileSystem;

        public CloseFileCommand(FileSystemReceiver fs){
                this.fileSystem=fs;
        }
        @Override
        public void execute() {
                this.fileSystem.closeFile();
        }

}
```

```
package com.journaldev.design.command;

public class WriteFileCommand implements Command {

        private FileSystemReceiver fileSystem;

        public WriteFileCommand(FileSystemReceiver fs){
                this.fileSystem=fs;
        }
        @Override
        public void execute() {
                this.fileSystem.writeFile();
        }

}
```

Now we have receiver and command implementations ready, so we can move to implement the invoker class.

Command Pattern Invoker Class

Invoker is a simple class that encapsulates the Command and passes the request to the command object to process it.

```
package com.journaldev.design.command;

public class FileInvoker {

	public Command command;

	public FileInvoker(Command c){
		this.command=c;
	}

	public void execute(){
		this.command.execute();
	}
}
```

Our file system utility implementation is ready and we can move to write a simple command pattern client program. But before that I will provide a utility method to create the appropriate `FileSystemReceiver` object.

Since we can use System class to get the operating system information, we will use this or else we can use Factory pattern to return appropriate type based on the input.

```
package com.journaldev.design.command;

public class FileSystemReceiverUtil {

	public static FileSystemReceiver getUnderlyingFileSystem(){
		String osName = System.getProperty("os.name");
		System.out.println("Underlying OS is:"+osName);
		if(osName.contains("Windows")){
			return new WindowsFileSystemReceiver();
		}else{
			return new UnixFileSystemReceiver();
		}
```

```
        }

}
```

Let's move now to create our command pattern example client program that will consume our file system utility.

```
package com.journaldev.design.command;

public class FileSystemClient {

        public static void main(String[] args) {
                //Creating the receiver object
                FileSystemReceiver fs =
FileSystemReceiverUtil.getUnderlyingFileSystem();

                //creating command and associating with receiver
                OpenFileCommand openFileCommand = new OpenFileCommand(fs);

                //Creating invoker and associating with Command
                FileInvoker file = new FileInvoker(openFileCommand);

                //perform action on invoker object
                file.execute();

                WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
                file = new FileInvoker(writeFileCommand);
                file.execute();
```

Notice that client is responsible to create the appropriate type of command object. For example if you want to write a file you are not supposed to create `CloseFileCommand` object.

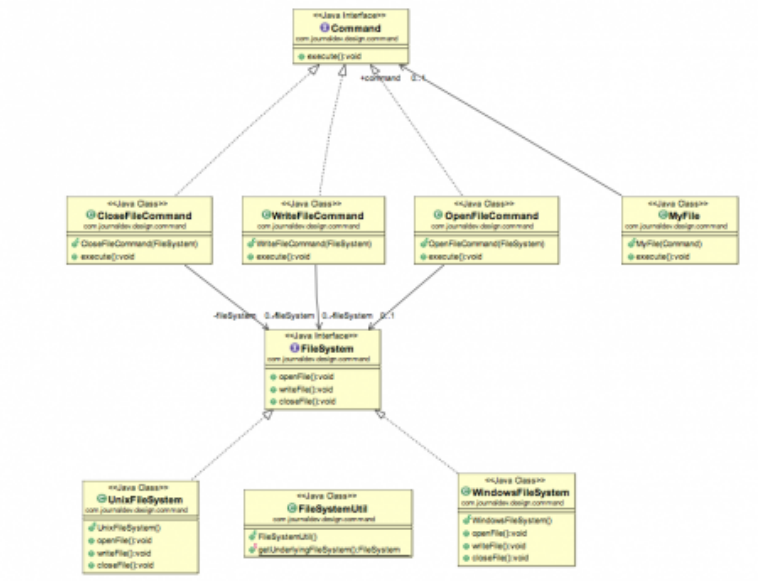
Client program is also responsible to attach receiver to the command and then command to the invoker class.

Output of the above command pattern example program is:

```
Underlying OS is:Mac OS X
Opening file in unix OS
Writing file in unix OS
Closing file in unix OS
```

Command Pattern Class Diagram

Here is the class diagram for our file system utility implementation.



Command Pattern Important Points

- Command is the core of command design pattern that defines the contract for implementation.
- Receiver implementation is separate from command implementation.
- Command implementation classes chose the method to invoke on receiver object, for every method in receiver there will be a command implementation. It works as a bridge between receiver and action methods.
- Invoker class just forward the request from client to the command object.
- Client is responsible to instantiate appropriate command and receiver implementation and then associate them together.
- Client is also responsible for instantiating invoker object and associating command object with it and execute the action method.
- Command design pattern is easily extendible, we can add new action methods in receivers and create new Command implementations without changing the client code.
- The drawback with Command design pattern is that the code gets huge and confusing with high number of action methods and because of so many associations.

Command Design Pattern JDK Example

Runnable interface (java.lang.Runnable) and Swing Action (javax.swing.Action) uses command pattern.

« PREVIOUS
Chain of Responsibility Design Pattern in Java

NEXT »
Interpreter Design Pattern in Java

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

Ravi says
JANUARY 22, 2017 AT 3:45 AM
Hi Pankaj

Nice post, I always like your comprehensive posts.
I just want to bring up couple points here, I don't see any point of having a Invoker class. Client is having references to command and receiver. So can't we just call command.execute()?
Could you give an example where we actually need Invoker?
Thanks,
Ravi.
Reply

Prakash K says
JULY 6, 2016 AT 3:42 AM
Its a very good explanation. Thank you.
It would be great to have the typo corrected for "lose coupling" to "loose coupling"
Thank you. Please keep up the good work.
Reply

Pankaj says
JULY 6, 2016 AT 9:13 AM
Thanks for catching the error, i have corrected it.
Reply

Ahamad says
OCTOBER 23, 2016 AT 9:36 AM
What ever in this site very easily and lay man language explained. Thank you so much.
Reply

Earl says
JULY 1, 2016 AT 5:23 AM
Sorry, but htis looks more like a bridge pattern than a command pattern. Also, for a request-response model, where is the response?
Reply

MalRaj says
JANUARY 22, 2016 AT 1:10 AM

Hi Pankaj,
Thanks for the fine example.
Reply

surjaj says
OCTOBER 24, 2015 AT 6:15 AM
why cant we implement
void openFile();
void writeFile();
void closeFile(); these method in execute() in command interface implementation.
Reply

Dusan says
OCTOBER 4, 2015 AT 11:44 AM
Why doesn't FileInvoker implement Command interface when it contains void execute() method?
Reply

Armando Flores says
JULY 24, 2015 AT 1:28 PM
I really like your example.
Just one thing, @Override annotations is not allowed for methods that implement an interface method.
Reply

md farooq says
JANUARY 3, 2015 AT 2:09 AM
nice explanation
Reply

You Know says
DECEMBER 26, 2014 AT 10:02 AM
This one was pretty good, Pankaj. I always knew you had it in you.
Reply

Vineet Kumar says
NOVEMBER 28, 2014 AT 10:21 PM
very good article
Reply

Hitesh says
SEPTEMBER 1, 2014 AT 9:19 AM
Very nice explanation.
Superb!
Amazing!
Masha'Allah!
Reply

Ajinder Singh says
AUGUST 27, 2014 AT 6:15 AM
Hi,
Very nice explanation.
Thanks
Reply

Chandra Shekhar says
JULY 16, 2014 AT 9:42 PM
Hi,
Pankaj very nicely explained. Thanks for sharing such a valuable post. I have gone through command pattern in few of the live projects and the implementation is with runnable or callable interface. Is there any particular feature of this pattern which makes it suitable for multithreaded system.
Regards,
Chandra Shekhar
Reply

Sunita Bansal says
APRIL 22, 2014 AT 10:11 PM
Sir, nice example of command pattern, can you tell us how the runnable interface is an example of command pattern.
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Structural Design Patterns

- Adapter
- Composite
- Proxy
- Flyweight
- Facade
- Bridge
- Decorator

Behavioral Design Patterns

- Template Method
- Mediator
- Chain of Responsibility
- Observer
- Strategy
- Command
- State
- Visitor
- Interpreter
- Iterator
- Memento

Miscellaneous Design Patterns

- Dependency Injection
- Thread Safety in Java Singleton

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial