**Command Pattern**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/command-design-pattern.jpg)

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](https://www.journaldev.com/1601/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](https://www.journaldev.com/721/java-annotations) and [override annotation benefits](https://www.journaldev.com/817/java-override-method-overriding).

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](https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java) 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](https://www.journaldev.com/904/java-system-getproperty), we will use this or else we can use [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) 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();

CloseFileCommand closeFileCommand = new CloseFileCommand(fs);

file = new FileInvoker(closeFileCommand);

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

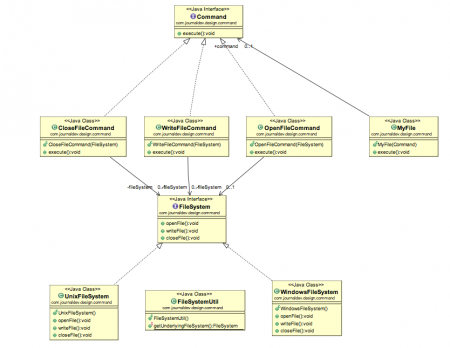
Opening file in unix OS

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.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Command-Pattern-Java-Class-Diagram.png)

**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](https://www.journaldev.com/1016/java-thread-example) (java.lang.Runnable) and Swing Action (javax.swing.Action) uses command pattern.