**Command pattern example code**

**generateSettlementObligationMessage gSom**

**DVP , CASH, STOCK,WRITEOFF AMOUT**

# Java Command Design Pattern example

Command design pattern decouples the object that invokes the operation, from the one that knows how to perform it.

Light and Fan objects are decoupled from Switch Class (Invoker)

### Intent

* Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
* Promote "invocation of a method on an object" to full object status
* An object-oriented callback

Summary: A [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29) “Command Pattern” example is demonstrated.

If you've ever seen Java source code that looks like this:

Based on SWift message type inrich the message

public void actionPerformed(ActionEvent e)

{

Object o = e.getSource();

if (o instanceof fileNewMenuItem)

doFileNewAction();

else if (o instanceof fileOpenMenuItem)

doFileOpenAction();

else if (o instanceof fileOpenRecentMenuItem)

doFileOpenRecentAction();

else if (o instanceof fileSaveMenuItem)

doFileSaveAction();

// and more ...

}

**The Command Design Pattern**

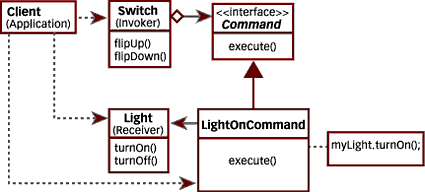


Figure 1 below shows the Switch -- an aggregation of Command objects.

Switch is called the invoker because it invokes the execute operation in the command interface.

The concrete command, LightOnCommand, implements the execute operation of the command interface.

The key idea here is that the concrete command registers itself (as a adapter) with the Invokerand the Invoker calls it back, executing the command on the Receiver.

Notice in the code example that the Command pattern completely decouples the object (FAN or Light) that invokes the operation -- (Switch ) -- from the ones having the knowledge to perform it -- Light and Fan. This gives us a lot of flexibility: the object issuing a request must know only how to issue it; it doesn't need to know how the request will be carried out.

The Command Pattern is a Design Pattern that does the following:

* Lets you encapsulate actions within Java classes, where each class has an "execute()" method which is declared in the Command interface the class implements.

Ex: –Command Interface with abstract execute() method

* Lets you build your Command objects into a "ready to run" state.

Ex: – LightOnCommand , LightOFFCommand ---Concrete class

* Pass those objects to an "invoker" class which makes a callback to the execute() method of each class at the appropriate time.

Ex: – Switch class

* Lets you build an invoker class that doesn't know anything about the logic each Command object contains; it simply knows that it should call the execute() method on an object when an event occurs.

Ex: – Switch class

* Lets you create macros in your applications by building a series of Command objects that have execute() -- and optionally undo() -- methods.

The Command Pattern has many uses in Java applications, but is commonly seen in:

* Handling actions for Java menu items and buttons.
* Providing support for macros (recording and playback of macros).
* Providing "undo" support.
* Java progress bars.
* Java wizards.

[**https://www.javaworld.com/article/2077569/core-java/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html**](https://www.javaworld.com/article/2077569/core-java/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html)

**Switch is like Invoker Class here**

Let's take a look at a simple example illustrating the callback mechanism achieved via the Command pattern.

The example shows a Fan and a Light. Our objective is to develop a Switch that can turn either object on or off. We see that the Fan and the Light have different interfaces, which means the Switch has to be independent of the Receiver interface or it has no knowledge of the code>Receiver's interface. To solve this problem, we need to parameterize each of the Switchs with the appropriate command. Obviously, the Switch connected to the Light will have a different command than the Switch connected to the Fan. The Command class has to be abstract or an interface for this to work.

When the constructor for a Switch is invoked, it is parameterized with the appropriate set of commands. The commands will be stored as private variables of the Switch.

When the flipUp() and flipDown() operations are called, they will simply make the appropriate command to execute( ). The Switch will have no idea what happens as a result of execute( ) being called.