WIKIPEDIA

Command pattern

In object-oriented programming, the **command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Four terms always associated with the command pattern are *command*, *receiver*, *invoker* and *client*. A *command* object knows about *receiver* and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver object to execute these methods is also stored in the command object by aggregation. The *receiver* then does the work when the execute() method in *command* is called. An *invoker* object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about the command *interface*. Invoker object(s), command objects and receiver objects are held by a *client* object, the *client* decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.

Contents

Overview

The Command [1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Command design pattern solve? [2]

- Coupling the invoker of a request to a particular request should be avoided. That is, hard-wired requests should be avoided.
- It should be possible to configure an object (that invokes a request) with a request.

Implementing (hard-wiring) a request directly into a class is inflexible because it couples the class to a particular request at compile-time, which makes it impossible to specify a request at run-time.

What solution does the Command design pattern describe?

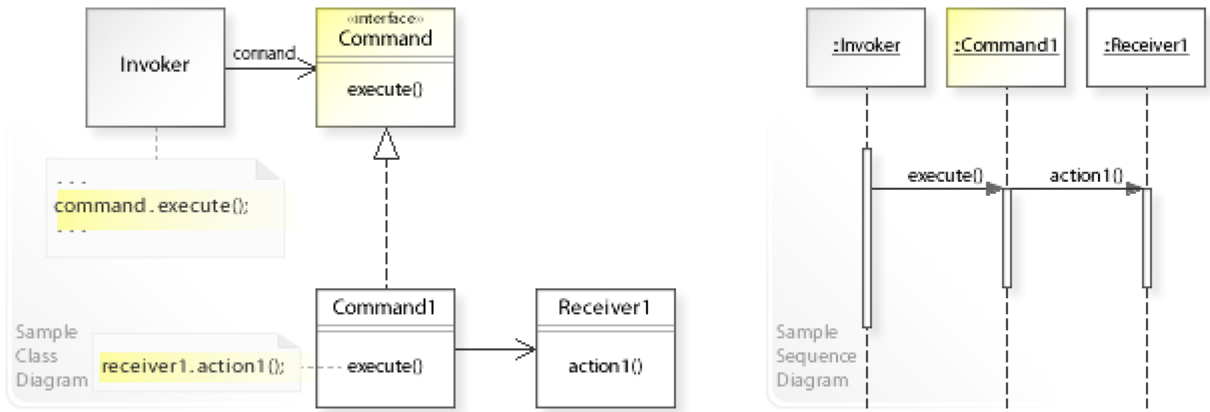
- Define separate (command) objects that encapsulate a request.
- A class delegates a request to a command object instead of implementing a particular request directly.

This enables one to configure a class with a command object that is used to perform a request. The class is no longer coupled to a particular request and has no knowledge (is independent) of how the request is carried out.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram

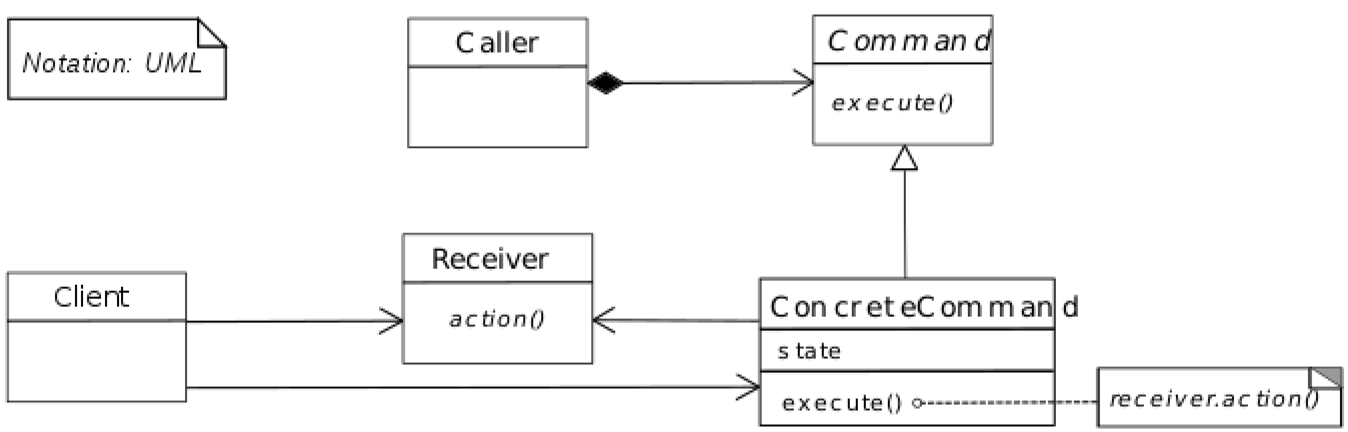


A sample UML class and sequence diagram for the Command design pattern. [3]

In the above UML class diagram, the `Invoker` class doesn't implement a request directly. Instead, `Invoker` refers to the `Command` interface to perform a request (`command.execute()`), which makes the `Invoker` independent of how the request is performed. The `Command1` class implements the `Command` interface by performing an action on a receiver (`receiver1.action1()`).

The UML sequence diagram shows the run-time interactions: The `Invoker` object calls `execute()` on a `Command1` object. `Command1` calls `action1()` on a `Receiver1` object, which performs the request.

UML class diagram



UML diagram of the command pattern

Uses

GUI buttons and menu items

In Swing and Borland Delphi programming, an `Action` (https://docs.oracle.com/javase/10/docs/api/javax/swing/Action.html) is a command object. In addition to the ability to perform the desired command, an `Action` may have an associated icon, keyboard shortcut, tooltip text, and so on. A toolbar button or menu item component may be completely initialized using only the `Action` object.

Macro recording

If all user actions are represented by command objects, a program can record a sequence of actions simply by keeping a list of the command objects as they are executed. It can then "play back" the same actions by executing the same command objects again in sequence. If the program embeds a scripting engine, each command object can implement a `toScript()` method, and user actions can then be easily recorded as scripts.

Mobile Code

Using languages such as Java where code can be streamed/slurped from one location to another via URLClassloaders and Codebases the commands can enable new behavior to be delivered to remote locations (EJB Command, Master Worker).

Multi-level undo

If all user actions in a program are implemented as command objects, the program can keep a stack of the most recently executed commands. When the user wants to undo a command, the program simply pops the most recent command object and executes its `undo()` method.

Networking

It is possible to send whole command objects across the network to be executed on the other machines, for example player actions in computer games.

Parallel Processing

Where the commands are written as tasks to a shared resource and executed by many threads in parallel (possibly on remote machines -this variant is often referred to as the Master/Worker pattern)

Progress bars

Suppose a program has a sequence of commands that it executes in order. If each command object has a `getEstimatedDuration()` method, the program can easily estimate the total duration. It can show a progress bar that meaningfully reflects how close the program is to completing all the tasks.

Thread pools

A typical, general-purpose thread pool class might have a public `addTask()` method that adds a work item to an internal queue of tasks waiting to be done. It maintains a pool of threads that execute commands from the queue. The items in the queue are command objects. Typically these objects implement a common interface such as `java.lang.Runnable` that allows the thread pool to execute the command even though the thread pool class itself was written without any knowledge of the specific tasks for which it would be used.

Transactional behavior

Similar to undo, a database engine or software installer may keep a list of operations that have been or will be performed. Should one of them fail, all others can be reversed or discarded (usually called *rollback*). For example, if two database tables that refer to each other must be updated, and the second update fails, the transaction can be rolled back, so that the first table does not now contain an invalid reference.

Wizards

Often a wizard presents several pages of configuration for a single action that happens only when the user clicks the "Finish" button on the last page. In these cases, a natural way to separate user interface code from application code is to implement the wizard using a command object. The command object is created when the wizard is first displayed. Each wizard page stores its GUI changes in the command object, so the object is populated as the user progresses. "Finish" simply triggers a call to `execute()`. This way, the command class will work.

Terminology

The terminology used to describe command pattern implementations is not consistent and can therefore be confusing. This is the result of ambiguity, the use of synonyms, and implementations that may obscure the original pattern by going well beyond it.

1. Ambiguity.
    1. The term **command** is ambiguous. For example, *move up, move up* may refer to a single (move up) command that should be executed twice, or it may refer to two commands, each of which happens to do the same thing (move up). If the former command is added twice to an undo stack, both items on the stack refer to the same command instance. This may be appropriate when a command can always be undone the same way (e.g. move down). Both the Gang of Four and the Java example below use this interpretation of the term *command*. On the other hand, if the latter commands are added to an undo stack, the stack refers to two separate objects. This may be appropriate when each object on the stack must contain information that allows the command to be undone. For example, to undo a *delete selection* command, the object may contain a copy of the deleted text so that it can be re-inserted, if the *delete selection* command must be undone. Note that using a separate object for each invocation of a command is also an example of the chain of responsibility pattern.
    2. The term **execute** is also ambiguous. It may refer to running the code identified by the command object's *execute* method. However, in Microsoft's Windows Presentation Foundation a command is considered to have been executed when the command's *execute* method has been invoked, but that does not necessarily mean that the application code has run. That occurs only after some further event processing.
2. Synonyms and homonyms.
    1. **Client, Source, Invoker**: the button, toolbar button, or menu item clicked, the shortcut key pressed by the user.
    2. **Command Object, Routed Command Object, Action Object**: a singleton object (e.g. there is only one CopyCommand object), which knows about shortcut keys, button images, command text, etc. related to the command. A source/invoker object calls the Command/Action object's execute/performAction method. The Command/Action object notifies the appropriate source/invoker objects when the availability of a command/action has changed. This allows buttons and menu items to become inactive (grayed out) when a command/action cannot be executed/performed.
    3. **Receiver, Target Object**: the object that is about to be copied, pasted, moved, etc. The receiver object owns the method that is called by the command's *execute* method. The receiver is typically also the target object. For example, if the receiver object is a *cursor* and the method is called *moveUp*, then one would expect that the cursor is the target of the moveUp action. On the other hand, if the code is defined by the command object itself, the target object will be a different object entirely.
    4. **Command Object, routed event arguments, event object**: the object that is passed from the source to the Command/Action object, to the Target object to the code that does the work. Each button click or shortcut key results in a new command/event object. Some implementations add more information to the command/event object as it is being passed from one object (e.g. CopyCommand) to another (e.g. document section). Other implementations put command/event objects in other event objects (like a box inside a bigger box) as they move along the line, to avoid naming conflicts. (See also chain of responsibility pattern.)
    5. **Handler, ExecutedRoutedEventHandler, method, function**: the actual code that does the copying, pasting, moving, etc. In some implementations the handler code is part of the command/action object. In other implementations the code is part of the Receiver/Target Object, and in yet other implementations the handler code is kept separate from the other objects.

6. **Command Manager, Undo Manager, Scheduler, Queue, Dispatcher, Invoker**: an object that puts command/event objects on an undo stack or redo stack, or that holds on to command/event objects until other objects are ready to act on them, or that routes the command/event objects to the appropriate receiver/target object or handler code.

3. Implementations that go well beyond the original command pattern.

   1. Microsoft's Windows Presentation Foundation (http://msdn.microsoft.com/en-us/library/ms752308.aspx) (WPF), introduces routed commands, which combine the command pattern with event processing. As a result, the command object no longer contains a reference to the target object nor a reference to the application code. Instead, invoking the command object's *execute* command results in a so-called *Executed Routed Event* that during the event's tunneling or bubbling may encounter a so-called *binding* object that identifies the target and the application code, which is executed at that point.

Example

Consider a "simple" switch. In this example we configure the Switch with two commands: to turn the light on and to turn the light off.

A benefit of this particular implementation of the command pattern is that the switch can be used with any device, not just a light. The Switch in the following C# implementation turns a light on and off, but the Switch's constructor is able to accept any subclasses of Command for its two parameters. For example, you could configure the Switch to start an engine.

C#

The following code is an implementation of Command pattern in C#.

```
using System;

namespace CommandPattern
{
    public interface ICommand
    {
        void Execute();
    }

    /* The Invoker class */
    public class Switch
    {
        ICommand _closedCommand;
        ICommand _openedCommand;

        public Switch(ICommand closedCommand, ICommand openedCommand)
        {
            this._closedCommand = closedCommand;
            this._openedCommand = openedCommand;
        }

        // Close the circuit / power on
        public void Close()
        {
           this._closedCommand.Execute();
```

```
    }

    // Open the circuit / power off
    public void Open()
    {
        this._openedCommand.Execute();
    }
}

/* An interface that defines actions that the receiver can perform */
public interface ISwitchable
{
    void PowerOn();
    void PowerOff();
}

/* The Receiver class */
public class Light : ISwitchable
{
    public void PowerOn()
    {
        Console.WriteLine("The light is on");
    }

    public void PowerOff()
    {
        Console.WriteLine("The light is off");
    }
}

/* The Command for turning off the device - ConcreteCommand #1 */
public class CloseSwitchCommand : ICommand
{
    private ISwitchable _switchable;

    public CloseSwitchCommand(ISwitchable switchable)
    {
        _switchable = switchable;
    }

    public void Execute()
    {
        _switchable.PowerOff();
    }
}

/* The Command for turning on the device - ConcreteCommand #2 */
public class OpenSwitchCommand : ICommand
{
    private ISwitchable _switchable;

    public OpenSwitchCommand(ISwitchable switchable)
    {
        _switchable = switchable;
    }
```

```
        public void Execute()
        {
            _switchable.PowerOn();
        }
    }

    /* The test class or client */
    internal class Program
    {
        public static void Main(string[] arguments)
        {
            string argument = arguments.Length > 0 ? arguments[0].ToUpper() : null;

            ISwitchable lamp = new Light();

            // Pass reference to the lamp instance to each command
            ICommand switchClose = new CloseSwitchCommand(lamp);
            ICommand switchOpen = new OpenSwitchCommand(lamp);

            // Pass reference to instances of the Command objects to the switch
            Switch @switch = new Switch(switchClose, switchOpen);

            if (argument == "ON")
            {
                // Switch (the Invoker) will invoke Execute() on the command object.
                @switch.Open();
            }
            else if (argument == "OFF")
            {
                // Switch (the Invoker) will invoke the Execute() on the command object.
                @switch.Close();
            }
            else
            {
                Console.WriteLine("Argument \"ON\" or \"OFF\" is required.");
            }
        }
    }
}
```

Java

In order to associate a name to an action to execute, the actions (switch on, switch off) on a lamp are wrapped into instances of the classes `SwitchOnCommand` and `SwitchOffCommand`, both implementing the interface `Command`.

```
import java.util.HashMap;

/** The Command interface */
interface Command {
  void execute();
}
```

```java
/** The Invoker class */
class Switch {
  private final HashMap<String, Command> commandMap = new HashMap<>();

  public void register(String commandName, Command command) {
    commandMap.put(commandName, command);
  }

  public void execute(String commandName) {
    Command command = commandMap.get(commandName);
    if (command == null) {
      throw new IllegalStateException("no command registered for " + commandName);
    }
    command.execute();
  }
}

/** The Receiver class */
class Light {
  public void turnOn() {
    System.out.println("The light is on");
  }

  public void turnOff() {
    System.out.println("The light is off");
  }
}

/** The Command for turning on the light - ConcreteCommand #1 */
class SwitchOnCommand implements Command {
  private final Light light;

  public SwitchOnCommand(Light light) {
    this.light = light;
  }

  @Override // Command
  public void execute() {
    light.turnOn();
  }
}

/** The Command for turning off the light - ConcreteCommand #2 */
class SwitchOffCommand implements Command {
  private final Light light;

  public SwitchOffCommand(Light light) {
    this.light = light;
  }

  @Override // Command
  public void execute() {
    light.turnOff();
  }
}
```

```java
public class CommandDemo {
  public static void main(final String[] arguments) {
    Light lamp = new Light();

    Command switchOn = new SwitchOnCommand(lamp);
    Command switchOff = new SwitchOffCommand(lamp);

    Switch mySwitch = new Switch();
    mySwitch.register("on", switchOn);
    mySwitch.register("off", switchOff);

    mySwitch.execute("on");
    mySwitch.execute("off");
  }
}
```

Using a functional interface

Since Java 8, it's not necessary to create the classes `SwitchOnCommand` and `SwitchOffCommand` anymore, one can use instead the :: operator as shown in the example below

```java
public class CommandDemo {
  public static void main(final String[] arguments) {
    Light lamp = new Light();

    Command switchOn = lamp::turnOn;
    Command switchOff = lamp::turnOff;

    Switch mySwitch = new Switch();
    mySwitch.register("on", switchOn);
    mySwitch.register("off", switchOff);

    mySwitch.execute("on");
    mySwitch.execute("off");
  }
}
```

Python

The following code is an implementation of Command pattern in Python.

```python
from collections import deque

class Switch(object):
    """The INVOKER class"""
    def __init__(self):
        self._history = deque()
```

```python
    @property
    def history(self):
        return self._history

    def execute(self, command):
        self._history.appendleft(command)
        command.execute()

class Command(object):
    """The COMMAND interface"""
    def __init__(self, obj):
        self._obj = obj

    def execute(self):
        raise NotImplementedError

class TurnOnCommand(Command):
    """The COMMAND for turning on the light"""
    def execute(self):
        self._obj.turn_on()

class TurnOffCommand(Command):
    """The COMMAND for turning off the light"""
    def execute(self):
        self._obj.turn_off()

class Light(object):
    """The RECEIVER class"""
    def turn_on(self):
        print("The light is on")

    def turn_off(self):
        print("The light is off")

class LightSwitchClient(object):
    """The CLIENT class"""
    def __init__(self):
        self._lamp = Light()
        self._switch = Switch()

    @property
    def switch(self):
        return self._switch

    def press(self, cmd):
        cmd = cmd.strip().upper()
        if cmd == "ON":
            self._switch.execute(TurnOnCommand(self._lamp))
        elif cmd == "OFF":
            self._switch.execute(TurnOffCommand(self._lamp))
        else:
            print("Argument 'ON' or 'OFF' is required.")

# Execute if this file is run as a script and not imported as a module
if __name__ == "__main__":
    light_switch = LightSwitchClient()
```

```
    print("Switch ON test.")
    light_switch.press("ON")
    print("Switch OFF test.")
    light_switch.press("OFF")
    print("Invalid Command test.")
    light_switch.press("****")

    print("Command history:")
    print(light_switch.switch.history)
```

Ruby

```ruby
# Invoker
class Switch
  attr_reader :history

  def execute(cmd)
    @history ||= []
    @history << cmd.execute
  end
end

# Command Interface
class Command
  attr_reader :obj

  def initialize(obj)
    @obj = obj
  end

  def execute
    raise NotImplementedError
  end
end

# Command for turning on
class TurnOnCommand < Command
  def execute
    obj.turn_on
  end
end

# Command for turning off
class TurnOffCommand < Command
  def execute
    obj.turn_off
  end
end

# Receiver
class Light
  def turn_on
```

```
    'the light is on'
  end

  def turn_off
    'the light is off'
  end
end

# Client
class LightSwitchClient
  attr_reader :switch

  def initialize
    @lamp = Light.new
    @switch = Switch.new
  end

  def switch_for(cmd)
    case cmd
    when 'on'  then @switch.execute(TurnOnCommand.new(@lamp))
    when 'off' then @switch.execute(TurnOffCommand.new(@lamp))
    else puts 'Sorry, I so sorry'
    end
  end
end

client = LightSwitchClient.new
client.switch_for('on')
client.switch_for('off')
client.switch.history #=> ['the light is on', 'the light is off']
```

Scala

```
/* The Command interface */
trait Command {
   def execute()
}

/* The Invoker class */
class Switch {
   private var history: List[Command] = Nil

   def storeAndExecute(cmd: Command) {
      cmd.execute()
      this.history :+= cmd
   }
}

/* The Receiver class */
class Light {
   def turnOn() = println("The light is on")
   def turnOff() = println("The light is off")
```

```
}

/* The Command for turning on the light - ConcreteCommand #1 */
class FlipUpCommand(theLight: Light) extends Command {
   def execute() = theLight.turnOn()
}

/* The Command for turning off the light - ConcreteCommand #2 */
class FlipDownCommand(theLight: Light) extends Command {
   def execute() = theLight.turnOff()
}

/* The test class or client */
object PressSwitch {
   def main(arguments: Array[String]) {
      val lamp = new Light()
      val switchUp = new FlipUpCommand(lamp)
      val switchDown = new FlipDownCommand(lamp)

      val s = new Switch()

      try {
         arguments(0).toUpperCase match {
            case "ON" => s.storeAndExecute(switchUp)
            case "OFF" => s.storeAndExecute(switchDown)
            case _ => println("Argument \"ON\" or \"OFF\" is required.")
         }
      } catch {
         case e: Exception => println("Arguments required.")
      }
   }
}
```

JavaScript

The following code is an implementation of Command pattern in JavaScript.

```
/* The Invoker function */
"use strict";

class Switch {
  constructor() {
    this._commands = [];
  }

  storeAndExecute(command) {
    this._commands.push(command);
    command.execute();
  }
}

class Light {
```

```
  turnOn() { console.log('turn on') }
  turnOff() { console.log('turn off') }
}

class FlipDownCommand {
  constructor(light) {
    this._light = light;
  }

  execute() {
    this._light.turnOff();
  }
}

class FlipUpCommand {
  constructor(light) {
    this._light = light;
  }

  execute() {
    this._light.turnOn();
  }
}

var light = new Light();
var switchUp = new FlipUpCommand(light);
var switchDown = new FlipDownCommand(light);
var s = new Switch();

s.storeAndExecute(switchUp);
s.storeAndExecute(switchDown);
```

CoffeeScript

The following code is an implementation of Command pattern in CoffeeScript.

```
# The Invoker function
class Switch
   _commands = []
   storeAndExecute: (command) ->
     _commands.push(command)
     command.execute()

#  The Receiver function
class Light
  turnOn: ->
    console.log ('turn on')
  turnOff: ->
    console.log ('turn off')

# The Command for turning on the light - ConcreteCommand #1
class FlipUpCommand
```

```
 constructor: (@light) ->

 execute: ->
   @light.turnOn()

# The Command for turning off the light - ConcreteCommand #2
class FlipDownCommand
 constructor: (@light) ->

 execute: ->
   @light.turnOff()

light = new Light()
switchUp = new FlipUpCommand(light)
switchDown = new FlipDownCommand(light)
s = new Switch()

s.storeAndExecute(switchUp)
s.storeAndExecute(switchDown)
```

C++

```
class ICommand
{
    public:
        virtual void Execute() = 0;
};

class Switcher
{
    private:
      std::vector<ICommand *> _commands;

    public:
        void StoreAndExecute(ICommand *command)
        {
            if (command){
                _commands.push_back(command);
                command->Execute();
            }
        }
};

class Light
{
    public:
        void TurnOn()
        {
            std::cout<<"The light is on."<<std::endl;
        }

        void TurnOff()
```

```
        {
            std::cout << "The light is off." << std::endl;
        }
};

/* The Command for turning on the light - ConcreteCommand #1 */
class FlipUpCommand : public ICommand
{
  private:
    Light *_light;

  public:
    FlipUpCommand(Light *light)
    {
        _light = light;
    }

    void Execute()
    {
        _light->TurnOn();
    }
};

/* The Command for turning off the light - ConcreteCommand #2 */
class FlipDownCommand : public ICommand
{
  private:
    Light *_light;

  public:
    FlipDownCommand(Light *light)
    {
        _light = light;
    }

    void Execute()
    {
        _light->TurnOff();
    }
};

int main()
{
    std::unique_ptr<Light> light = std::make_unique<Light>();
    std::unique_ptr<ICommand> switchOn (new FlipUpCommand(light.get()));
    std::unique_ptr<ICommand> switchDown (new FlipDownCommand(light.get()));

    std::unique_ptr<Switcher> switcher = std::make_unique<Switcher>();
    switcher->StoreAndExecute(switchOn.get());
    switcher->StoreAndExecute(switchDown.get());

    return 0;
}
```

See also

- Batch queue
- Closure
- Command queue
- Function object
- Job scheduler
- Model-view-controller
- Priority queue
- Software design pattern
- GoF - Design Patterns

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.
2. "The Command design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=b02&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
3. "The Command design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=b02&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- http://c2.com/cgi/wiki?CommandPattern
- http://www.javaworld.com/javaworld/javatips/jw-javatip68.html