Design patterns. Behavioural software design pattern

Command pattern

1. Design pattern description

The intent of the Command design pattern is to:

"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable opera-tions." [GoF]

A **request** is an operation that one object performs on another. From a more general point of view, a request is an arbitrary action to perform.

Problems that the command pattern solves

- Avoid the implementation of (hard-wire) a request (`receiver1.action1()`) directly within the class (`Invoker`) that invokes the request.

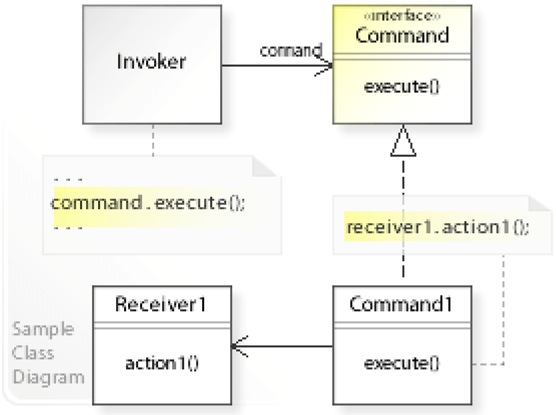
- Commiting (or coupling) the invoker of a request to a particular request at compile-time and making it impossible to specify a request at run-time. "When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and run-time." [GoF, p24]

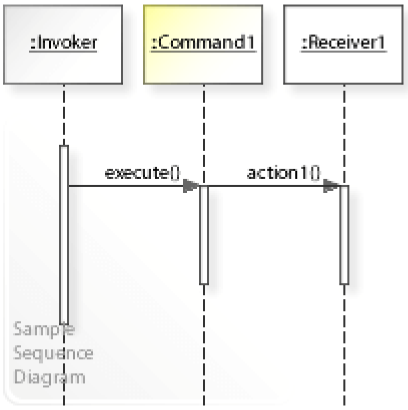
Key Relationships

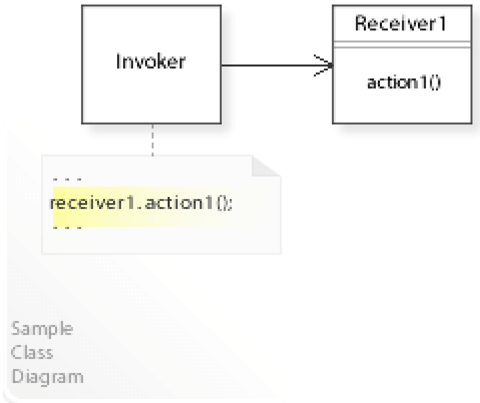
Strategy - Command

Strategy provides a way to configure an object with an algorithm at run-time instead of committing to an algorithm at compile-time.

Command provides a way to configure an object with a request at run-time instead of committing to a request at compile-time.







2. Design pattern example

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

    @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:")
```

```python
print(light_switch.switch.history)
```

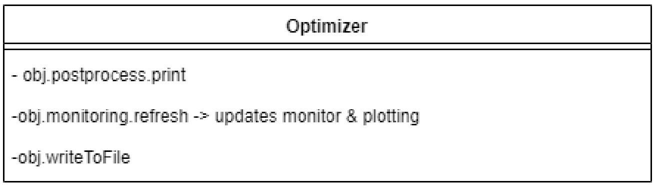
3. Existing pattern in FEM-MAT-OO?

Currently not implemented.

4. Design proposal in FEM-MAT-OO

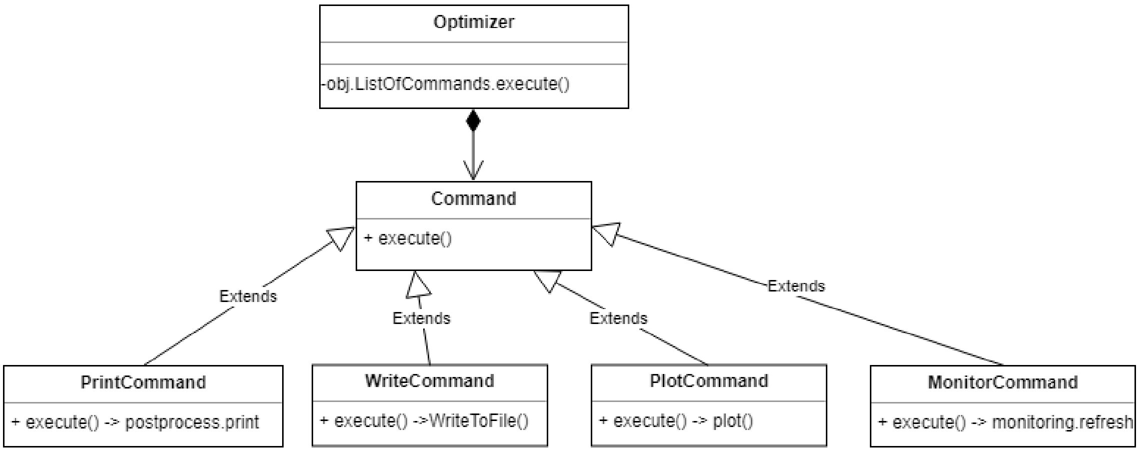
Optimizer and “finalize” processes calls. Currently hard wired to optimizer:

Current scheme:



Proposed:

Optimizer executes a list of commands, depending on which options are activated or not at the beginning of the problem.


Optimizer
-obj.ListOfCommands.execute()
Command
+ execute()
Extends
Extends
Extends
Extends
PrintCommand
+ execute() -> postprocess.print
WriteCommand
+ execute() ->WriteToFile()
PlotCommand
+ execute() -> plot()
MonitorCommand
+ execute() -> monitoring.refresh