# **Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics Software Engineering and Automation Department**

**Report**

**TMPS**

**LAB 3**

**Student**:   
Olednic Diana  
**Group**: FAF-192

Chișinău, 2022

# Topic: Behavioral Design Patterns

## Objectives:

**1. Study and understand the Behavioral Design Patterns.**

**2. As a continuation of the previous laboratory work, think about what communication between software entities might be involed in your system.**

**3. Implement some additional functionalities using behavioral design patterns.**

## Theoretical background:

    In software engineering, behavioral design patterns have the purpose of identifying common communication patterns between different software entities. By doing so, these patterns increase flexibility in carrying out this communication.

    Some examples from this category of design patterns are :

* Chain of Responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Observer
* Strategy

## Main tasks :

**1. By extending your project, implement at least 1 behavioral design pattern in your project:**

* The implemented design pattern should help to perform the tasks involved in your system.
* The behavioral DPs can be integrated into you functionalities alongside the structural ones.
* There should only be one client for the whole system.

**2. Keep your files grouped (into packages/directories) by their responsibilities (an example project structure):**

* client;
* domain;
* utilities;
* data(if applies);

**3. Document your work in a separate markdown file according to the requirements presented below (the structure can be extended of course):**

* Topic of the laboratory work.
* Author.
* Introduction/Theory/Motivation.
* Implementation & Explanation (you can include code snippets as well):
  + Indicate the location of the code snippet.
  + Emphasize the main idea and motivate the usage of the pattern.
* Results/Screenshots/Conclusions;

## Some Theory :

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication. Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it. Unlike the Creational and Structural patterns, which deal with the instantiation process and the blueprint of objects and classes, the central idea here is to concentrate on the way objects are interconnected.

* **Mediator Design Pattern**

Mediator helps in establishing loosely coupled communication between objects and helps in reducing the direct references to each other.

* **Observer Design Pattern**

Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the publish-subscribe pattern.

* **State Design Pattern**

A state allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

* **Memento Design Pattern**

Memento pattern is used to restore state of an object to a previous state. It is also known as snapshot pattern. The intent of memento pattern is to capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

* **Iterator Design Pattern**

An iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

* **Command Design Pattern**

Command pattern is a behavioral design pattern which is useful to abstract business logic into discrete actions i.e. commands. It motivates loose coupling.

* **Strategy Design Pattern**

Strategy design pattern is behavioral design pattern where we choose a specific implementation of algorithm or task in run time – out of multiple other implementations for same task.

* **Template Method Design Pattern**

Template method design pattern is widely accepted behavioral design pattern to enforce some sort of algorithm (fixed set of steps) in the context of programming. It defines the sequential steps to execute a multi-step algorithm and optionally can provide a default implementation as well (based on requirements).

* **Visitor Design Pattern**

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

* **Chain of Responsibility Design Pattern**

The main objective of this pattern is that it avoids coupling the sender of the request to the receiver, giving more than one object the opportunity to handle the request.

### Problem without using Chain of Responsibility Method

Imagine you are building a simple website that takes input strings and tells about the various properties of the strings such as is the string palindrome? is string upperCase? is string lowerCase? and many other properties too. After the complete planning, you decide that these checks for the input string should be performed sequentially. So, here the problem arises for the developer that he/she has to implement such an application that can decide on run-time which action should be performed next.

### Solution using Chain of Responsibility Method

Chain of Responsibility Method provides the solution for the above-described problem. It creates a separate **Abstract handler** which is used to handle the sequential operations which should be performed dynamically. For eg., we create four handlers named as **FirstConcreteHandler, SecondConcreteHandler, ThirdConcreteHandler**, and **Defaulthandler** and calls them sequentially from the user class.

My example shows the structure of the Chain of Responsibility pattern, namely what classes it consists of, what roles these classes perform and how they interact with each other.

**from** \_\_future\_\_ **import** annotations

**from** abc **import** ABC, abstractmethod

**from** typing **import** Any, Optional

**class** **Handler**(ABC):

"""

The Handler interface declares a method for constructing a chain of handlers. He also declares a method for executing the request. """

**@abstractmethod**

**def** **set\_next**(self, handler: Handler) -> Handler:

**pass**

**@abstractmethod**

**def** **handle**(self, request) -> Optional[str]:

**pass**

**class** **AbstractHandler**(Handler):

"""

The default chaining behavior can be implemented inside the base class

handler.

"""

\_next\_handler: Handler = **None**

**def** **set\_next**(self, handler: Handler) -> Handler:

self.\_next\_handler = ha

# Returning a handler from here will allow us to bind handlers with a simple # way, like this:

# monkey.set\_next(squirrel).set\_next(dog) return handler

@abstractmethod

def handle(self, request: Any) -> str:

if self.\_next\_handler:

return self.\_next\_handler.handle(request)

return None

"""

All Concrete Handlers either process the request or forward it

next handler in the chain.

"""

**class** **CatHandler**(AbstractHandler):

**def** **handle**(self, request: Any) -> str:

**if** request == "Fish":

**return** f"Cat: I'll eat the {request}"

**else**:

**return** super().handle(request)

**class** **DogHandler**(AbstractHandler):

**def** **handle**(self, request: Any) -> str:

**if** request == "MeatBall":

**return** f"dog: I'll eat the {request}"

**else**:

**return** super().handle(request)

**class** **RabbitHandler**(AbstractHandler):

**def** **handle**(self, request: Any) -> str:

**if** request == "Carrot":

**return** f"Rabbit: I'll eat the {request}"

**else**:

**return** super().handle(request)

**def** **client\_code**(handler: Handler) -> **None**:

"""

Typically, client code is designed to work with a single handler. In most cases, the client is not even aware that this handler is part of the chain. """

**for** food **in** ["MealBall", "Fish", "Cup of tea"]:

print(f"\nClient: Who wants a {food}?")

result = handler.handle(food)

**if** result:

print(f" {result}", end="")

**else**:

print(f" {food} was left untouched.", end="")

**if** \_\_name\_\_ == "\_\_main\_\_":

cat = CatHandler()

dog = DogHandler()

rabbit = RabbitHandler()

cat.set\_next(dog).set\_next(rabbit)

# The client should be able to send a request to any handler, not

# only the first one in the chain part of the chain.

print("Chain: Cat > Dog > Rabbit")

client\_code(cat)

print("\n")

print("Subchain: Dog > Rabbit")

client\_code(dog)

Conclusion

In laboratory 3 was important to understand Behavioral Design Patterns.

Generally speaking, design patterns are a very important field of computer science that is important in order to become a developper, as design patterns are reusable in multiple projects and provide solutions that can help define and build a good system architecture. I understood hot to orporate entities, directors, partners and that managers have a duty of care to the transport chain and are held accountable for the actions of people under their control.