# **Technical University of Moldova**

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

**Report**

**TMPS**

**LAB 2**

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

Chișinău, 2022

## Objectives:

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

**2. As a continuation of the previous laboratory work, think about the functionalities that your system will need to provide to the user.**

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

## Theoretical background:

    In software engineering, the Structural Design Patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to create a hierarchy of classes/abstractions, but the structural object patterns use composition which is generally a more flexible alternative to inheritance.

    Some examples of from this category of design patterns are:

* Adapter
* Bridge
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

## Main tasks:

**1. By extending your project, implement atleast 3 structural design patterns in your project:**

* The implemented design pattern should help to perform the tasks involved in your system.
* The object creation mechanisms/patterns can now be buried into the functionalities instead of using them into the client.
* 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
  + factories
  + builder
  + models
* 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 information about structural design patterns.**

Structural design patterns are **those that simplify the design of large object structures by identifying relationships between them**. They describe common ways of composing classes and objects so that they become repeatable as solutions.

**There are following 7 types of structural design patterns.**

* Adapter Pattern. Adapting an interface into another according to client expectation.
* Bridge Pattern. Separating abstraction (interface) from implementation.
* Composite Pattern. ...
* Decorator Pattern. ...
* Facade Pattern. ...
* Flyweight Pattern. ...
* proxy Pattern.

Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes.

Design Patterns are categorized mainly into three categories: **Creational Design Pattern, Structural Design Pattern, and Behavioral Design Pattern**.

What is incorrect about structural design? Explanation: The biggest drawback or problem is **a data flow diagram of structure design**.

The observer pattern is a **software design pattern in** which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

def add\_listener(obj, method\_name, listener):

# Get any existing listeners

listener\_attr = method\_name + '\_listeners'

listeners = getattr(obj, listener\_attr, None)

# If this is the first listener, then set up the method wrapper

if not listeners:

listeners = [listener]

setattr(obj, listener\_attr, listeners)

# Get the object's method

method = getattr(obj, method\_name)

@wraps(method)

def method\_wrapper(\*args, \*\*kwags):

method(\*args, \*\*kwags)

for l in listeners:

l(obj, \*args, \*\*kwags) # Listener also has object argument

# Replace the original method with the wrapper

setattr(obj, method\_name, method\_wrapper)

else:

# Event is already set up, so just add another listener

listeners.append(listener)

def remove\_listener(obj, method\_name, listener):

# Get any existing listeners

listener\_attr = method\_name + '\_listeners'

listeners = getattr(obj, listener\_attr, None)

if listeners:

# Remove the listener

next((listeners.pop(i)

for i, l in enumerate(listeners)

if l == listener),

None)

# If this was the last listener, then remove the method wrapper

if not listeners:

method = getattr(obj, method\_name)

delattr(obj, listener\_attr)

setattr(obj, method\_name, method.\_\_wrapped\_\_)

**These methods can then be used to add a listener to any class method. For example:**

class MyClass(object):

def \_\_init\_\_(self, prop):

self.prop = prop

def some\_method(self, num, string):

print('method:', num, string)

def listener\_method(obj, num, string):

print('listener:', num, string, obj.prop)

my = MyClass('my\_prop')

add\_listener(my, 'first\_method', listener\_method)

my.some\_method(42, 'with listener')

remove\_listener(my, 'first\_method', listener\_method)

my.some\_method(42, 'without listener')

And the output is:

method: 42 with listener

listener: 42 with listener my\_prop

method: 42 without listener

Conclusion

In laboratory 2 I put into practice Structural Design Patterns. After studying all the theory and putting it into practice in this laboratory work, I implemented my knowledge about Design Patterns, and namely about SDPs. I focus on how classes and objects can be composed into larger structures and at the same time to keep these structures flexible and efficient. Also, a challenge that I have faced was to respect the project structure, but from my point of view, I managed to do it not so bad.