**Модуль 1: Прийоми ООП.**

**Заняття 2: Абстрактні класи Патерни проектування**

**Design Patterns**

Паттерн проектування - це популярне рішення певної проблеми при проектуванні архітектури програм.

Паттерн не є конкретним кодом, а є загальною концепцією або прикладом вирішення тієї чи іншої проблеми, яку необхідно реалізувати відповідно до вимог вашої програми. Паттерн також є алгоритмом. Алгоритм — це чіткий набір дій, кулінарний рецепт приготування страви. Паттерн, за аналогією, — це креслення або схема будівлі, і конкретна програмна реалізація може відрізнятися від програми до програми.

Паттерни за призначенням поділяються на три основні групи:

* *Породжувальні патерни* турбуються про гнучке створення об'єктів без внесення до програми зайвих залежностей.
* *Структурні патерни* показують різноманітні способи побудови зв'язків між об'єктами.
* *Поведінкові патерни* піклуються про ефективну комунікацію між об'єктами.

**Абстрактні базові класи**

Абстрактні класи — це класи, призначені для наслідування, але уникають реалізації конкретних методів, залишаючи лише сигнатури методів, які повинні реалізовувати підкласи. Абстрактні класи активно використовуються при написанні патернів.

**Наївна реалізація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/abc#%D0%BD%D0%B0%D1%97%D0%B2%D0%BD%D0%B0-%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F)Найпростіший та найінтуїтивніший приклад створення абстрактного класу:

class MyABC:

def foo(self):

raise NotImplementedError()

def bar(self):

raise NotImplementedError()

У цьому прикладі ми створили клас MyABC у якого є методи foo та bar, які викликають виняток NotImplementedError, якщо ми спробуємо їх викликати.

Таким чином ми визначаємо інтерфейс — клас-батько, від якого потрібно наслідуватись і перевизначити методи foo та bar.

Клас інтерфейс або абстрактний базовий клас — це спосіб писати код, що самодокументується.

Однак очевидно, що за такого підходу у вас все ж таки є можливість припуститися помилки і не перевизначити якийсь із методів класу батька.

Виняток трапиться лише тоді, коли буде викликаний метод, який не перевизначили:

class MyABC:

def foo(self):

raise NotImplementedError()

def bar(self):

raise NotImplementedError()

class ActualMy(MyABC):

def foo(self):

print('foo')

a = ActualMy()

a.foo() # foo

a.bar() # raises NotImplementedError

Така особливість реалізації зводить нанівець всі переваги абстрактних класів. Цих недоліків позбавлений наступний спосіб реалізації абстрактних класів із використанням модуля abc.

**Реалізація з використанням abc**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/abc#%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%B7-%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F%D0%BC-abc)Для роботи з абстрактними базовими класами можна скористатися модулем abc, в якому є метаклас для створення абстрактних класів та декоратор abstractmethod, щоб позначити, який метод є обов'язковим для реалізації у спадкоємців.

from abc import abstractmethod, ABCMeta

class MyBaseClass(metaclass=ABCMeta):

@abstractmethod

def foo(self):

pass

@abstractmethod

def baz(self):

pass

class Child(MyBaseClass):

pass

c = Child() # TypeError: Can't instantiate abstract class Child with abstract methods baz, foo

Визначивши метаклас для абстрактного класу MyBaseClass та задекорувавши методи foo та baz, ми отримаємо виняток TypeError, як тільки спробуємо створити об'єкт класу спадкоємця, в якому якийсь із абстрактних методів не був визначений.

Такий підхід набагато надійніший.

Метакласи – це класи, екземпляри яких є класами. Щоб створити свій власний метаклас у Python, потрібно скористатися підкласом type, — стандартним метакласом у Python.

ExampleClass = type('ExampleClass', (object,) , {'some\_var': 1})

print(ExampleClass.\_\_name\_\_) # ExampleClass

print(ExampleClass.some\_var) # 1

Метакласи в Python потрібні для метапрограмування, коли вам, з якоїсь причини, необхідно динамічно генерувати нові класи безпосередньо під час виконання коду, залежно від якихось умов.

У 99.9% випадків вам це не знадобиться. Більше того, майже завжди є кращий і/або зручніший спосіб зробити те саме, але без використання метакласів.

Якщо ж використовувати метакласи незручно або може виникнути конфлікт, то можна досягти того самого без метакласів:

from abc import abstractmethod, ABC

class MyBaseClass(ABC):

@abstractmethod

def foo(self):

pass

@abstractmethod

def baz(self):

pass

class Child(MyBaseClass):

pass

c = Child() # TypeError: Can't instantiate abstract class Child with abstract methods baz, foo

**Коли варто використовувати абстрактні класи**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/abc#%D0%BA%D0%BE%D0%BB%D0%B8-%D0%B2%D0%B0%D1%80%D1%82%D0%BE-%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%BE%D0%B2%D1%83%D0%B2%D0%B0%D1%82%D0%B8-%D0%B0%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%96-%D0%BA%D0%BB%D0%B0%D1%81%D0%B8)Оскільки Python — мова з динамічною типізацією, то використовувати об'єкт того або іншого класу можна без огляду на його тип (клас, ланцюжок наслідування), аби потрібні атрибути були реалізовані.

Це робить розробку легшою на початкових етапах, поки коду не багато і задум програміста не складно розгадати. Але з розвитком проекту залежності стають складнішими, і ймовірність помилки зростає. Використання абстрактних базових класів робить код самодокументованим і дозволяє значно зменшити ймовірність припуститися помилки, забувши реалізувати потрібний метод або зробивши це неправильно.

Приклад використання абстрактних базових класів — це робота з різними інтерфейсами представлення або отримання інформації.

Наприклад, нам потрібно надати одну й ту саму інформацію у різних форматах: pdf, cls, txt, html.

Щоб мати можливість додавати нові формати без помилок, ми можемо створити абстрактний базовий клас створення документів:

from abc import ABC, abstractmethod

class Document(ABC)

@abstractmethod

def generate(self, data):

pass

Тепер будь-який клас, який відповідає за конкретний формат генерації документів, повинен наслідуватись від Document та реалізовувати generate.

З таким підходом додавати нові формати стане набагато легше.

**Породжувальні патерни**

**Абстрактна фабрика (Abstract Factory)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/creational#%D0%B0%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B0-%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B0-abstract-factory)Паттерн проектування, який дозволяє створювати сімейства пов'язаних об'єктів, не прив'язуючись до конкретних класів об'єктів, що створюються. Наприклад, для генерації сімейства звітів для відділу маркетингу вам потрібно згенерувати річний, квартальний, щомісячний звіти. При цьому є вимоги, що звіти потрібно генерувати у різних форматах: pdf, html, csv.

from abc import ABC, abstractmethod

class AbstractReport(ABC):

@abstractmethod

def create\_month\_report(self):

pass

@abstractmethod

def create\_quarter\_report(self):

pass

@abstractmethod

def create\_year\_report(self):

pass

class PdfReport(AbstractReport):

def create\_month\_report(self):

return PdfMonthReport()

def create\_quarter\_report(self):

return PdfQuarterReport()

def create\_year\_report(self):

return PdfYearReport()

class HtmlReport(AbstractReport):

def create\_month\_report(self):

return HtmlMonthReport()

def create\_quarter\_report(self):

return HtmlQuarterReport()

def create\_year\_report(self):

return HtmlYearReport()

class CsvReport(AbstractReport):

def create\_month\_report(self):

return CsvMonthReport()

def create\_quarter\_report(self):

return CsvQuarterReport()

def create\_year\_report(self):

return CsvYearReport()

Зверніть увагу, що абстрактна фабрика AbstractReport визначає, які періоди звітів повинні генерувати конкретні фабрики (щомісячний, квартальний, річний), а конкретні фабрики повертають безпосередньо класи звітів, що відповідають формату та потрібному періоду. Таким чином, коли знадобиться додати ще один період (піврічний наприклад), то ви зміните AbstractReport, і будь-яка сучасна IDE підкаже, які класи спадкоємці ще повинні реалізовувати необхідний метод. Це спрощує розробку, зменшує ймовірність помилки та робить код легшим для розуміння.

Часто може виникнути бажання збільшити зону відповідальності кожного класу та, замість безлічі дуже маленьких по суті CsvMonthReport, CsvQuarterReport, CsvYearReport, реалізувати один великий CsvReport, який буде вміти все, що треба. Це помилковий підхід, намагайтеся писати компактні та вузькоспеціалізовані класи, замість великих та складних.

**Фабричний метод (Factory Method)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/creational#%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D0%B8%D0%B9-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-factory-method)Паттерн проектування, який визначає загальний інтерфейс для створення об'єктів в суперкласі, дозволяючи підкласам змінювати тип об'єктів, що створюються.

Ви зробили для вашої компанії маркетингову SMS розсилку для споживачів. У якийсь момент компанія замовила мобільний застосунок, щоб представляти свій продукт. І виникла необхідність реалізувати маркетингові push розсилки для користувачів. Звучить чудово, але ось невдача, більшість вашого коду жорстко зав'язана на смс розсилці. І додавання push розсилок торкнеться більшої частини написаного вами коду. А поява email, telegram тощо розсилок змусить виконувати цю роботу знову і знову. Все це призводить до коду, що важко читати, наповненого умовними перевірками.

Рішенням є використання патерну фабричний метод. Він пропонує створювати об'єкти не напряму, а через виклик особливого фабричного методу. Щоб ця система працювала, всі об'єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виробляти об'єкти різних класів, що відповідають одному й тому самому інтерфейсу. Найкраще це зрозуміти, розглянувши конкретний приклад.

from abc import ABC, abstractmethod

class Creator(ABC):

@abstractmethod

def create(self):

pass

def send\_messages(self) -> str:

product = self.create()

result = product.sending()

return result

class SendingMessages(ABC):

@abstractmethod

def sending(self) -> str:

pass

class CreatorPush(Creator):

def create(self) -> SendingMessages:

return SendingPushMessages()

class CreatorSMS(Creator):

def create(self) -> SendingMessages:

return SendingSMSMessages()

class SendingPushMessages(SendingMessages):

def sending(self) -> str:

return "Push mailing has been completed"

class SendingSMSMessages(SendingMessages):

def sending(self) -> str:

return "SMS mailing has been completed"

def client\_code(creator: Creator) -> None:

print("We know nothing about the creator code that works")

result = creator.send\_messages()

print(f"Result: {result}")

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

print("The application performs Push mailing lists.")

client\_code(CreatorPush())

print("\n")

print("The application performs SMS mailing.")

client\_code(CreatorSMS())

Виведення під час виконання коду:

The application performs Push mailing lists.

We know nothing about the creator code that works

Result: Push mailing has been completed

The application performs SMS mailing.

We know nothing about the creator code that works

Result: SMS mailing has been completed

**Одинак (Singletone)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/creational#%D0%BE%D0%B4%D0%B8%D0%BD%D0%B0%D0%BA-singletone)Паттерн проектування, який гарантує, що клас має лише один екземпляр і надає до нього глобальну точку доступу.

Одинак вирішує дві проблеми:

1. Гарантує наявність єдиного екземпляра класу.
2. Надає глобальну точку доступу.

import random

class Singleton:

"""Classic singleton"""

\_\_instance = None

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

self.number = random.randint(1, 10)

def \_\_new\_\_(cls):

if cls.\_\_instance is None:

cls.\_\_instance = super().\_\_new\_\_(Singleton)

return cls.\_\_instance

class Regular:

"""Simple class to compare behavior"""

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.number = random.randint(1, 10)

def testing():

print("Singleton instances")

list\_singleton = [Singleton() for i in range(0, 5)]

for index, element in enumerate(list\_singleton):

print(f"Element: {index} number : {element.number}")

print("Instances of a regular class")

list\_regular = [Regular() for i in range(0, 5)]

for index, element in enumerate(list\_regular):

print(f"Element: {index} number : {element.number}")

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

testing()

Паттерн одинак гарантує, що жодний інший код не замінить створений екземпляр класу, тому ви завжди впевнені в наявності лише одного об'єкта-одинака. І як бачимо з прикладу, одинак завжди повертає 10, в той час як звичайний клас у нас виводить різні довільні числа.

Виведення:

Singleton instances

Element: 0 number : 10

Element: 1 number : 10

Element: 2 number : 10

Element: 3 number : 10

Element: 4 number : 10

Instances of a regular class

Element: 0 number : 9

Element: 1 number : 2

Element: 2 number : 10

Element: 3 number : 3

Element: 4 number : 6

Найчастіше застосовується для доступу до якогось спільного ресурсу, наприклад бази даних.

Запустіть для демонстрації живий приклад:

**Структурні патерни**

**Фасад (Facade)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/structurial#%D1%84%D0%B0%D1%81%D0%B0%D0%B4-facade)Паттерн проектування, який надає простий інтерфейс до складної системі класів, бібліотеки або фреймворку.

Паттерн вирішує проблему, коли бізнес-логіка ваших класів тісно починає переплітатися з деталями реалізації сторонніх бібліотек. Код стає заплутаним, а заплутаний код стає важче підтримувати. Для таких цілей і використовують патерн Фасад. Простими словами, всю складну логіку ми ховаємо за простим інтерфейсом.

Прикладом може бути замовлення в інтернет-магазині. У вас є фасад – це сайт, на якому ви робите замовлення. Від вас захована вся функціональність, ви не телефонуєте на склад, не виписуєте накладну тощо. Все просто – є фасад сайту, де мінімум необхідного функціоналу для роботи.

Якщо коротко підбити підсумок – ми використовуємо патерн Фасад, замість прямої роботи з об'єктами складної форми.

class FacadeNewsletter:

def \_\_init\_\_(self, users\_system, email\_system) -> None:

self.\_users\_system = users\_system

self.\_email\_system = email\_system

def sending(self) -> str:

users = self.\_users\_system.get\_users()

male, female = self.\_users\_system.separate\_users(users)

text\_for\_male = self.\_email\_system.get\_text\_email("male")

text\_for\_female = self.\_email\_system.get\_text\_email("female")

self.\_email\_system.send\_emails(male, text\_for\_male)

self.\_email\_system.send\_emails(female, text\_for\_female)

return "Done"

class UsersSystem:

def get\_users(self) -> list:

users = [

{

"name": "Allen Raymond",

"email": "nulla.ante@vestibul.co.uk",

"gender": "male",

},

{

"name": "Chaim Lewis",

"email": "dui.in@egetlacus.ca",

"gender": "male",

},

{

"name": "Kennedy Lane",

"email": "mattis.Cras@nonenimMauris.net",

"gender": "female",

},

{

"name": "Wylie Pope",

"email": "est@utquamvel.net",

"gender": "female",

},

]

return users

def separate\_users(self, users) -> tuple:

male = []

female = []

for person in users:

if person.get("gender", None) == "male":

male.append(person)

else:

female.append(person)

return male, female

class EmailSystem:

def get\_text\_email(self, gender) -> str:

text = "Default text"

if gender == "male":

text = "Male text email"

if gender == "female":

text = "Female text email"

return text

def send\_emails(self, users, text) -> str:

for person in users:

print(f"Send {person.get('name')} email: {text}")

return "Done"

def client\_code(newsletter) -> None:

print(newsletter.sending(), end="")

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

facade = FacadeNewsletter(UsersSystem(), EmailSystem())

client\_code(facade)

Виведення:

Send Allen Raymond email: Male text email

Send Chaim Lewis email: Male text email

Send Kennedy Lane email: Female text email

Send Wylie Pope email: Female text email

Done

**Адаптер**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/structurial#%D0%B0%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80)Патерн проектування, що дозволяє об'єктам з несумісними інтерфейсами працювати разом. З патерном Адаптер ми вже познайомилися, коли розглядали принцип SOLID – Dependency inversion.

У нашому випадку банк надавав інформацію у форматі JSON виду:

[

{

'ccy': 'EUR',

'base\_ccy': 'UAH',

'buy': '37.89060',

'sale': '39.06250'

},

{

'ccy': 'USD',

'base\_ccy': 'UAH',

'buy': '36.56860',

'sale': '37.45318'

}

]

Наша функція виведення даних pretty\_view працює зі словниками наступного вигляду:

{

'EUR': {

'buy': 37.8906,

'sale': 39.0625

},

'USD': {

'buy': 36.5686,

'sale': 37.45318

}

}

На допомогу нам приходить функція-адаптер data\_adapter, яка і перетворює дані з API в необхідний словник:

import requests

class RequestConnection:

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

self.request = request

def get\_json\_from\_url(self, url):

return self.request.get(url).json()

class ApiClient:

def \_\_init\_\_(self, fetch: RequestConnection):

self.fetch = fetch

def get\_data(self, url):

response = self.fetch.get\_json\_from\_url(url)

return response

def data\_adapter(data: dict):

return [{f"{el.get('ccy')}": {"buy": float(el.get('buy')), "sale": float(el.get('sale'))}} for el in data]

def pretty\_view(data):

pattern = '|{:^10}|{:^10}|{:^10}|'

print(pattern.format('currency', 'sale', 'buy'))

for el in data:

currency, \*\_ = el.keys()

buy = el.get(currency).get('buy')

sale = el.get(currency).get('sale')

print(pattern.format(currency, sale, buy))

if \_\_name\_\_ == '\_\_main\_\_':

api\_client = ApiClient(RequestConnection(requests))

data = api\_client.get\_data('https://api.privatbank.ua/p24api/pubinfo?exchange&coursid=11')

pretty\_view(data\_adapter(data))

За підсумком data\_adapter – це перекладач, який трансформує інтерфейс або дані одного об'єкта у такий вигляд, щоб він став зрозумілим іншому об'єкту.

ПІДКАЗКА

Пакет requests у прикладі вище є зовнішнім та вимагає попереднього встановлення

pip install requests

**Заступник (Proxy)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/structurial#%D0%B7%D0%B0%D1%81%D1%82%D1%83%D0%BF%D0%BD%D0%B8%D0%BA-proxy)Паттерн проектування, який дозволяє підставляти замість реальних об'єктів спеціальні об'єкти-замінники.

Абстрактний клас Request оголошує загальні операції як для класу RealRequest, так і для класу Proxy. Якщо клієнт працює з RealRequest, використовуючи цей інтерфейс, ви зможете передати йому Proxy, замість RealRequest.

Найпоширенішими сферами застосування патерну Proxy є ліниве завантаження, кешування, контроль доступу, ведення журналу тощо.

Клас Proxy може виконати одне з цих завдань, а потім, залежно від результату, передати виконання однойменному методу у пов'язаному об'єкті класу RealRequest.

Розглянемо приклад логування деякого запиту за допомогою патерну Proxy:

from abc import ABC, abstractmethod

from time import time, sleep

class Request(ABC):

@abstractmethod

def request(self) -> None:

pass

class RealRequest(Request):

def request(self) -> None:

print("RealRequest: Handling request.")

sleep(0.5)

class Proxy(Request):

def \_\_init\_\_(self, real\_request) -> None:

self.\_real\_request = real\_request

self.start = None

def request(self) -> None:

self.start = time()

self.\_real\_request.request()

self.log\_access()

def log\_access(self) -> None:

print(f"Proxy: Logging the time of request. {time() - self.start}")

def client\_code(subject) -> None:

subject.request()

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

proxy = Proxy(RealRequest())

client\_code(proxy)

Виведення:

RealRequest: Handling request.

Proxy: Logging the time of request.

Запустіть для демонстрації живий приклад:

**Поведінкові патерни**

Ці патерни вирішують завдання ефективної та безпечної взаємодії між об'єктами програми.

**Команда (Command)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/behavioral#%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-command)Паттерн проектування, який перетворює запити на об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

from abc import ABC, abstractmethod

class Command(ABC):

@abstractmethod

def execute(self) -> None:

pass

class CommandCreateXMLOrder(Command):

def \_\_init\_\_(self, receiver, text: str) -> None:

self.\_receiver = receiver

self.\_text = text

def execute(self) -> None:

self.\_receiver.createXMLOrder(self.\_text)

class CommandSendEmail(Command):

def \_\_init\_\_(self, receiver, html: str) -> None:

self.\_receiver = receiver

self.\_html = html

def execute(self) -> None:

self.\_receiver.send\_email(self.\_html)

class Receiver:

def createXMLOrder(self, text: str) -> None:

print(f"Create XML order: {text} ")

def send\_email(self, text: str) -> None:

print(f"Send email: {text} ")

class Invoker:

def \_\_init\_\_(self) -> None:

self.\_on\_order = None

self.\_on\_email = None

def set\_on\_order(self, command: Command):

self.\_on\_order = command

def set\_on\_email(self, command: Command):

self.\_on\_email = command

def generate\_general\_order(self) -> None:

self.\_on\_order.execute()

self.\_on\_email.execute()

def client():

invoker = Invoker()

invoker.set\_on\_order(CommandSendEmail(Receiver(), "Send email"))

invoker.set\_on\_email(CommandCreateXMLOrder(Receiver(), "Save report"))

invoker.generate\_general\_order()

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

client()

Виведення:

Send email: Send email

Create XML order: Save report

Відправник клас Invoker зберігає посилання на об'єкти команд, у нашому випадку self.\_on\_order та self.\_on\_email, і звертається до них, коли потрібно виконати якусь дію. Він працює з командами лише через їхній спільний інтерфейс. Він не знає, яку конкретно команду використовує, оскільки отримує готовий об'єкт команди від клієнта.

Команда — абстрактний клас Command, описує загальний для всіх конкретних команд інтерфейс. Зазвичай, це один метод для запуску команди.

Конкретні команди, класи CommandCreateXMLOrder та CommandSendEmail, реалізують різні запити, дотримуючись спільного інтерфейсу команд. Майже завжди команда передає виклик одержувачу об'єкту бізнес-логіки, клас Receiver.

Одержувач клас Receiver містить бізнес-логіку програми, це може бути будь-який об'єкт. Зазвичай, команди перенаправляють виклики одержувачам. Але іноді, щоб спростити програму, ви можете позбавитися одержувачів, перемістивши їх код в класи команд.

Клієнт, у нашому випадку функція client, створює об'єкти конкретних команд, передаючи в них усі необхідні параметри, а іноді й посилання на об'єкти одержувачів. Після цього клієнт конфігурує відправників створеними командами.

**Спостерігач (Observer)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-02/behavioral#%D1%81%D0%BF%D0%BE%D1%81%D1%82%D0%B5%D1%80%D1%96%D0%B3%D0%B0%D1%87-observer)Паттерн проектування, який створює механізм підписки, що дозволяє одним об'єктам стежити і реагувати на події, що відбуваються в інших об'єктах.

Уявіть, що ви хочете придбати якийсь унікальний товар в інтернет-магазині. Ви щодня заходите на сайт і перевіряєте, чи не з'явився товар. Ви витрачаєте час і не щасливі, що товару все немає. Можна підписатися на спам розсилку нових товарів у магазині. Але більшість товарів вам зовсім не потрібні, і шукати в розсилці, чи з'явилася потрібна вам річ, чи ні — те ще заняття.

Рішення — це паттерн Спостерігач. В інтернет-магазині з'являється кнопка повідомити мене, коли товар з'явиться. Ви натискаєте на неї, підписуєтеся на подію, і з появою товару магазин повідомляє вам, що товар з'явився. Фактично інтернет-магазин — це видавець, який володіє внутрішнім станом, поява товару якого цікава для підписників (покупців). Він містить механізм підписки — список підписників, а також методи підписки/відписки. Коли товар з'являється видавець сповіщає своїх підписників. Для цього видавець проходить списком підписників (всіх, хто натиснув кнопку) і викликає їх метод повідомлення, заданий в інтерфейсі підписника.

Давайте розглянемо конкретний приклад:

from abc import ABC, abstractmethod

class Publisher(ABC):

@abstractmethod

def attach(self, observer):

pass

@abstractmethod

def detach(self, observer):

pass

@abstractmethod

def notify(self):

pass

class PublisherMessages(Publisher):

\_observers = []

\_indicator = 0

def attach(self, observer):

self.\_observers.append(observer)

def detach(self, observer):

self.\_observers.remove(observer)

def notify(self):

for observer in self.\_observers:

observer.update(self)

def business\_logic\_execution(self):

print(f"Application logic is being executed. Indicator: {self.\_indicator}")

self.\_indicator += 1

self.notify()

class Observer(ABC):

@abstractmethod

def update(self, publisher):

pass

class ObserverA(Observer):

def update(self, publisher):

if publisher.\_indicator <= 3:

print("ObserverA: reacts to the indicator less than 2")

class ObserverB(Observer):

def update(self, publisher):

if publisher.\_indicator > 2:

print("ObserverB: reacts to the indicator greater than 2")

def client():

publisher = PublisherMessages()

observer\_a = ObserverA()

publisher.attach(observer\_a)

observer\_b = ObserverB()

publisher.attach(observer\_b)

publisher.business\_logic\_execution()

publisher.business\_logic\_execution()

publisher.detach(observer\_a)

publisher.business\_logic\_execution()

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

client()

Що відбувається у нас. Ми створюємо екземпляр Видавця

publisher = PublisherMessages()

Після цього ми створюємо екземпляри двох підписників і підписуємо їх на видавця

observer\_a = ObserverA()

publisher.attach(observer\_a)

observer\_b = ObserverB()

publisher.attach(observer\_b)

Починаємо виконувати метод business\_logic\_execution з якоюсь бізнес-логікою Видавця.

publisher.business\_logic\_execution()

publisher.business\_logic\_execution()

publisher.detach(observer\_a)

publisher.business\_logic\_execution()

І при кожному виконанні методу business\_logic\_execution виконується метод self.notify(), який проходить списком підписників та виконує у них метод update. Всередину методу update ми передаємо посилання на об'єкт видавця, щоб підписник міг стежити за властивістю \_indicator та реагувати відповідно.

Все це призводить до наступного виведення:

Application logic is being executed. Indicator: 0

ObserverA: reacts to the indicator less than 2

Application logic is being executed. Indicator: 1

ObserverA: reacts to the indicator less than 2

Application logic is being executed. Indicator: 2

ObserverB: reacts to the indicator greater than 2