

ММП 2025/2026

Викладач Канцедал Георгій Олегович

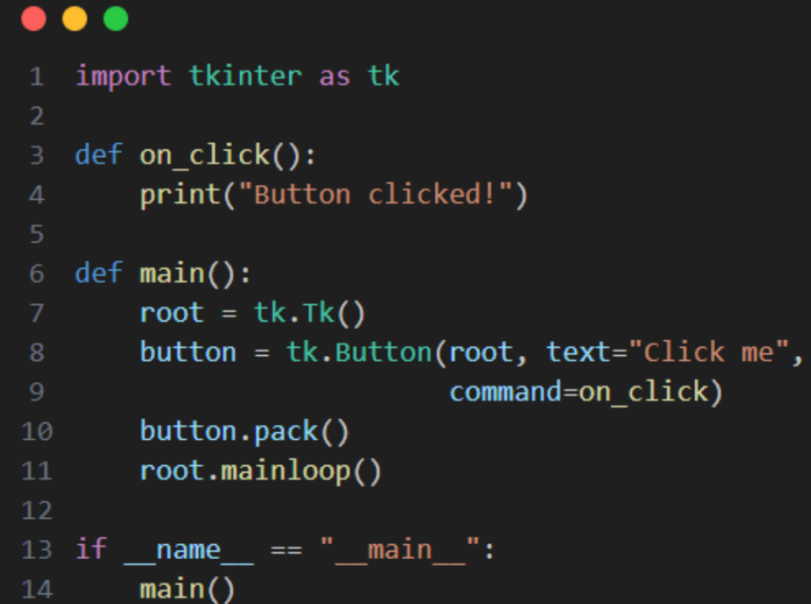
Парадигма логічного програмування

Логічне програмування в Python передбачає визначення правил і взаємозв'язків, що дозволяє системі робити висновки. Хоча Python не є нативною мовою логічного програмування, бібліотеки, такі як `kanren`, дозволяють використовувати цю парадигму. Вона особливо корисна в галузях штучного інтелекту та вирішення задач, де рішення можуть бути виведені на основі заданої логіки та обмежень.

```
1 from kanren import run, var, fact, Relation
2
3 parent = Relation()
4
5 def define_family():
6     fact(parent, 'Alice', 'Bob')
7     fact(parent, 'Bob', 'Charlie')
8
9 def main():
10     x = var()
11     print(run(1, x, parent(x, 'Charlie')))
12
13 if __name__ == "__main__":
14     define_family()
15     main()
```

Подіякерована парадигма

Програмування, кероване подіями, у Python зосереджується на реакції на події, такі як введення користувача або повідомлення. Ця парадигма широко використовується в графічних інтерфейсах (GUI) та мережевих системах. Бібліотека Tkinter у Python, разом із asyncio для асинхронної обробки подій, забезпечує надійну основу для створення застосунків, які динамічно реагують на події в реальному часі.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a Python script for a simple GUI using Tkinter. The script defines a function `on_click()` that prints "Button clicked!" and a `main()` function that creates a Tk window, adds a button with the text "Click me", and starts the main loop. The script is wrapped in an `if __name__ == "__main__":` block.

```
1 import tkinter as tk
2
3 def on_click():
4     print("Button clicked!")
5
6 def main():
7     root = tk.Tk()
8     button = tk.Button(root, text="Click me",
9                        command=on_click)
10    button.pack()
11    root.mainloop()
12
13 if __name__ == "__main__":
14     main()
```

Аспектно-орієнтоване парадигма (АОР)

Аспектно-орієнтоване програмування (АОР) у Python дозволяє відокремлювати наскрізні аспекти, такі як логування або безпека, від основної бізнес-логіки. Хоча Python не має вбудованої підтримки АОР, використання декораторів і бібліотек, таких як `aspectlib`, надає можливості аспектно-орієнтованого програмування, дозволяючи розробникам додавати додаткову поведінку до існуючого коду без його змін.

```
1 def log_decorator(func):
2     def wrapper(*args, **kwargs):
3         print(f"Calling {func.__name__}")
4         result = func(*args, **kwargs)
5         print(f"Finished {func.__name__}")
6         return result
7     return wrapper
8
9 @log_decorator
10 def add(x, y):
11     return x + y
12
13 if __name__ == "__main__":
14     print(add(5, 3))
```

Висновки

- Більшість сучасних інструментів заради універсальності можна розглядати як багато парадигмові.
- Парадигми можуть бути вбудовані в мову програмування або реалізовані завдяки додатковим бібліотекам (з розвитком мови програмування)
- Це розкриває широкі можливості для написання універсального коду, з довгою можливістю підтримки.

```
1 def log_decorator(func):
2     def wrapper(*args, **kwargs):
3         print(f"Calling {func.__name__}")
4         result = func(*args, **kwargs)
5         print(f"Finished {func.__name__}")
6         return result
7     return wrapper
8
9 @log_decorator
10 def add(x, y):
11     return x + y
12
13 if __name__ == "__main__":
14     print(add(5, 3))
```

Вибір парадигми програмування

Для обрання потрібної парадигми необхідно спочатку відповісти на наступні питання:

- Як довго код буде підтримуватись?
- Скільки працівників працюють над проектом?
- Чи є повторення в коді?
- Чи може прочитати ваш код одногрупник чи ви самі через тиждень?
- Чи в вас більше 100 рядків коду в одному файлі ?

Для більшої зручності більшість випадків що повторюються були класифіковані в патерни програмування та принципи програмування.

- Імперативна парадигма (лише 1 раз)
- Імперативна парадигма (менше 2х).
Більше 2х – необхідні елементи ООП
- Менше 2х – необхідні елементи функціонального програмування, більше 2х – комбінований підхід
- Не можете – це не мультипарадигмовий підхід а спагеті код
- Більше – не відповідність парадигмам.
Необхідно читати принципи солід

SOLID

SOLID (скорочення від англ. *Single Responsibility, Open–Closed, Liskov Substitution, Interface Segregation, Dependency Inversion*) в програмуванні — це мнемонічний акронім, введений Майклом Фезерсом (*Michael Feathers*) для перших п'яти принципів, сформульованих Робертом Мартіном на початку 2000-х. Вони визначають п'ять основних принципів об'єктно-орієнтованого проектування та програмування. Однак принципи **SOLID** застосовні не лише до об'єктно-орієнтованого програмного коду. Використання терміна «клас» означає лише інструмент об'єднання функцій і даних у групи. Будь-яка програмна система має подібні об'єднання, наприклад, це може бути «модуль» або функція



В усіх випадках коли не
знаєш що робити дивися
солід

Принцип єдиної відповідальності

Принцип єдиної відповідальності (*Single Responsibility Principle, SRP*) — це принцип об'єктно-орієнтованого програмування, згідно з яким кожен об'єкт повинен мати лише одну відповідальність, яка має бути повністю інкапсульована в класі. Усі його поведінкові аспекти повинні бути спрямовані виключно на забезпечення цієї відповідальності.

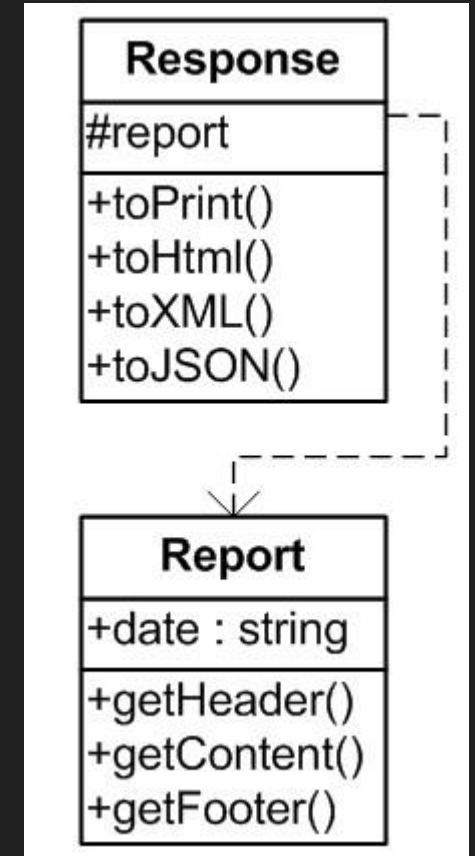


Мартін визначає відповідальність як **причину для змін** і робить висновок, що клас повинен мати **лише одну причину для змін**.

Наприклад, розглянемо клас, який формує та друкує звіт. Такий клас може змінитися з двох причин:

- Якщо змінюється **зміст звіту**.
- Якщо змінюється **формат звіту**.

Щоб дотримуватися **Принципу єдиної відповідальності (SRP)**, ці аспекти мають бути розділені між різними класами: один відповідатиме за **генерацію даних звіту**, а інший — за **відображення або друк**.



Принцип єдиної відповідальності

Принцип **SRP** — це не суворий закон, а **рекомендація**, яка має сенс лише в контексті змін у додатку:

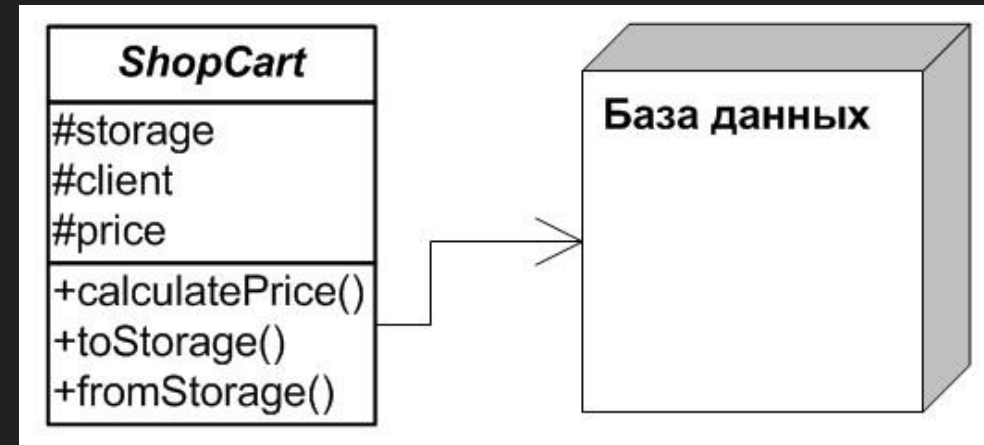
- Якщо при зміні коду, що відповідає за **одну відповідальність**, доводиться змінювати код, що стосується **іншої відповідальності**, це **перший сигнал** про порушення SRP.
- Якщо ж такі зміни **не зачіпають** код інших відповідальностей, то застосування SRP може бути **необов'язковим**.

Принцип **SRP** — це не суворий закон, а **рекомендація**, яка має сенс лише в контексті змін у додатку:

- Якщо при зміні коду, що відповідає за **одну відповідальність**, доводиться змінювати код, що стосується **іншої відповідальності**, це **перший сигнал** про порушення SRP.
- Якщо ж такі зміни **не зачіпають** код інших відповідальностей, то застосування SRP може бути **необов'язковим**.

Коли **SRP** дійсно варто застосовувати?

- Клас бере на себе занадто багато функцій.
- Уся доменна логіка концентрується в одному класі.
- Будь-які зміни в логіці об'єкта впливають на інші частини програми.
- Тестування, компіляція та виправлення коду зачіпає незалежні частини програми.
- Клас важко відокремити та використовувати в іншому місці через зайві залежності.



Однак **сліпе** слідування SRP може призвести до **надмірної складності** та **ускладнення підтримки**. Його варто застосовувати **розумно**, лише якщо це дійсно спрощує розробку та підтримку.

Принцип єдиної відповідальності

Принцип **SRP** — це не суворий закон, а **рекомендація**, яка має сенс лише в контексті змін у додатку:

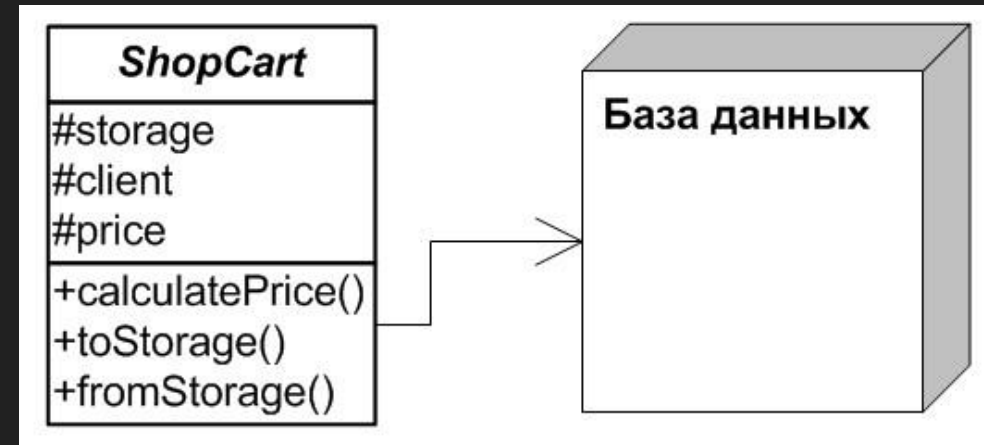
- Якщо при зміні коду, що відповідає за **одну відповідальність**, доводиться змінювати код, що стосується **іншої відповідальності**, це **перший сигнал** про порушення SRP.
- Якщо ж такі зміни **не зачіпають** код інших відповідальностей, то застосування SRP може бути **необов'язковим**.

Принцип **SRP** — це не суворий закон, а **рекомендація**, яка має сенс лише в контексті змін у додатку:

- Якщо при зміні коду, що відповідає за **одну відповідальність**, доводиться змінювати код, що стосується **іншої відповідальності**, це **перший сигнал** про порушення SRP.
- Якщо ж такі зміни **не зачіпають** код інших відповідальностей, то застосування SRP може бути **необов'язковим**.

Коли **SRP** дійсно варто застосовувати?

- Клас бере на себе занадто багато функцій.
- Уся доменна логіка концентрується в одному класі.
- Будь-які зміни в логіці об'єкта впливають на інші частини програми.
- Тестування, компіляція та виправлення коду зачіпає незалежні частини програми.
- Клас важко відокремити та використовувати в іншому місці через зайві залежності.



Однак **сліпе** слідування SRP може призвести до **надмірної складності** та **ускладнення підтримки**. Його варто застосовувати **розумно**, лише якщо це дійсно спрощує розробку та підтримку.

Принцип відкритості/закритості

Принцип відкритості/закритості (*Open–Closed Principle, OCP*) — це принцип об'єктно-орієнтованого програмування, який визначає: **«Програмні сутності (класи, модулі, функції тощо) повинні бути відкриті для розширення, але закриті для зміни».**

Це означає, що при додаванні нових можливостей у програму слід уникати модифікації вже існуючого коду. Замість цього потрібно використовувати механізми **спадкування, абстракції, інтерфейси або композицію**, щоб **розширювати** функціонал без зміни вихідного коду.

Принцип **OCP** допомагає зробити код **гнучким, масштабованим і менш схильним до помилок** при внесенні нових змін.

Бертран Мейєр найбільш відомий як **засновник терміна «Принцип відкритості/закритості» (OCP)**, який він сформулював у 1988 році у своїй книзі *Object-Oriented Software Construction*. Він відповідав на важливе питання: **«Як можна створити проєкт, стійкий до змін, життєвий цикл якого перевищує термін існування першої версії?»** Його відповідь полягала у формулюванні **OCP**, який дозволяє розширювати функціональність програми **без зміни її існуючого коду**, що забезпечує **гнучкість, стабільність і зручність у підтримці** програмних систем.



Принцип відкритості/закритості (поліморфний)

Протягом 1990-х років принцип відкритості/закритості (ОСР) був переосмислений у контексті абстрактних інтерфейсів. Згідно з цією інтерпретацією:

- Інтерфейси залишаються незмінними, але можуть мати кілька реалізацій, які можна поліморфно замінювати одна одною.
- Це підхід, протилежний початковій ідеї Мейєра, оскільки тепер ОСР підтримує успадкування від абстрактних базових класів.
- Інтерфейси можуть бути повторно використані через наслідування, але їхні конкретні реалізації можуть змінюватися або доповнюватися без модифікації існуючого коду.

Таким чином, існуючий інтерфейс має бути закритий для змін, а нові реалізації повинні як мінімум реалізовувати цей інтерфейс. Це забезпечує розширюваність без порушення стабільності системи.

```
class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == "credit_card":
            print(f"Processing credit card payment: ${amount}")
        elif payment_type == "paypal":
            print(f"Processing PayPal payment: ${amount}")

from abc import ABC, abstractmethod

# Абстрактний інтерфейс для платежів
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

# Реалізації платежів
class CreditCardPayment(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing credit card payment: ${amount}")

class PayPalPayment(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing PayPal payment: ${amount}")

# Новий метод оплати додається без зміни існуючого коду
class CryptoPayment(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing crypto payment: ${amount}")

# Використання поліморфізму
def process_transaction(payment_processor: PaymentProcessor, amount):
    payment_processor.process_payment(amount)
```

Принцип підстановки Лісков (Liskov Substitution Principle, LSP)

Принцип **LSP** описує коректну організацію підтипів у об'єктно-орієнтованому програмуванні (ООП).

Формулювання Барбари Лісков (1987):

- Якщо $q(x)$ — це властивість, що є істинною для об'єктів типу T , тоді $q(y)$ також має бути істинною для об'єктів типу S , де S є підтипом T .

Формулювання Роберта Мартіна (більш популярне):

- Функції, які використовують базовий клас, повинні мати можливість використовувати його підкласи, не знаючи про це.

Суть принципу LSP

- Об'єкти похідного класу повинні коректно замінювати об'єкти базового класу.
- Замінюваність підкласів не повинна змінювати поведінку програми.
- Успадкування не повинно порушувати логіку базового класу.



Принцип підстановки Лісков

```
class Bird:
    def fly(self):
        print("Птах летить")

class Penguin(Bird):
    def fly(self):
        raise Exception("Пінгвіни не вміють літати!")

# Використання:
def make_bird_fly(bird: Bird):
    bird.fly()

penguin = Penguin()
make_bird_fly(penguin) # ❌ Помилка! Пінгвін не може літати!
```

Проблема: базовий клас Bird припускає, що всі птахи вміють літати, але у підкласі Penguin це не так. Це порушує принцип LSP, оскільки Penguin не може замінити Bird без змін у логіці програми.

```
class Bird:
    pass

class FlyingBird(Bird):
    def fly(self):
        print("Птах летить")

class NonFlyingBird(Bird):
    pass

class Sparrow(FlyingBird):
    pass

class Penguin(NonFlyingBird):
    pass

# Використання:
def make_bird_fly(bird: FlyingBird):
    bird.fly()

sparrow = Sparrow()
make_bird_fly(sparrow) # ✅ Працює

penguin = Penguin()
# make_bird_fly(penguin) # ❌ Не передаємо нелітаючого птаха у функцію для літаючих
```

Принцип розділення інтерфейсу

Принцип розділення інтерфейсу (ISP) є одним із п'яти принципів SOLID у об'єктно-орієнтованому програмуванні (ООП).

Формулювання принципу ISP:

- «Клієнти не повинні залежати від інтерфейсів, які вони не використовують».

Суть принципу ISP

- Великі інтерфейси слід розділяти на кілька специфічних, менших інтерфейсів.
- Класам не потрібно реалізовувати методи, які вони не використовують.
- Зменшується кількість непотрібних залежностей між класами.

Принцип розділення інтерфейсу

```
from abc import ABC, abstractmethod

class Worker(ABC):
    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def eat(self):
        pass

# Реалізація для звичайного працівника
class HumanWorker(Worker):
    def work(self):
        print("Людина працює")

    def eat(self):
        print("Людина їсть")

# Реалізація для робота
class RobotWorker(Worker):
    def work(self):
        print("Робот працює")

    def eat(self): # ❌ Роботи не їдять!
        raise NotImplementedError("Роботи не їдять!")
```

Рішення: Ми створили два **окремих** інтерфейси (Workable та Eatable), і тепер робот не зобов'язаний реалізовувати непотрібний метод eat().

Проблема: клас RobotWorker змушений реалізовувати метод eat(), хоча це йому не потрібно.

```
from abc import ABC, abstractmethod

# Окремий інтерфейс для роботи
class Workable(ABC):
    @abstractmethod
    def work(self):
        pass

# Окремий інтерфейс для харчування
class Eatable(ABC):
    @abstractmethod
    def eat(self):
        pass

# Людина реалізує обидва інтерфейси
class HumanWorker(Workable, Eatable):
    def work(self):
        print("Людина працює")

    def eat(self):
        print("Людина їсть")

# Робот реалізує лише необхідний інтерфейс
class RobotWorker(Workable):
    def work(self):
        print("Робот працює")

# Використання:
def start_work(worker: Workable):
    worker.work()

human = HumanWorker()
robot = RobotWorker()
```

Принцип інверсії залежностей (Dependency Inversion Principle, DIP)

DIP є одним із ключових принципів SOLID у об'єктно-орієнтованому програмуванні (ООП).

Формулювання принципу DIP:

- «Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва повинні залежати від абстракцій».
- «Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій».

Суть принципу DIP

- Класи повинні залежати від абстракцій (інтерфейсів), а не від конкретних реалізацій.
- Зменшується зв'язаність коду, що робить його більш гнучким та масштабованим.
- Полегшується тестування та внесення змін у програму.

Принцип інверсії залежностей

```
class GmailService:
    def send_email(self, message):
        print(f"Sending email via Gmail: {message}")

class EmailNotification:
    def __init__(self):
        self.email_service = GmailService() # ✗ Жорстка залежність від GmailService

    def notify(self, message):
        self.email_service.send_email(message)

# Використання:
notifier = EmailNotification()
notifier.notify("Hello, world!")
```

Проблема: якщо ми захочемо змінити GmailService на OutlookService, доведеться змінювати код у EmailNotification, що **порушує DIP**.

Рішення:

- EmailNotification тепер залежить від інтерфейсу EmailService, а не від конкретної реалізації.
- Ми можемо легко замінити GmailService на OutlookService, не змінюючи код EmailNotification.

```
from abc import ABC, abstractmethod

# Абстракція (інтерфейс)
class EmailService(ABC):
    @abstractmethod
    def send_email(self, message):
        pass

# Реалізації, які дотримуються інтерфейсу
class GmailService(EmailService):
    def send_email(self, message):
        print(f"Sending email via Gmail: {message}")

class OutlookService(EmailService):
    def send_email(self, message):
        print(f"Sending email via Outlook: {message}")

# Клас тепер залежить від абстракції, а не конкретної реалізації
class EmailNotification:
    def __init__(self, email_service: EmailService):
        self.email_service = email_service # ✓ Залежність через абстракцію

    def notify(self, message):
        self.email_service.send_email(message)

# Використання:
gmail_notifier = EmailNotification(GmailService())
gmail_notifier.notify("Hello via Gmail!")

outlook_notifier = EmailNotification(OutlookService())
outlook_notifier.notify("Hello via Outlook!")
```

Кінець