**Міністерство освіти і науки України**
**Національний університет “Львівська політехніка”**
**Кафедра ІКТ**


# Лабораторна робота №1

*з дисципліни “ Об`єктно-орієнтоване програмування, частина 2”»*

підготував: студент групи ІХ-21

Котлінський Маркіян

**Львів - 2026**

**Тема заняття:** Застосування принципів SOLID об’єктно-орієнтованого програмування при проєктуванні програмних систем у сфері телекомунікацій.

**Мета роботи:** Ознайомитися з принципами SOLID. Навчитися застосовувати їх під час проєктування класів на Python. Формувати навички створення гнучкого, масштабованого та підтримуваного коду.

## ТЕОРЕТИЧНІ ВІДОМОСТІ

S — Single Responsibility (Принцип єдиної відповідальності)

**Суть:** Клас повинен мати лише **одну причину для зміни**. Він відповідає лише за одну логічну частину функціоналу.

- **Порушення:** Клас CallService, який одночасно обробляє дзвінок, рахує гроші та зберігає дані в БД.
- **Рішення:** Розділити на три спеціалізовані класи.

Python

```
class CallProcessor:
    def process(self): print("Дзвінок оброблено")

class CostCalculator:
    def calculate(self): print("Ціну пораховано")

class CallRepository:
    def save(self): print("Дані збережено")
```

O — Open/Closed (Принцип відкритості/закритості)

**Суть:** Програмні сутності мають бути **відкритими для розширення, але закритими для модифікації**. Ми додаємо новий функціонал, не змінюючи старий код.

- **Приклад:** Розрахунок тарифів. Замість if tariff == "basic", ми використовуємо успадкування.

Python

```
class Tariff:
    def price(self, min): return 0

class BasicTariff(Tariff):
    def price(self, min): return min * 1

class NightTariff(Tariff):
    def price(self, min): return min * 0.5
```

*Новий тариф додається просто створенням нового класу.*

L — Liskov Substitution (Принцип підстановки Лісков)

**Суть:** Об'єкти підкласів повинні замінювати об'єкти базового класу **без зміни коректності програми**. Нащадок не має "ламати" очікувану поведінку батька.

- **Приклад:** Різні типи з'єднань повинні мати однаковий інтерфейс connect().

```
class Network:
    def connect(self): pass
```

```
class WiFi(Network):
    def connect(self): print("WiFi підключено")

class LTE(Network):
    def connect(self): print("LTE підключено")
```

*Будь-яка функція, що приймає Network, повинна працювати з WiFi або LTE однаково.*

I — Interface Segregation (Принцип розділення інтерфейсу)

**Суть:** Краще багато вузькоспеціалізованих інтерфейсів, ніж один "товстий". Клас не повинен залежати від методів, які він не використовує.

- **Приклад:** IoT-датчик не вміє дзвонити, тому не має успадковувати методи телефонії.

```
class Callable:
    def make_call(self): pass

class DataTransferable:
    def send_data(self): pass

class Smartphone(Callable, DataTransferable):
    def make_call(self): print("Дзвінок...")
    def send_data(self): print("Дані...")

class IoTDevice(DataTransferable):
    def send_data(self): print("Дані датчика")
```

D — Dependency Inversion (Принцип інверсії залежностей)

**Суть:** Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва повинні залежати від **абстракцій**.

- **Приклад:** NetworkMonitor не знає, як саме працює логер (файл чи консоль), він просто викликає метод log().

```
class Logger: # Абстракція
    def log(self, msg): pass


class FileLogger(Logger):
    def log(self, msg): print(f"Файл: {msg}")


class NetworkMonitor:
    def __init__(self, logger: Logger): # Залежність від абстракції
        self.logger = logger

    def check(self):
        self.logger.log("Мережа OK")
```

## Хід роботи

### 1. Single Responsibility Principle (SRP)

### 1.1

Є клас CallReport, який формує звіт про дзвінки та зберігає його у файл.

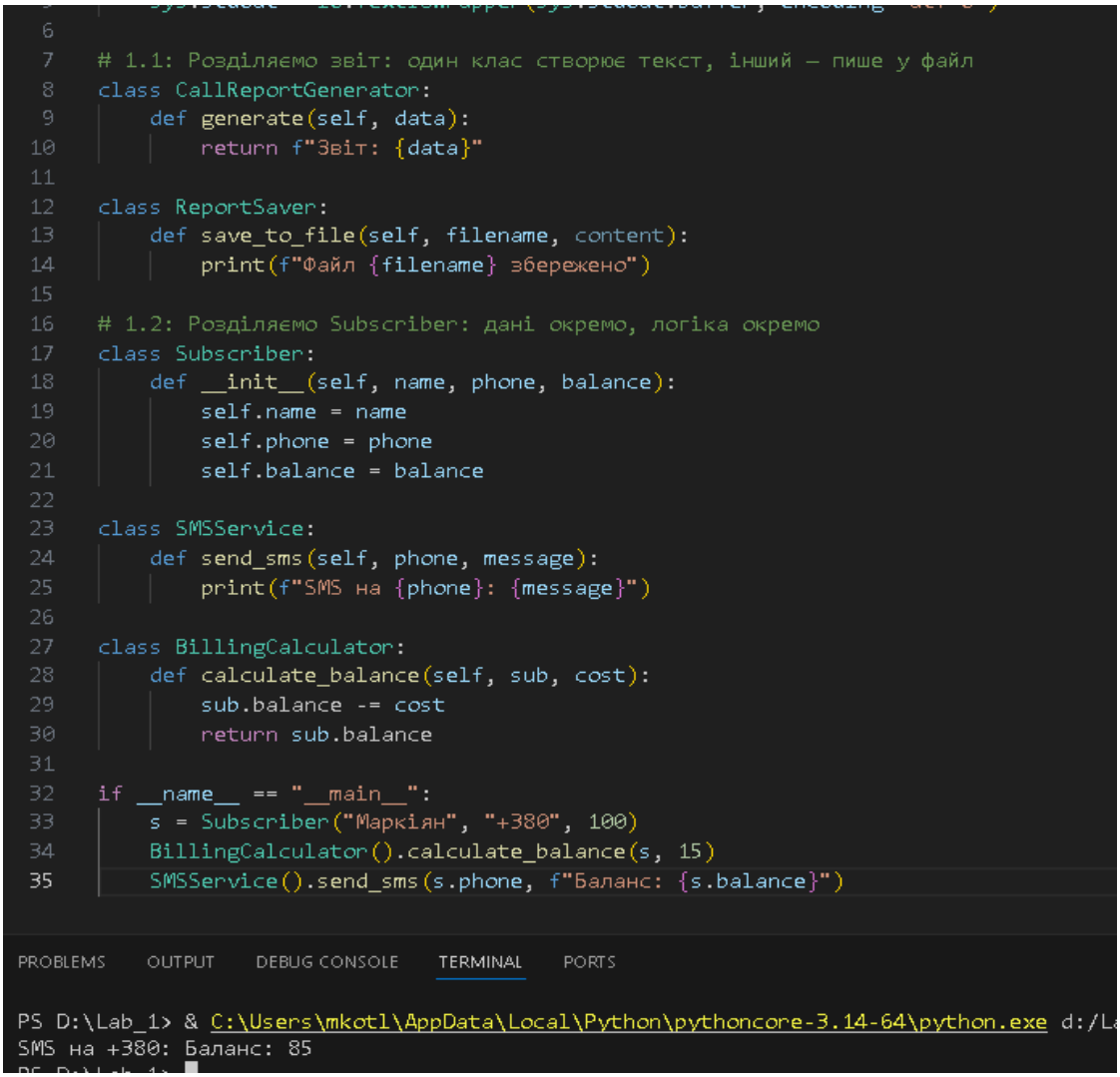
Розділити відповідальності відповідно до SRP.

### 1.2

Клас Subscriber:

- зберігає дані абонента
- відправляє SMS
- розраховує баланс

Перепроєктувати систему так, щоб кожен клас мав одну відповідальність.



## 2. Open/Closed Principle (OCP)

### 2.1

Реалізувати систему тарифікації дзвінків з базовим тарифом.

Додати новий тариф **без зміни існуючого коду**.

### 2.2

Система підтримує тарифи: VoiceTariff, DataTariff.

Розширити систему тарифом RoamingTariff, не змінюючи логіку розрахунку вартості.

```python

# 2.1: База для тарифів. Закрита для змін, відкрита для нових класів
class Tariff:
    def calculate_price(self, units):
        raise NotImplementedError

class BaseTariff(Tariff):
    def calculate_price(self, units):
        return units * 0.5

# 2.2: Додаємо нові тарифи просто новими класами
class VoiceTariff(Tariff):
    def calculate_price(self, units):
        return units * 0.6

class DataTariff(Tariff):
    def calculate_price(self, units):
        return units * 0.1

class RoamingTariff(Tariff):
    def calculate_price(self, units):
        return units * 5.0

if __name__ == "__main__":
    for t in [BaseTariff(), RoamingTariff()]:
        print(f"{t.__class__.__name__}: {t.calculate_price(100)} грн")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Lab_1> & C:\Users\mkotl\AppData\Local\Python\pythoncore-3.14-64\python.exe d:/Lab_1/Zavd_2.py
BaseTariff: 50.0 грн
RoamingTariff: 500.0 грн
```

## 3. Liskov Substitution Principle (LSP)

### 3.1

Є базовий клас NetworkConnection.

Реалізувати підкласи LTEConnection, WiFiConnection, які можна взаємозамінно використовувати.

### 3.2

Виправити створену ієрархію, де клас SatelliteConnection порушує очікувану поведінку базового класу (супутник не може працювати як звичайне з’єднання).

```python
# 3.1: WiFi та LTE працюють однаково через базовий клас
class NetworkConnection:
    def connect(self):
        return "З'єднання..."

class WiFiConnection(NetworkConnection):
    def connect(self):
        return "WiFi ок"

class LTEConnection(NetworkConnection):
    def connect(self):
        return "LTE ок"

# 3.2: Супутник специфічний, тому виносимо підготовку в окремий інтерфейс
class AdvancedConnection(NetworkConnection):
    def prepare(self): pass

class SatelliteConnection(AdvancedConnection):
    def prepare(self):
        return "Шукаю супутник..."
    def connect(self):
        return "Супутник підключено"

if __name__ == "__main__":
    conns = [WiFiConnection(), SatelliteConnection()]
    for c in conns:
        if isinstance(c, AdvancedConnection): print(c.prepare())
        print(c.connect())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Lab_1> & C:\Users\mkotl\AppData\Local\Python\pythoncore-3.14-64\python.exe d:/Lab_1/Zavd_3.py
WiFi ок
Шукаю супутник...
Супутник підключено
```

## 4. Interface Segregation Principle (ISP)

### 4.1

Інтерфейс TelecomDevice має методи:

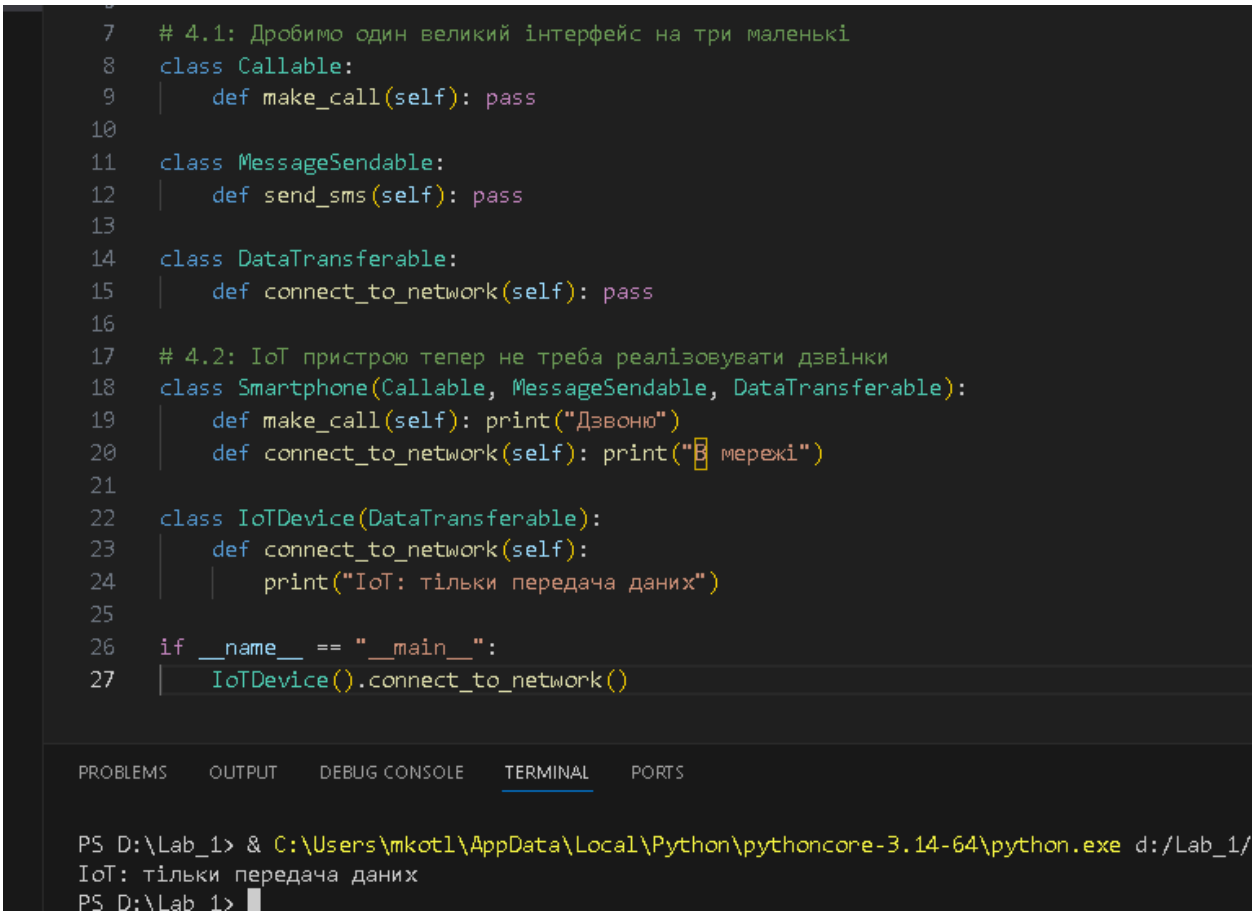
- make_call()
- send_sms()
- connect_to_network()

Розділити інтерфейс відповідно до ISP.

### 4.2

IoT-пристрій у мережі оператора повинен лише передавати дані.

Спроєктувати систему інтерфейсів без зайвих методів.



## 5. Dependency Inversion Principle (DIP)

### 5.1

Система моніторингу мережі напряму використовує FileLogger.

Перепроєктувати систему відповідно до DIP.

### 5.2

Реалізувати можливість підключення:

- FileLogger
- ServerLogger
- ConsoleLogger

```python
import sys
import io

if sys.platform == "win32":
    sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')

# 5.1: Монітор залежить від абстрактного Logger, а не від конкретного файлу
class Logger:
    def log(self, msg): pass

# 5.2: Можемо підставити будь-який логер без змін у моніторі
class FileLogger(Logger):
    def log(self, msg): print(f"Лог у файл: {msg}")

class ServerLogger(Logger):
    def log(self, msg): print(f"Лог на сервер: {msg}")

class NetworkMonitor:
    def __init__(self, logger: Logger):
        self.logger = logger
    def check(self):
        self.logger.log("Мережа працює")

if __name__ == "__main__":
    # Підставляємо серверний логер
    m = NetworkMonitor(ServerLogger())
    m.check()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Lab_1> & C:\Users\mkotl\AppData\Local\Python\pythoncore-3.14-64\python.exe d:/La
Лог на сервер: Мережа працює
```

## Контрольні запитання

**1. У чому полягає принцип єдиної відповідальності?** Клас повинен мати лише одну задачу та одну причину для зміни.

**2. Чому порушення SRP ускладнює підтримку коду?** Зміна однієї функції може зламати іншу в тому ж класі, а сам код стає заплутаним і важким для тестів.

**3. Як принцип OCP допомагає масштабувати систему?** Дозволяє додавати нові функції (тарифи) через нові класи, не змінюючи старий, перевірений код.

**4. Наведіть приклад порушення LSP.** Супутникове з'єднання, яке потребує специфічного пошуку сигналу, тоді як програма очікує від нього стандартної поведінки як від звичайного 4G/WiFi.

**5. Чим інтерфейс відрізняється від абстрактного класу?** Абстрактний клас може мати готовий код, а інтерфейс — це лише перелік методів, які «зобов'язаний» реалізувати нащадок.

**6. Чому ISP важливий у великих системах?** Щоб прості пристрої (як IoT-датчики) не залежали від складних методів (дзвінки, SMS), які вони не використовують.

**7. Поясніть різницю між DIP та Dependency Injection.** DIP — це стратегія (залежність від абстракцій), а DI — це інструмент (передача об'єкта через конструктор).

**8. Як принципи SOLID впливають на тестування?** Код стає модульним, тому кожен клас можна тестувати окремо від решти системи.

**9. Які принципи SOLID найважливіші для телекомунікаційних систем?** OCP (для нових тарифів), LSP (для заміни типів мереж) та ISP (для IoT).

**10. Які наслідки ігнорування SOLID у промислових проєктах?** «Ефект доміно» (одна правка ламає все), гігантська вартість підтримки та неможливість додати нові фічі без переписування всього проєкту.

## Висновок

Під час виконання лабораторної роботи я ознайомився з принципами SOLID та успішно застосував їх для проєктування об'єктно-орієнтованої архітектури мовою Python. На основі практичних завдань навчився розділяти логіку класів для уникнення їхнього перевантаження (SRP), безпечно розширювати систему новими тарифами без зміни існуючого коду (OCP) та правильно будувати ієрархії мережевих з'єднань (LSP). У результаті я здобув практичні навички створення гнучкого, чистого та масштабованого програмного забезпечення, що легко піддається тестуванню та подальшій підтримці