**Модуль 1. Прийоми об'єктно-орієнтованого програмування**

**Заняття 1: Software Engineering та принципи SOLID**

**Software Engineering як процес**

[**Програмна інженерія**](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B0_%D1%96%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%96%D1%8F) (Software Engineering) - це інженерна галузь, пов'язана з розробкою програмного продукту з використанням чітко визначених наукових принципів, методів та процедур. Результатом розробки програмного забезпечення є ефективний та надійний програмний продукт.

**Причини виникнення**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%BF%D1%80%D0%B8%D1%87%D0%B8%D0%BD%D0%B8-%D0%B2%D0%B8%D0%BD%D0%B8%D0%BA%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F)Приблизно кожні 5 років кількість розробників подвоюється. Очевидний наслідок цього факту — в будь-який момент часу половина всіх розробників на ринку мають менше 5 років досвіду та недостатній рівень кваліфікації, щоб створювати якісний програмний продукт.

Зараз будь-яку людину на нашій планеті оточує мінімум 5 обчислювальних пристроїв різного рівня складності, які виконують кимось написаний код. Наше життя дуже залежить від того, що код в навколишніх пристроях працює без збоїв та помилок.

Поєднання цих двох факторів призводить до того, що вимоги до якості програмного забезпечення постійно зростають, але недостатній рівень кваліфікації більшості розробників не дозволяє відповідати цим вимогам.

В результаті ми отримуємо сучасну бізнес-модель розробки ПЗ, коли користувачі перших "стабільних" версій виступають фактично тестувальниками для неякісного, неналагодженого та випущеного поспіхом програмного продукту.

Фактично, зараз, на початку 2020-х, індустрія розробки програмного забезпечення переживає кризу становлення.

Перші ознаки кризи, що насувається, проявилися наприкінці 1960-х років. Багато програмних проектів зазнали невдачі і вийшли за межі бюджету. На виході вийшло ненадійне програмне забезпечення, обслуговування якого вимагає високих витрат. Програмне забезпечення більшого розміру стало досить дорогим в обслуговуванні. Це призводило до того, що програмне забезпечення не здатне було задовольнити зростаючі вимоги замовника, а попит на нове програмне забезпечення збільшувався швидше, ніж можливість створення нового програмного забезпечення.

Вирішенням проблеми було перетворення неорганізованого програмування на дисципліну програмної інженерії. Ці інженерні моделі покликані допомогти компаніям оптимізувати операції та надати програмне забезпечення, яке відповідає вимогам клієнтів.

З кінця 1970-х років настає початок розробки та широкого використання різних підходів та принципів програмної інженерії. У 1980-х роках відбулася автоматизація процесу розробки програмного забезпечення та зростання автоматизованої розробки програмного забезпечення. 1990-ті роки стали часом підвищеної уваги до «управлінських» аспектів проектів, стандартів якості та процесів, таких як ISO 9001. На початку 2000-х років дуже швидкими темпами наростає популярність гнучких підходів (Agile) до розробки ПЗ та стає практично стандартом навіть у тих сферах, де не зовсім підходить. У 2010-х тривале зростання складності ПЗ виводить на перше місце чисто управлінську проблему складності розробки ПЗ паралельно декількома командами розробників в межах одного продукту. Популярним вирішенням проблеми паралельної розробки стає мікросервісна архітектура, постійна інтеграція мікросервісів (CI) та постійна доставка (CD).

**Основні проблеми, які стоять перед розробниками ПЗ**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%96-%D0%BF%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%B8-%D1%8F%D0%BA%D1%96-%D1%81%D1%82%D0%BE%D1%8F%D1%82%D1%8C-%D0%BF%D0%B5%D1%80%D0%B5%D0%B4-%D1%80%D0%BE%D0%B7%D1%80%D0%BE%D0%B1%D0%BD%D0%B8%D0%BA%D0%B0%D0%BC%D0%B8-%D0%BF%D0%B7)Розмір та складність програмного забезпечення. У реальному житті рідко зустрінеш реальний продукт із кодовою базою менше 100000 рядків.

* Масштабованість. Успішний продукт може зростати експоненційно та розробники повинні закладати можливість швидко та легко налаштовувати ПЗ, щоб воно справлялося з тисячократно збільшеним навантаженням.
* Адаптивність. Вимоги бізнесу до програмного продукту постійно змінюються та продукт повинен бути досить гнучким, щоб швидко та дешево задовольняти вимогам, що змінюються.
* Вартість. Розвиток промисловості та масове виробництво знизило вартість комп'ютерів та електронного обладнання, і тепер основна частина вартості більшості сучасних продуктів — це їхнє ПЗ. У свою чергу, це породжує вимогу знизити накладні витрати на розробку ПЗ, збільшивши ефективність розробки, щоб успішно конкурувати на ринку.
* Управління якістю та ціною якості ПЗ. Бізнес хоче мати можливість гнучко контролювати накладні витрати на виробництво ПЗ та розуміти, яку якість він отримає за яку вартість.

**Методології програмної інженерії**

* [​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D1%96%D1%97-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D1%97-%D1%96%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%96%D1%97)Підходи до управління процесом розробки (Agile, Scrum, Kanban та ін.)
* Принципи проектування (SOLID, аспекти, Domain Driven Design та ін.)
* Архітектурні принципи (мікросервісна архітектура, event-driven архітектура).
* Підходи до розробки (Test Driven Design, парне програмування та ін.)
* Патерни проектування

**Важливість використання підходів та методологій програмної інженерії**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%B2%D0%B0%D0%B6%D0%BB%D0%B8%D0%B2%D1%96%D1%81%D1%82%D1%8C-%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%BF%D1%96%D0%B4%D1%85%D0%BE%D0%B4%D1%96%D0%B2-%D1%82%D0%B0-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D1%96%D0%B9-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D1%97-%D1%96%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%96%D1%97)

**Знижує складність**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%B7%D0%BD%D0%B8%D0%B6%D1%83%D1%94-%D1%81%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D1%96%D1%81%D1%82%D1%8C)Велике програмне забезпечення — це завжди складно, і його складно розвивати та підтримувати. У програмної інженерії є чудове рішення, яке дозволяє знизити складність будь-якого проекту. Програмна інженерія розділяє великі проблеми на різні дрібні. А потім починає вирішувати кожну невелику проблему по черзі. Всі ці маленькі проблеми вирішуються незалежно одна від одної.

**Зводить до мінімуму вартість**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D1%8C-%D0%B4%D0%BE-%D0%BC%D1%96%D0%BD%D1%96%D0%BC%D1%83%D0%BC%D1%83-%D0%B2%D0%B0%D1%80%D1%82%D1%96%D1%81%D1%82%D1%8C)Програмне забезпечення вимагає багато зусиль, а програмісти — високооплачувані спеціалісти. Для розробки програмного забезпечення з великою кількістю рядків коду потрібно багато робочої сили. Але при розробці програмного забезпечення програмісти все проектують і цим зменшують кількість непотрібних речей. У свою чергу, вартість виробництва програмного забезпечення стає меншою у порівнянні з будь-яким програмним забезпеченням, в якому не використовується метод програмної інженерії.

**Скорочує час**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D1%81%D0%BA%D0%BE%D1%80%D0%BE%D1%87%D1%83%D1%94-%D1%87%D0%B0%D1%81)Все, що зроблено не відповідно до проекту, завжди призводить до марнування часу. І дуже трудомістка процедура — отримати остаточний робочий код. Якщо з нею не впоратися належним чином, це може зайняти багато часу. Тому, якщо ви створюєте своє програмне забезпечення відповідно до методу розробки програмного забезпечення, це значно скоротить вам час.

**Робота з великими проектами**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%B2%D0%B5%D0%BB%D0%B8%D0%BA%D0%B8%D0%BC%D0%B8-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B0%D0%BC%D0%B8)Великі проекти не виконуються за декілька днів, і вони вимагають великого терпіння, планування та управління. А щоб інвестувати шість або сім місяців у будь-який проект, потрібно багато планування, управління, тестування та обслуговування. Ніхто не хоче, щоб на виконання проекту було заплановано чотири місяці, а після цього часу виявилося, що проект все ще перебуває на першій стадії. Компанія виділила багато ресурсів на план, і він повинен бути виконаний. Таким чином, щоб без проблем впоратися з великим проектом, компанія повинна вдатися до методів розробки програмного забезпечення.

**Надійність**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%BD%D0%B0%D0%B4%D1%96%D0%B9%D0%BD%D1%96%D1%81%D1%82%D1%8C)Програмне забезпечення повинно бути надійним. Якщо ви поставили програмне забезпечення кінцевому споживачеві, воно повинно працювати, принаймні, протягом певного часу або передплати. І якщо у програмному забезпеченні виникають якісь помилки, компанія несе відповідальність за усунення всіх цих помилок. Це називається підтримка проекту. Оскільки у програмній інженерії проводяться тестування та супровід, тому не варто турбуватися про його надійність.

**Ефективність**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%B5%D1%84%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D1%96%D1%81%D1%82%D1%8C)Ефективність буде, якщо щось зроблено відповідно до деяких стандартів. Стандарти програмного забезпечення — основна ціль розробки, щоб зробити проект ефективнішим. Таким чином, програмне забезпечення стає ефективнішим за допомогою програмної інженерії.

**Додаткові матеріали**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-se/#%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D0%BE%D0%B2%D1%96-%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D0%B8)

Для глибшого занурення у тему можна почитати такі мануали:

1. [Software Engineering Tutorial](https://www.javatpoint.com/software-engineering-tutorial) (https://www.javatpoint.com/software-engineering)
2. [Software Engineering Tutorial](https://www.tutorialspoint.com/software_engineering/index.htm) (<https://www.tutorialspoint.com/software_engineering/index.htm>)

**Мова UML**

Знання мови UML — це обов'язкова вимога для будь-якого розробника на будь-якій мові. Розроблений на початку 1990-х набір умовних позначень для відображення елементів ПЗ та зв'язків між ними застосовується на проектах будь-якого розміру та складності, незалежно від мови, що застосовується.

Основна частина документації містить діаграми UML і, без вміння читати документацію, розробнику зараз просто не можна.

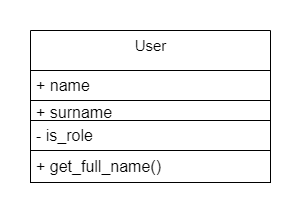
Відповідно до [Wiki](https://uk.wikipedia.org/wiki/Unified_Modeling_Language):

**UML** (Unified Modeling Language — уніфікована мова моделювання) — мова графічного опису для об'єктного моделювання в галузі розробки програмного забезпечення, для моделювання бізнес-процесів, системного проектування та відображення організаційних структур.

UML є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, яка називається UML-моделлю. UML була створена для визначення, візуалізації, проектування та документування, переважно, програмних систем. UML не є мовою програмування, але на основі UML-моделей можлива генерація коду.

**UML-діаграма класу**

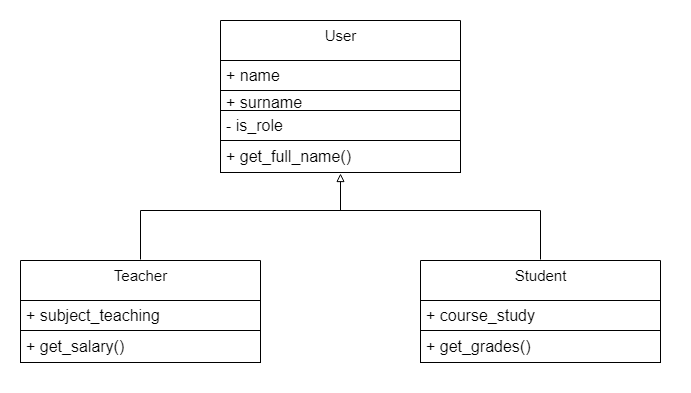
[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#uml-%D0%B4%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0-%D0%BA%D0%BB%D0%B0%D1%81%D1%83)Базово пройдемося основними моментами, щоб почати розуміти UML діаграми



На зображенні представлена UML-діаграма класу. Перший блок User — це назва класу. Друга секція нижче назви — це поля або атрибути класу, тобто його властивості. Плюс або мінус перед назвою властивості говорить про області видимості. Знак '+' відповідає public властивості, а знак '-' — відповідно private область видимості. І останніми йдуть методи класу.

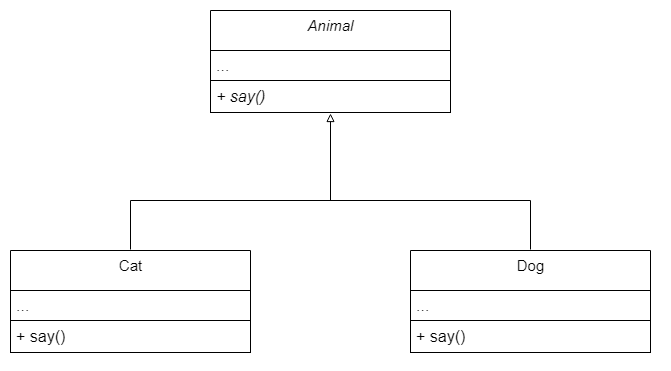
**Наслідування класів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%BD%D0%B0%D1%81%D0%BB%D1%96%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BB%D0%B0%D1%81%D1%96%D0%B2)Щоб зобразити ієрархію класів, використовують стрілки



На цій діаграмі класи Teacher та Student є частиною ієрархії класу User. Стрілка з трикутним порожнім закінченням говорить про наслідування між класами. Клас User — це суперклас, а класи Teacher та Student — підкласи.

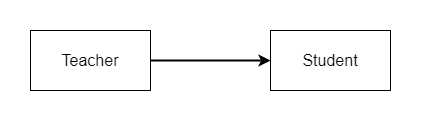
Якщо наслідування походить від абстрактного класу, то назва абстрактного класу в UML діаграмах пишеться курсивом:



**Відношення між об'єктами**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%B2%D1%96%D0%B4%D0%BD%D0%BE%D1%88%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BC%D1%96%D0%B6-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D0%B0%D0%BC%D0%B8)

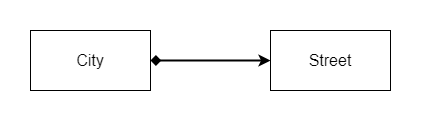
**Асоціація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%B0%D1%81%D0%BE%D1%86%D1%96%D0%B0%D1%86%D1%96%D1%8F)Коли один об'єкт використовує інший або залежить від нього, відношення називається асоціацією. В UML асоціація позначається простою стрілкою, яка спрямована у бік залежності



**Композиція**

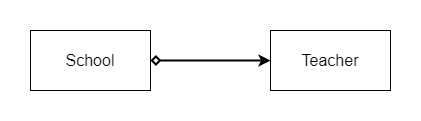
[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D1%96%D1%8F)Це відношення «частина-ціле» між двома об'єктами, коли один з них включає в себе інший. Особливість цього відношення полягає в тому, що компонент може існувати лише як частина контейнера. Місто складається з вулиць.



В UML композиція позначається лінією зі стрілкою на одному кінці та заповненим ромбом на іншому кінці. Ромб направлений у бік контейнера, а стрілка — у бік об'єкта, що включається.

**Агрегація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%B0%D0%B3%D1%80%D0%B5%D0%B3%D0%B0%D1%86%D1%96%D1%8F)Це менш суворий варіант композиції, коли один об'єкт просто має посилання на інший об'єкт. Тут контейнер не керує життєвим циклом компонента. Компонент може існувати окремо від контейнера. Наприклад, школа має вчителів, але зв'язок тут слабший у порівнянні з містом-вулиця.



В UML агрегація зображується як композиція, але з порожнім ромбом.

**Додаткові матеріали**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-01/module-01-01/lesson-01-01-uml#%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D0%BE%D0%B2%D1%96-%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D0%B8)Про UML існує досить багато джерел та статей. Дамо перелік деяких для кращого розуміння матеріалу:

1. [UML: від теорії до практики](https://javarush.ru/groups/posts/uml-v-java) (https://javarush.com/groups/posts/uml-v-java)
2. [UML для початківців](https://www.youtube.com/watch?v=0I9aIP5gKCg&list=PLPPIc-4tm3YTw3FUu75jsW4QgrXopfXhX) (<https://www.youtube.com/watch?v=0I9aIP5gKCg&list=PLPPIc-4tm3YTw3FUu75jsW4QgrXopfXhX>)

**Принципи проектування SOLID**

SOLID — це абревіатура, складена з перших літер п'яти базових принципів об'єктно-орієнтованого програмування (Single responsibility, Open-closed, Liskov substitution, Interface segregation та Dependency inversion.) та дизайну, запропонована Робертом Мартіном.

Принципи SOLID використовуються для дизайну та розробки таких програмних систем, які, з великою ймовірністю, зможуть тривалий час розвиватися, розширюватися та підтримуватися. Ці принципи дають змогу будувати на базі ООП масштабовані та супроводжувані програмні продукти зі зрозумілою бізнес-логікою.

Розшифровка:

* Single responsibility — принцип єдиної відповідальності
* Open-closed — принцип відкритості / закритості
* Liskov substitution — принцип підстановки Барбари Лісков
* Interface segregation — принцип розділення інтерфейсу
* Dependency inversion — принцип інверсії залежностей

**Принцип єдиної відповідальності (Single responsibility)**

Принцип єдиної відповідальності (Single Responsibility Principle, SRP) означає, що у модуля повинна бути лише одна причина для зміни. Весь код, який змінюється з цієї причини, повинен бути зібраний у цьому модулі.

Під причиною слід розуміти бізнес-причину. Такою причиною може бути відділ, який замовляє деякий функціонал, власник продукту, група користувачів тощо.

Під модулем ми розумітимемо якусь частину коду, відокремлену від інших. Це може бути клас, функція, об'єкт, файл — загалом усе, що має межі і визначає код як самостійну одиницю.

З SRP добре пов'язаний прийом – виділення класу. Це коли з великого класу з безліччю слабо-пов'язаних за змістом полів та методів, виділяється один або кілька класів. Суть у тому, щоб явно виділити призначення класу або, іншими словами, виділити клас, який можна описати однією фразою.

Важливою непрямою ознакою правильно обраного класу буде наявність зручного, зрозумілого імені для цього класу, та ім'я повинно бути іменником з прикметником. Якщо доводиться використовувати віддієслівні іменники, то, найімовірніше, що призначення класу вибрано неправильно.

Для розуміння цього принципу розглянемо наступний приклад:

class Person:

def \_\_init\_\_(self, name, zip, city, street):

self.name = name

self.zip = zip

self.city = city

self.street = street

def get\_address(self):

return f'{self.zip}, {self.city}, {self.street}'

person = Person('Alexander', '36007', 'Poltava', 'European, 28')

print(person.get\_address())

У нас є клас Person. Відповідно до принципу єдиної відповідальності клас повинен вирішувати лише якесь одне завдання. Зараз він вирішує два: зберігає дані користувача та виконує логіку перетворення адреси користувача.

Необхідно зробити так, щоб клас Person працював тільки з даними користувача, а завдання перетворення адреси делегувалося іншому екземпляру класу PersonAddress через залежність у конструкторі.

class PersonAddress:

def \_\_init\_\_(self, zip, city, street):

self.zip = zip

self.city = city

self.street = street

def value\_of(self):

return f'{self.zip}, {self.city}, {self.street}'

class Person:

def \_\_init\_\_(self, name, address):

self.name = name

self.address = address

def get\_address(self):

return self.address.value\_of()

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

person = Person('Alexander', PersonAddress('36007', 'Poltava', 'European, 28'))

print(person.get\_address())

Тепер ми дотримуємося принципу єдиної відповідальності.

**Принцип відкритості-закритості (Open-closed)**

Цей принцип допомагає вирішувати проблему, коли невелика зміна в одній частині системи викликає лавину змін в інших частинах. Якщо у програмі при зміні поліпшення потрібно виправити десятки модулів, така система, швидше за все, спроектована погано.

Принцип відкритості-закритості декларує, що модулі повинні бути відкриті для розширення, але закриті для зміни.

Іншими словами — модулі потрібно проектувати так, щоб їх не можна було змінювати, а нова функціональність у програмі повинна з'являтися лише за допомогою створення нових сутностей та композиції їх зі старими. Звичайно, завжди є зміни, які неможливо внести, не змінивши код якогось модуля — жодна система не може бути закрита від змін повністю.

Щоб зрозуміти принцип, потрібно розглянути застосування його у місцях з'єднання модулів.

Розглянемо наступний приклад:

class Rect:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def total\_area(shapes):

sum = 0

for el in shapes:

sum += el.width \* el.height

return sum

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

shapes = [Rect(10, 10), Rect(4, 5), Rect(3, 3)]

area = total\_area(shapes)

print(area)

Є клас Rect, що описує прямокутник, і є функція total\_area, яка обчислює загальну площу фігур. У чому може виникнути неприємність для такого коду?

Якщо у нас з'явиться нова фігура, наприклад, коло — Circle, нам доведеться змінити роботу функції total\_area.

from math import pi

class Rect:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

class Circle:

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

self.radius = radius

def total\_area(shapes):

sum = 0

for el in shapes:

if isinstance(el, Rect):

sum += el.width \* el.height

if isinstance(el, Circle):

sum += el.radius \*\* 2 \* pi

return sum

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

shapes = [Rect(10, 10), Circle(5), Rect(4, 5), Rect(3, 3), Circle(3)]

area = total\_area(shapes)

print(area)

І з появою нової фігури нам потрібно щоразу вносити зміни в роботу функції total\_area. Щоб виправити ситуацію, потрібно перекласти обчислення площі фігури на самі класи:

from math import pi

class Rect:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area\_of(self):

return self.width \* self.height

class Circle:

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

self.radius = radius

def area\_of(self):

return self.radius \*\* 2 \* pi

def total\_area(shapes):

sum = 0

for el in shapes:

sum += el.area\_of()

return sum

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

shapes = [Rect(10, 10), Circle(5), Rect(4, 5), Rect(3, 3), Circle(3)]

area = total\_area(shapes)

print(area)

Тепер все добре, і при появі нової фігури вносити зміну в роботу функції total\_area більше не потрібно.

**Принцип підстановки Барбари Лісков (Liskov substitution)**

Принцип відкритості-закритості говорить про те, що у добре спроектованих програмах нова функціональність вводиться шляхом додавання нового коду, а не зміною старого, що вже працює. Принцип підстановки Лісков (далі LSP) — це як реалізувати цей принцип при побудові ієрархії наслідування класів в об'єктно-орієнтованих мовах програмування. По суті, правильна ієрархія наслідування в ООП - це ієрархія, побудована відповідно до LSP, щоб відповідати принципу відкритості-закритості.

У попередньому прикладі порушенням LSP була функція:

def total\_area(shapes):

sum = 0

for el in shapes:

if isinstance(el, Rect):

sum += el.width \* el.height

if isinstance(el, Circle):

sum += el.radius \*\* 2 \* pi

return sum

Тепер давайте розглянемо, як можна порушити LSP не таким очевидним способом. Припустимо, ми розробляємо програму, яка працює з геометричними фігурами. У ній є клас для роботи з прямокутниками:

class Rect:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def set\_width(self, width):

self.width = width

def set\_height(self, height):

self.height = height

def area\_of(self):

return self.width \* self.height

Тепер потрібно реалізувати фігуру квадрат. Квадрат – це очевидно прямокутник і цілком логічно, що клас Square повинен бути спадкоємцем класу Rect.

class Square(Rect):

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

Rect.\_\_init\_\_(self, size, size)

def set\_width(self, width):

self.width = width

self.height = width

def set\_height(self, height):

self.width = height

self.height = height

І якщо у функції ми використовуємо клас Rect, а працюємо з конкретним класом Square, то можуть виникнути проблеми:

def test\_shape\_size(shape):

shape.set\_width(10)

shape.set\_height(20)

return shape.area\_of() == 200 # умова не спрацює, якщо shape — екземпляр класу Square

Відповідно до LSP нам необхідно використовувати спільний інтерфейс для обох класів і не наслідувати Square від Rect. Цей спільний інтерфейс повинен бути таким, щоб класи-спадкоємці могли б використовуватися замість батьківських класів, від яких вони утворені, не порушуючи роботу програми.

from enum import Enum

class SideType(Enum):

TYPE\_WIDTH = 'width'

TYPE\_HEIGHT = 'height'

class Shape:

def set\_side(self, size, side):

pass

def area\_of(self):

pass

class Rect(Shape):

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def set\_side(self, size, side):

if SideType.TYPE\_WIDTH == side:

self.width = size

if SideType.TYPE\_HEIGHT == side:

self.height = size

def set\_width(self, width):

self.set\_side(width, SideType.TYPE\_WIDTH)

def set\_height(self, height):

self.set\_side(height, SideType.TYPE\_HEIGHT)

def area\_of(self):

return self.width \* self.height

class Square(Shape):

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

self.edge = size

def set\_side(self, size, side=None):

self.edge = size

def set\_width(self, width):

self.set\_side(width)

def area\_of(self):

return self.edge \*\* 2

def get\_area\_of\_shape(figure: Shape):

return figure.area\_of()

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

square = Square(10)

rect = Rect(5, 10)

print('Square area: ', get\_area\_of\_shape(square))

print('Rect area: ', get\_area\_of\_shape(rect))

Тепер поведінка спадкоємців не конфліктує із поведінкою базового класу. Це дозволить використовувати і Rect, і Square там, де оголошено використання Shape.

**Принцип розділення інтерфейсу (Interface segregation)**

Сутності не повинні залежати від інтерфейсів, які вони не використовують.

Коли принцип порушується, модулі схильні до всіх змін в інтерфейсах, від яких вони залежать. Це призводить до високої зв'язаності модулів один з одним.

ISP допомагає проектувати інтерфейси так, щоб зміни зачіпали тільки ті модулі, на функціональність яких вони справді впливають. Найчастіше це змушує інтерфейси дробити (розділяти).

Припустимо, що у нас є клас Programmer, який описує програміста з офісу деякої компанії. Співробітники пишуть код та іноді їдять піцу, яку компанія замовляє в офіс.

class Programmer:

def write\_code(self):

pass

def eat\_pizza(self, slice\_count):

pass

class OfficeProgrammer(Programmer):

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

self.name = name

def eat\_pizza(self, slice\_count):

print(f'{self.name} eat {slice\_count} slice pizza!')

def write\_code(self):

print(f'{self.name} write code!')

Через деякий час компанія почала наймати фрілансерів, які працюють віддалено і піцу не їдять. Якщо ми використовуємо той самий інтерфейс, то клас RemoteProgrammer повинен буде реалізувати метод eat\_pizza, хоча він йому і не потрібен.

class RemoteProgrammer(Programmer):

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

self.name = name

def write\_code(self):

print(f'{self.name} write code!')

def eat\_pizza(self, slice\_count):

pass

Ми можемо уникнути проблеми з прикладу вище, якщо розділимо клас Programmer. Ми можемо поділити його на дві ролі: CodeProducer та PizzaConsumer.

class CodeProducer:

def write\_code(self):

pass

class PizzaConsumer:

def eat\_pizza(self, slice\_count):

pass

class OfficeProgrammer(CodeProducer, PizzaConsumer):

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

self.name = name

def eat\_pizza(self, slice\_count):

print(f'{self.name} eat {slice\_count} slice pizza!')

def write\_code(self):

print(f'{self.name} write code!')

class RemoteProgrammer(CodeProducer):

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

self.name = name

def write\_code(self):

print(f'{self.name} write code!')

Тепер і OfficeProgrammer, і RemoteProgrammer будуть реалізовувати лише ті інтерфейси, які їм справді потрібні.

**Принцип інверсії залежностей (Dependency inversion)**

Відповідно до принципу, класи повинні залежати від інших класів не напряму, а від абстракцій. Класи верхніх рівнів не повинні залежати від класів нижніх рівнів. Обидва типи класів повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій. У процесі розробки програмного забезпечення існує момент, коли функціонал застосунку перестає поміщатися в межах одного модуля або класу. Коли це відбувається, нам доводиться вирішувати проблему залежностей класів (модулів). В результаті, наприклад, може виявитися так, що високорівневі компоненти залежать від низькорівневих компонентів.

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))

Клас ApiClient (високорівневий модуль) залежить від класу RequestConnection (абстракція). Він не використовує пакет requests напряму. Надалі це, наприклад, дозволить підмінити пакет requests всередині класу RequestConnection, не зачіпаючи роботу класу ApiClient. У прикладі ми знаходимо курс продажу долара та євро на поточну дату за допомогою API ПриватБанк.