приклад успадкування: постановка завдання

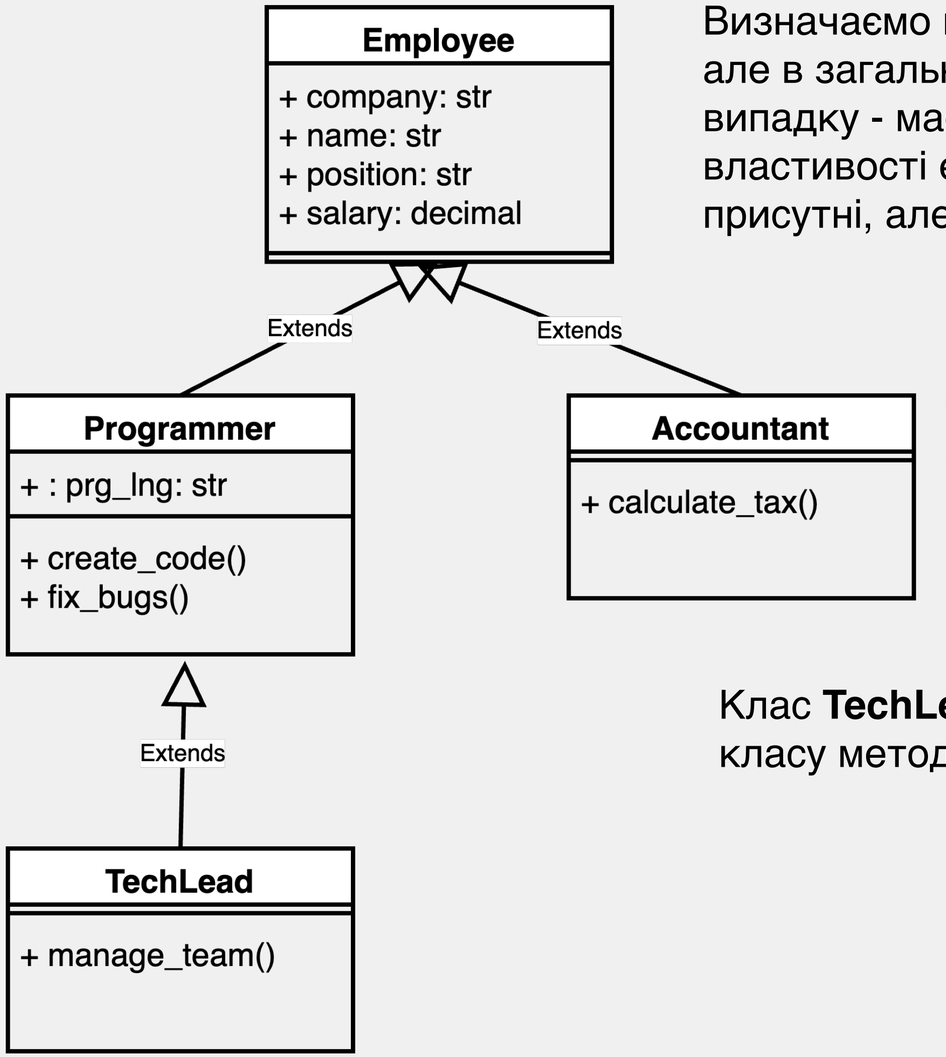
Давайте уявимо що нам треба написати код, який описує співробітників невеликої IT-компанії:
- всі співробітники мають ім’я, посаду, рівень зарплати.
- у нас є бухгалтер, який вміє розраховувати податки.
- частина співробітників - програмісти. Вони спеціалізуються на якійсь мові програмування і вміють фіксити баги і писати код (ну і прихована функція при написанні коду - створювати баги).
- є техлід, який також вміє робити все що і інші програмісти, але ще вміє керувати командою.

Зверніть увагу що ми можемо одразу визначити структури з загальними ознаками, а потім специфікувати їх покроково, додаючи на кожному кроці специфічні ознаки. Наприклад, у нас вимальовується клас Employee, який описує співробітника компанії з усіма специфічними атрибутами:
- атрибут класу company
- атрибути екземпляру name, position, salary

Так як бухгатер (клас Accountant), це також співробітник, то всі атрибути, які є в класі Employee, повинні бути і там. Тобто - Accountant необхідно наслідувати від Employee, і додати йому специфічну властивість - можливість розраховувати податки. Додамо специфічний метод calculate_tax() (він буде символічним, нам на цьому етапі важлива структура, а не деталізація розрахунків податків).

Саме таким чином ми і будемо рухатись і на першому етапі - створимо діаграму класів які хочемо створити.

приклад успадкування: діаграма класів



Визначаємо клас **Employee**, який включає в собі всі атрибути і методи (на цей час їх немає, але в загальному випадку можуть бути) які є загальними для всіх інших класів. В нашому випадку - маємо одну властивість класу (company) яка загальна для всіх екземплярів, і властивості екземплярів (name, position, salary) - які для кожного співробітника обов'язково присутні, але мають специфічні значення.

Класи **Programmer** і **Accountant** - розширюють клас Employee, додаючи кожен свої атрибути, специфікуючи два різних класа співробітників. І це ніяк не заперечує існування атрибутів, які визначаються батьківським класом.

- при реалізації методів ми будемо робити це формально, без деталізації. Нам зараз просто треба розуміти що методи є і вони можуть працювати.

Клас **TechLead** - розширює клас Programmer, додаючи йому специфічні саме для цього класу методи.

приклад успадкування: код

```python
class Employee:
    company = "Web factory"

    def __init__(self, name, salary, position):
        self.name = name
        self.salary = salary
        self.position = position


class Accountant(Employee):
    def calculate_tax(self):
        return f"Accountant {self.name} calculates tax"


class Programmer(Employee):
    def __init__(self, name, salary, position, prg_lng):
        super().__init__(name, salary, position)
        self.prg_lng = prg_lng

    def create_code(self):
        return f"Programmer {self.name} writes code"

    def fix_bug(self):
        return f"Programmer {self.name} fixes bug"


class TechLead(Programmer):

    def manage_team(self):
        return f"TechLead {self.name} manages team"
```

Спочатку створимо клас **Employee** - який описує загальні для всіх співробітників властивості і методи. Атрибут **company** - який є загальним для всіх екземплярів, виносимо у властивості класу, всі властивості які є для кожного співробітника індивідуальні - виновимо у властивості екземплярів, які будемо ініціювати в методі **__init__.**

Наслідування класу **Accountant** від **Employee** автоматично передає йому всі атрибути, визначені в **Employee**. Нам необхідно визначити лише специфічний метод **calculate_tax()**.

При визначенні класу **Programmer** дещо цікавіше: його екземпляри повинні мати додаткову властивість - **prg_lng**. Відповідно, нам необхідно перевизначити метод **__init__()**. Зверніть увагу на функцію **super()** (насправді це не функція, але давайте так вважати для спрощення) - вона дає можливість звернутись до методів батьківського класу і викликати його методи при необхідності (виклик з атрибутами дає можливість керувати детальніше - дивіться документацію). Ми викликаємо метод **__init__** класу Employee. Перевикористання існуючих методів - це вірний шлях, так як 1) ми не повторюємо код 2) якщо відбудуться якісь зміни - то їх необхідно робити в одному місті коду.

При визначенні класу **TechLead** - ми не зустрічаємо нічого незвичайного: клас повторює **Programmer** і доповнює його специфічною поведінкою. Головна ідея цього прикладу - глибина наслідування може бути якою завгодно.

приклад успадкування: розширення завдання

Наступним кроком розвиваємо завдання: від нас вимагається створити метод класу Accountant, який дасть можливість друкувати щомісячну відомість для перерахувань коштів на рахунки співробітників. Всі наші співробітники діляться на дві категорії - ФОП і ті що знаходяться в штаті компанії. За тих хто в штаті компанія сплачує податки і перераховує їм узгоджену суму зарплати. ФОПи сплачують податки самостійно, тому згідно домовленості компанія повинна перерахувати їм обумовлену суму зарплати плюс необхідні податки.
Всі програмісти і техліди - є ФОПи, а бухгалтери знаходяться в штаті компанії.

```
class Accountant(Employee):
    def calculate_tax(self):
        return f"Accountant {self.name} calculates tax"

    def get_payment_statement(self, employees: list[Employee]) -> str:
        statement = f"pay by company '{self.company}': \n"
        for employee in employees:
            statement += (f"{employee.name:>20}, "
                          f"{employee.position:>20} - gets "
                          f"{employee.get_salary_payment():<15}\n")
        return statement
```

Спочатку визначимо додатковий метод в класі Accountant, який повинен сформувати відомість. Цей метод очікує у якості параметра список з усіма співробітниками, для яких треба сформувати відомість.
Цей метод фактично висуває вимоги щодо інтерфейсу до кожного співробітника - від звертається в циклі до методу співробітника get_salary_payment(), який повинен повернути суму оплати саме цьому співробітнику за місяць.

Зверніть увагу на f-рядок: взяття його в дужки вказує Python що все це треба об’єднати в один рядок. При виводі значень f-рядка використана мова форматування яка дозволяє покращити вивід (наприклад: {value:>20} - для виводу value використати 20 знакомісць, друк value зсунути вправо, і так далі - дивіться документацію).

Це поширена поведінка, коли сам екземпляр знає як цому треба розрахувати платіж (згадайте “магічні методи”, які описують для кожного екземпляру як з ним поводитись в різних контекстах). Тобто - тепер нам треба безпосередньо в класах, які описують конкретних співробітників (для нас це класи Accountant, Programmer, TechLead), визначити метод get_salary_payment().

приклад успадкування: розширення завдання

```python
class Programmer(Employee):
    def __init__(self, name, salary, position, prg_lng):
        super().__init__(name, salary, position)
        self.prg_lng = prg_lng

    def create_code(self):
        return f"Programmer {self.name} writes code"

    def fix_bug(self):
        return f"Programmer {self.name} fixes bug"

    def get_salary_payment(self):
        return f"{(self.salary / 0.95):.2f} UAH"
```

Тепер створимо відповідний метод в класі **Programmer**. Для спрощення вважаємо що від суми яку отримає ФОП треба платити 5% податків і це всі відрахування. Таким чином, для вирахування суми до сплати, треба очікувану зарплата розділити на 0.95. Ми там використовуємо форматування - вказуємо виводити два знака після коми.

клас **TechLead** не потребує визначення в ньому такого методу, так як він унаслідує цей метод від класу **Programmer**.

Зверніть увагу на два факта:

- при великій кількості класів, які будуть описувати різні типи співробітників компанії - і це може робитись різними розробниками і в різний час - доволі легко забути імплементувати метод **get_salary_payment()** - його присутність абсолютно неочевидна і з’ясовується лише при ознайомленні з методом **get_payment_statement()** в класі **Accountant**. Це неочевидно і заплутано. Було б здорово якость підказати розробникам, які будуть розвивати і супроводжувати цей код що метод **get_salary_payment()** повинен бути присутнім в кожному екземплярі співробітника.
- клас **Employee**, який визначає досить велику кількість атрибутів, сам ніде напряму не використовується для створення екземплярів. Едина причина існування цього класу - він необхідний для наслідування.

Такі класи - які створюються лише для наслідування і описують частину атрибутів, а також можуть висувати умови реалізації певного інтерфейсу (методів) в похідних класах - називають **абстрактними класами**.

приклад успадкування: абстрактні класи

```python
from abc import ABC, abstractmethod


class Employee(ABC):
    company = "Web factory"

    def __init__(self, name, salary, position):
        self.name = name
        self.salary = salary
        self.position = position

    @abstractmethod
    def get_salary_payment(self)-> float:
        """Implementation is carried out using
        the Accountant.get_payment_statement() method.
        """
        pass
```

Python надає модуль abc, який включає в себе інструменти для створення абстрактних класів.
Щоб створити абстрактний клас достатньо наслідувати його від класу ABC з цього методу, і після цього будь-яка спроба створити екземпляр з цього класу призведе до виникнення виключення з відповідним повідомленням.
Також такий клас може ставити вимоги до похідних класів імплементувати якісь методи. Щоб це зробити необхідно визначити цей метод (можна просто “пустим”) і огорнути його декоратором @abstractmethod. Це створює вимогу в будь-якому похідному класі обов’язково перевизначити цей метод. Якщо цього не буде зроблено - генерується виключення з відповідним повідомленням.

Таким чином, завдяки визначенню класу Employee як абстрактного і визначення в ньому абстрактних методів, ми виключили можливість його використання для створення екземплярів і висунули вимоги обов’язкової імплементації певних методів. Хороша практика в документації цього метода вказати що саме висуває вимогу створення цього методу і що він нього очікується.

Зверніть увагу що ми не перевизначали метод **get_salary_payment()** для класу **TechLead** - тому що визначили в класі **Programer** і та реалізація нас влаштовує.