Programowanie
przed 9 wykładem

Andrzej Giniewicz

10.05.2024

W tym tygodniu na zajęciach zajmiemy się diagramami klas w standardzie UML.

UML (od ang. *Unified Modeling Language*, czyli zunifikowany język modelowania) jest zbiorem standardów opisujących różne, często wykorzystywane w projektowaniu oprogramowania diagramy. Najczęściej stosowanym rodzajem diagramów opisywanych przez UML są diagramy klas. Diagramy klas opisują klasy oraz relacje pomiędzy nimi.

Do rysowania diagramów klas można użyć dodatku do VSCode PlantUML[1]. PlantUML to darmowe narzędzie i język, za pomocą którego możemy zdefiniować diagramy w kodzie i automatycznie wygenerować ich graficzną reprezentację. Więcej na temat PlantUML można przeczytać na stronie projektu[2] oraz w dokumentacji[3].

1 Diagramy klas

Głównym elementem budulcowym diagramu klas jest ramka mająca trzy segmenty: nazwę klasy, atrybuty oraz metody. Klasa poniżej

```
class Osoba:
    def __init__(self, imię, nazwisko):
        self.imię = imię
        self.nazwisko = nazwisko

    def przedstaw_się(self):
        print(f"Dzień dobry, nazywam się {self.imię} {self.nazwisko}.")
```

może być zwizualizowana za pomocą diagramu

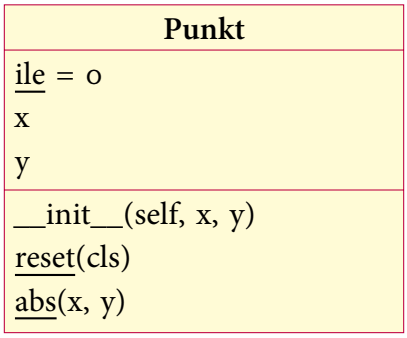
Osoba
imię nazwisko
__init__(self, imię, nazwisko) przedstaw_się(self)

[1] https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml
[2] https://www.plantuml.com/
[3] https://crashedmind.github.io/PlantUMLHitchhikersGuide/

Gdyby jakieś atrybuty były atrybutami klasy, należy je podkreślić. Podobna sytuacja ma miejsce z metodami klasy lub statycznymi, które również podkreślamy. UML nie posiada rozróżnienia na metody klas i statyczne, jednak jeśli wykorzystujemy standardową nazwę `cls` dla pierwszego parametru metod klas, możemy wykorzystać go do odróżnienia. Klasę

```
class Punkt:
    ile = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        Punkt.ile += 1

    @classmethod
    def reset(cls):
        cls.ile = 0

    @staticmethod
    def abs(x, y):
        return sqrt(x**2+y**2)
```

zwizualizujemy jako



Opisywany wariant uwzględnia różne rodzaje metod i atrybutów, ale nie uwzględnia modyfikatorów dostępu. Aby uwzględnić modyfikatory dostępu, dodajemy przed nazwą atrybutu lub metody odpowiedni symbol

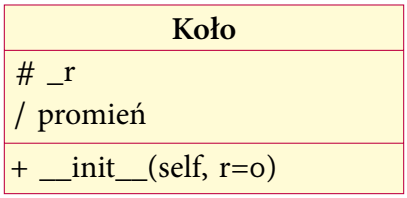
Symbol	Znaczenie
+	dostęp publiczny
#	dostęp chroniony
–	dostęp prywatny
/	właściwość (atrybut wyliczany w locie)

Przykładowo

```
class Koło:
    def __init__(self, r=0):
```

```
        self._r = r

    @property
    def promień(self):
        return self._r

    @promień.setter
    def promień(self, r):
        if r<0:
            raise ValueError("Promień musi być nieujemny")
        self._r = r
```

zapiszemy jako



Warto zwrócić uwagę, że opis klasy w diagramie nie zawiera implementacji, czyli z samego diagramu nie odczytamy, za co dana metoda jest odpowiedzialna, jednakże w połączeniu z dokumentacją i innymi diagramami, stanowią formę opisu wykonywanego projektu. Same diagramy pojedynczych klas dają niewielkie zyski, jednakże, gdy dodamy do nich odpowiednie relacje, możemy zwiększyć ich użyteczność.

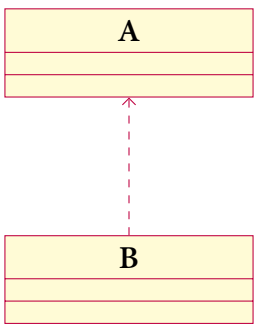
Na diagramach UML zaznaczamy sześć rodzajów połączeń pomiędzy klasami: zależność, związek, agregację, kompozycję, realizację oraz dziedziczenie.

1.1 Zależność

Mówimy, że pomiędzy klasami A i B występuje **zależność** (ang. *dependency*), jeśli zmiana definicji klasy A pociąga za sobą konieczność zmiany definicji klasy B.

Zależność może występować na przykład, gdy klasa B używa metod z klasy A. Gdy klasa A zostanie zmodyfikowana i będzie miała inne nazwy metod, konieczna jest aktualizacja kodu klasy B. Zaznaczenie takiej informacji na diagramie klas ułatwia znalezienie wszystkich miejsc, w których konieczne są zmiany.

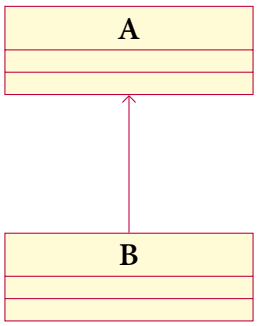
Zależność jest kierunkowa i zaznaczana jest linią przerywaną zakończoną otwartym grotem strzałki, kierującym od klasy B (używającej) do klasy A (definiującej).



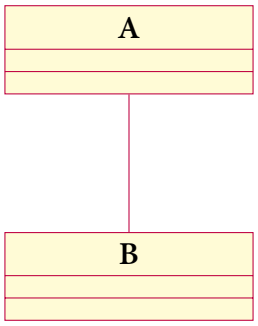
1.2 Związek

Mówimy, że pomiędzy klasami A i B występuje **związek** (ang. *association*), gdy klasa B przechowuje referencję do klasy A.

Zależność może występować na przykład, gdy w klasie `Samochód` przechowujemy instancję klasy `Ubezpieczenie` jako atrybut — pomiędzy tymi obiektami występuje stały związek. Związek może być kierunkowy lub dwukierunkowy. Związek jednokierunkowy zaznaczany jest linią ciągłą zakończoną otwartym grotem strzałki, kierującym od klasy B (zawierającej referencję) do klasy A (stanowiącej cel referencji).



Związek dwukierunkowy, w którym A jest z wiązane z B oraz B jest związane z A, zaznaczamy poprzez linię ciągłą pomiędzy klasami (bez grotów strzałek).

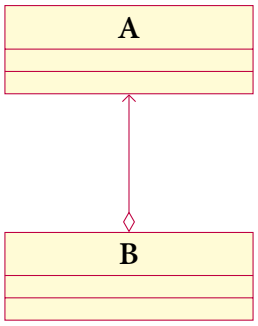


1.3 Agregacja

Mówimy, że pomiędzy klasami A i B występuje **agregacja** (ang. *aggregation*), gdy pomiędzy A i B występuje związek i logicznie oznacza „bycie częścią czegoś".

Pomiędzy agregacją a związkiem nie musi być różnicy w implementacji, jednak jest potencjalna różnica w znaczeniu. Związek (jak `Samochód` i `Ubezpieczenie`) nie musi oznaczać zawierania (ubezpieczenie nie jest częścią samochodu, jest z nim jedynie związane). Przykładem agregacji będzie natomiast `Samochód` oraz `Koło`, ponieważ koło jest częścią samochodu.

Agregację zaznaczamy strzałką będącą linią ciągłą zaczynającą się niezamalowanym równoległobokiem a kończącą otwartym grotem strzałki, kierującą od klasy składowej (B) do klasy nadrzędnej (A).

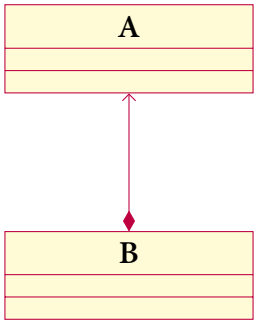


1.4 Kompozycja

Mówimy, że pomiędzy klasami A i B występuje **kompozycja** (ang. *composition*), gdy pomiędzy A i B występuje agregacja i logicznie oznacza „bycie nierozerwalną częścią czegoś".

Pomiędzy kompozycją a agregacją oraz związkiem nie musi być różnicy w implementacji, jednak ponownie jest różnica w znaczeniu. O ile agregacja może być rozerwalna (koła samochodu możemy przełożyć do innego), o tyle kompozycja jest nierozerwalna — o ile nie projektujemy systemu dla kliniki transplantologii, serca człowieka nie przełożymy do innej instancji klasy. Z punktu widzenia implementacji jest to informacja, że gdy obiekt nadrzędny przestanie istnieć, można również zwolnić pamięć przeznaczoną na jego składowe, ponieważ na pewno nigdzie już się nie przydadzą.

Kompozycję zaznaczamy podobnie jak agregację, ale używamy zamalowanego równoległoboku.



1.5 Realizacja

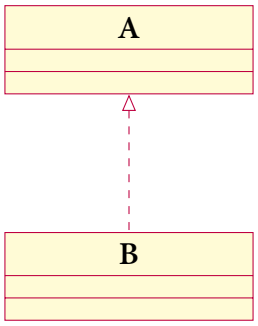
Mówimy, że klasa B **realizuje** klasę A, jeśli spełnia wymogi, aby być używaną w roli A.

Ten rodzaj zależności jest używany, jeśli w dowolnej metodzie oczekującej obiektu klasy B, możemy też wykorzystać obiekt klasy A. Jest to użyteczne w języku Python, który wyznaje zasadę **duck-typing** (jeśli coś wygląda jak kaczka i kwacze jak kaczka, prawdopodobnie jest to kaczka[4]). Dzięki duck-typing na przykład możemy w wielu funkcjach korzystać zamiennie z krotek, list i tablic NumPy. Dzieje się tak, ponieważ wszystkie te

[4]Precyzyjniejsza definicja dostępna w dokumentacji `https://docs.python.org/3/glossary.html#term-duck-typing`.

obiekty możemy indeksować, zatem, pomimo że nie są powiązane dziedziczeniem, dla wielu zastosowań można traktować dowolny z tych typów jako realizację dowolnego z pozostałych. Większość połączeń, w których typów możemy używać zamiennie, można zaznaczyć na diagramie UML jako realizacje.

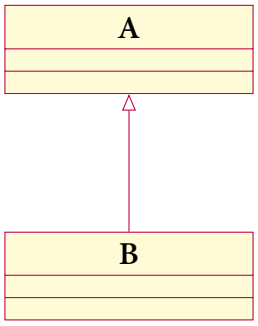
Realizację zaznaczamy jako strzałkę będącą linią przerywaną, zakończoną trójkątem, kierującą od klasy B do A.



1.6 Dziedziczenie

Mówimy, że klasa B **dziedziczy z** klasy A, jeśli jest jej realizacją zaimplementowaną za pomocą dziedziczenia.

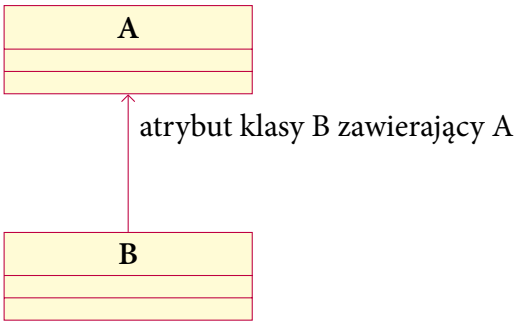
Dziedziczenie zaznaczamy jako strzałkę będącą linią ciągłą, zakończoną trójkątem, kierującą od klasy B do A.



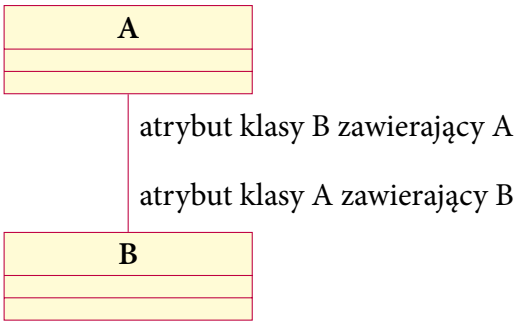
Dziedziczenie jest najczęstszym rodzajem realizacji, w niektórych językach jedynym dostępnym.

2 Związek, agregacja i kompozycja po raz drugi

Związek, agregacja i kompozycja nie muszą różnić się z punktu widzenia implementacji, dlatego uzupełnimy ich opis w jednym miejscu. W każdej z wersji muszą jakoś przechowywać referencje, zatem muszą mieć odpowiednie atrybuty na to przeznaczone. Atrybuty te pomijamy z listy atrybutów i zapisujemy nazwę atrybutu przy strzałce.



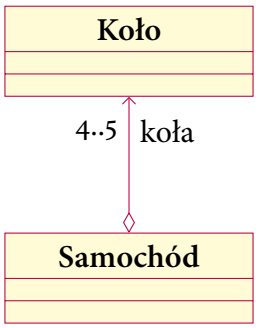
Dla zależności dwukierunkowych możemy napisać dwa atrybuty przy linii.



Dodatkowo przy liniach możemy zaznaczać krotności. Często spotykane krotności to

Krotność	Znaczenie
1	dokładnie jeden
1..∗	jeden lub więcej
∗ albo 0..∗	zero lub więcej
0..1	zero lub jeden
n..k	od n do k włącznie

W przypadku samochodu, posiadającego cztery koła i opcjonalnie zapasowe, napisalibyśmy



Oznacza to, że w klasie `Samochód` jest atrybut `koła`, zawierająca od czterech do pięciu obiektów klasy `Koło`.

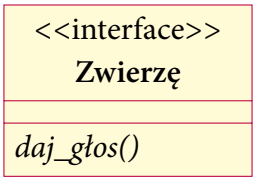
3 Typy i abstrakcyjne klasy bazowe a UML

Adnotacje typów mogą pojawić się nie tylko przy stałych i funkcjach, jak miało to miejsce w pierwszych przykładach, ale również przy metodach, choć wtedy pomijamy adnotację dla argumentu `self`. Oznacza to, że warto je jakoś zaznaczyć w diagramach UML, jeśli z nich korzystamy. Również abstrakcyjne klasy bazowe warto wyszczególnić w specjalny sposób.

W przypadku klas i metod abstrakcyjnych zapisujemy je za pomocą kursywy. Oznacza to, że klasa

```
from abc import ABC, abstractmethod

class Zwierzę(ABC):
    @abstractmethod
    def daj_głos(self):
        pass
```

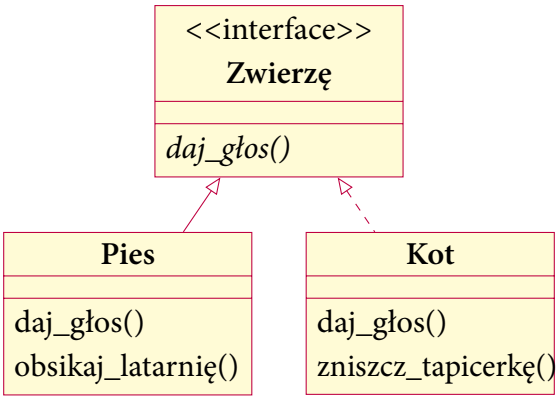
będzie zapisana jako



W przypadku dziedziczenia z klasy bazowej użyjemy strzałki do dziedziczenia, natomiast w przypadku wirtualnego dziedziczenia przez rejestrację, użyjemy strzałki do realizacji. Dla klas z przykładu

```
class Pies(Zwierzę):
    def daj_głos(self):
        print("hau hau")

    def obsikaj_latarnię(self):
        pass  # TODO: zaimplementować

class Kot:
    def daj_głos(self):
        print("miau miau")

    def zniszcz_tapicerkę(self):
        pass  # TODO: zaimplementować

Zwierzę.register(Kot)
```

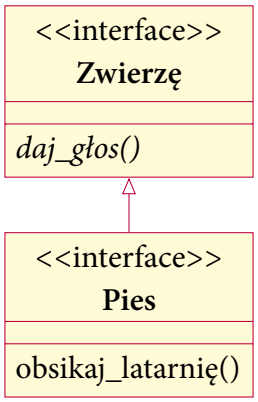
będziemy mieli



Pamiętajmy też, że jeśli nasza klasa nie implementuje metod abstrakcyjnych, sama również jest abstrakcyjna, czyli jej nazwa powinna pojawić się kursywą. Wariant klasy dziedziczącej bez implementacji metody z przykładu

```
class Pies(Zwierzę):
    def obsikaj_latarnię(self):
        pass  # TODO: zaimplementować
```

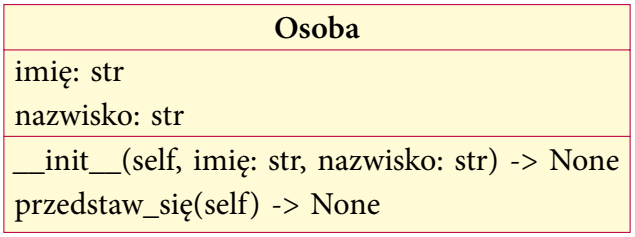
zapiszemy jako



Jeśli klasę z poprzednich zajęć uzupełnimy o typy jak poniżej,

```
class Osoba:
    def __init__(self, imię: str, nazwisko: str) -> None:
        self.imię = imię
        self.nazwisko = nazwisko

    def przedstaw_się(self) -> None:
        print(f"Dzień dobry, nazywam się {self.imię} {self.nazwisko}.")
```

powinny one pojawić się na diagramie z wykorzystaniem tej samej składni, czy przy definicji



4 Podsumowanie

Standard UML jest stosunkowo elastyczny. Znajdziemy wiele różnych jego implementacji i rekomendacji. Zasady przedstawione w niniejszym dokumencie stanowią pewne dostosowanie często spotykanych reguł oraz ich zakresu do języka Python i jego charakterystyki.

Oprócz diagramów klas często spotykanymi diagramami UML są diagramy przypadków użycia (ang. *use case diagram*), sekwencji (ang. *sequence diagram*) oraz czynności (ang. *activity diagram*). O tych i innych diagramach można przeczytać w wielu książkach, na przykład dostępnej na platformie Safari „The Unified Modeling Language User Guide”[5].

[5]Autorzy to Grady Booch, James Rumbaugh, Ivar Jacobson. Polecam drugie wydanie z 2005 roku, które ukazało się nakładem wydawnictwa Addison-Wesley Professional.