OOP a návrhové vzory

KIV/ADT – 11. přednáška

Miloslav Konopík

19. května 2023

Obsah

Objektově orientované programování.

[PPA] Objekty

Objekt:

- Entita reálného světa:
  - data (atributy/vlastnosti).
  - akce pro jejich zpracování (funkce/metody).

Abstrakce:

- Zjednodušení reality – práce s daty relevantními pro danou aplikaci.
- Například:
  - Firma má zaměstnance, každý má jméno a bere nějakou mzdu.
  - Mapa deskové hry obsahuje dílky krajiny, jejich typ ovlivňuje rychlost pohybu jednotek.
  - Úkolník obsahuje úkoly, které je možné přidávat a které mohou být splněny.

[PPA] OOP – zapouzdření

Zapouzdření:

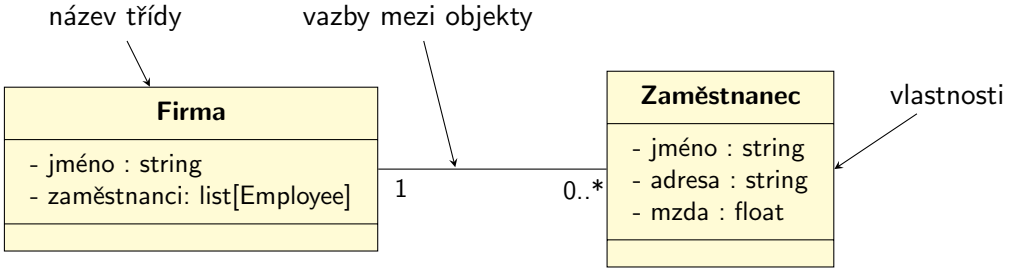
- data objektu jsou schována uvnitř,
- interakce probíhá pomocí metod nebo vlastností (properties),
- zajišťuje konzistenci objektu,
- možnost výměny konkrétní implementace (polymorfismus – dále).

Asociace / Kompozice / Agregace:

- vlastnosti objektů nemusí být jen primitivní datové typy,
- popisuje vztah mezi objekty,
- může klidně obsahovat vlastnost stejného typu (rekuzivní podstata).

[PPA] Zapouzdření – příklad

Firma má zaměstnance, každý má jméno a bere nějakou mzdu.



[PPA] Zapouzdření – příklad v Pythonu

```
1  class Employee:
2      def __init__(self, name:str, adresa:str, salary:float) -> None:
3          self.name = name
4          self.adresa = adresa
5          self.salary = salary

6  class Company:
7      def __init__(self, name: str) -> None:
8          self.name = name
9          self.employees:list[Employee] = []

10 company = Company("ZČU")
11 company.employees.append(Employee("Martin","Plzeň", 32564))
```

[PPA] Dědičnost

Dědičnost

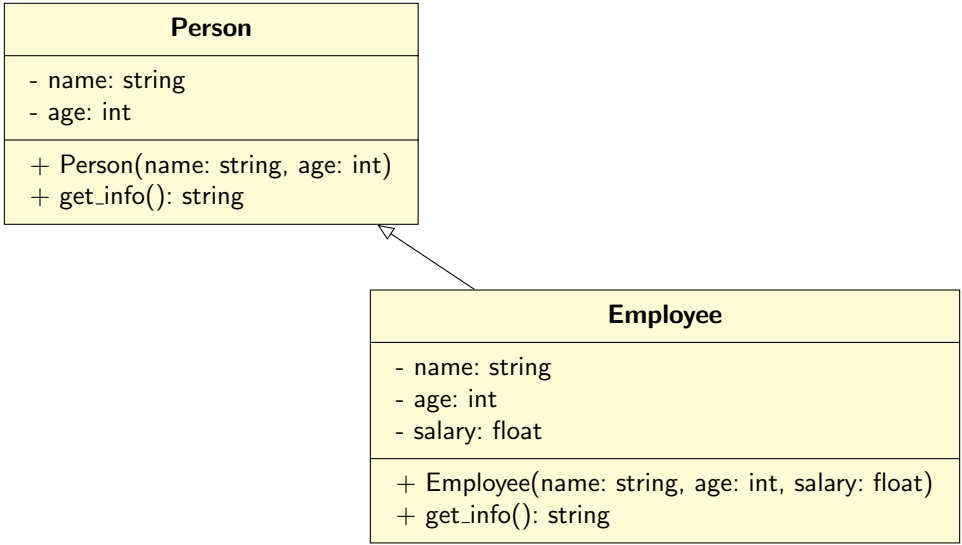
- Koncept rodičovské třídy a odvozené třídy.
- Redukuje duplicitní kód (menší náchylnosti k chybám).
- Odvozená třída:
  - přebírá strukturu (atributy) a chování (metody) rodičovské třídy.
  - může přidávat nové atributy,
  - může překrývat (upravovat) metody.
- V přetížené metodě lze volat metody rodiče klíčovým slovem `super()`

Dědičnost – příklad

```python
class Person:
    def __init__(self, name, age):
        if age < 0:
            raise ValueError("Věk musí být větší než 0.")
        self.name = name
        self.age = age
    def get_info(self):
        return f"{self.name} ({self.age})."

class Employee(Person):
    def __init__(self, name, age, salary):
        super().__init__(name, age) # rodičovský konstruktor
        self.salary = salary
    def get_info(self):
        return f"{self.name} ({self.age}) bere {self.salary} za rok."
```

Dědičnost – UML diagram



[PPA] Polymorfismus

Polymorfismus – schopnost objektů různých typů být vzájemně zaměnitelnými:

- umožňuje zacházet s odvozenou třídou jako s jejím rodičem,
- volají se odpovídající překryté metody,
- lze tak snadno pracovat s množinou rodičovských objektů bez nutnosti rozlišování oddděděných typů,
- lze používat pouze metody rodiče.
- Duck typing: S objekty se pracuje na základě vlastností konkrétní instance, nikoli na základě typu.

Definice (Duck typing)

"Pokud to chodí jako kachna a kváká to jako kachna, pak to musí být kachna."

[PPA] Polymorfismus – příklad

```
1 osoby: list[Person] = [Person("Jeníček", 5), Employee("Martin", 32,
  ↪ 500000)]

2 for osoba in osoby:
3     print(osoba.get_info())
```

Výstup:

```
Jeníček (5).
Martin (32) bere 500000 za rok.
```

Návrhové vzory

Co jsou návrhové vzory a proč jsou důležité?

- Návrhové vzory jsou obecné řešení často se vyskytujících problémů v softwarovém návrhu
- Například: jak vytvářet objekty, jak organizovat třídy a objekty, jak definovat interakce mezi objekty
- Používání návrhových vzorů má několik výhod:
  - Zvyšuje znovupoužitelnost a udržovatelnost kódu
  - Zlepšuje čitelnost a srozumitelnost kódu
  - Umožňuje lepší spolupráci mezi programátory

Typy návrhových vzorů

- Návrhové vzory se dělí do tří základních typů podle toho, jaký druh problému řeší:
  - Tvorba objektů (creational): řeší problémy spojené s vytvářením objektů
  - Organizace tříd a objektů (structural): řeší problémy spojené s uspořádáním tříd a objektů
  - Interakce mezi objekty (behavioral): řeší problémy spojené s definováním chování a komunikace mezi objekty
- V této prezentaci se zaměříme na některé z nejznámějších a nejužitečnějších návrhových vzorů z každého typu

Příklady návrhových vzorů

- Tvorba objektů (creational):
  - Jedináček (singleton): zajišťuje, že existuje pouze jedna instance dané třídy
  - Tovární metoda (factory method): umožňuje vytvářet objekty bez specifikace konkrétní třídy
  - Abstraktní továrna (abstract factory): umožňuje vytvářet rodiny souvisejících objektů bez specifikace konkrétních tříd

Příklady návrhových vzorů

- Tvorba objektů (creational):
  - Jedináček (singleton): zajišťuje, že existuje pouze jedna instance dané třídy
  - Tovární metoda (factory method): umožňuje vytvářet objekty bez specifikace konkrétní třídy
  - Abstraktní továrna (abstract factory): umožňuje vytvářet rodiny souvisejících objektů bez specifikace konkrétních tříd
- Organizace tříd a objektů (structural):
  - Adaptér (adapter): umožňuje spolupráci dvou nekompatibilních rozhraní
  - Most (bridge): odděluje abstrakci od implementace
  - Složenina (composite): umožňuje pracovat se skupinou objektů jako s jedním objektem

Příklady návrhových vzorů

- Tvorba objektů (creational):
  - Jedináček (singleton): zajišťuje, že existuje pouze jedna instance dané třídy
  - Tovární metoda (factory method): umožňuje vytvářet objekty bez specifikace konkrétní třídy
  - Abstraktní továrna (abstract factory): umožňuje vytvářet rodiny souvisejících objektů bez specifikace konkrétních tříd
- Organizace tříd a objektů (structural):
  - Adaptér (adapter): umožňuje spolupráci dvou nekompatibilních rozhraní
  - Most (bridge): odděluje abstrakci od implementace
  - Složenina (composite): umožňuje pracovat se skupinou objektů jako s jedním objektem
- Interakce mezi objekty (behavioral):

Příklady návrhových vzorů

- Tvorba objektů (creational):
  - Jedináček (singleton): zajišťuje, že existuje pouze jedna instance dané třídy
  - Tovární metoda (factory method): umožňuje vytvářet objekty bez specifikace konkrétní třídy
  - Abstraktní továrna (abstract factory): umožňuje vytvářet rodiny souvisejících objektů bez specifikace konkrétních tříd
- Organizace tříd a objektů (structural):
  - Adaptér (adapter): umožňuje spolupráci dvou nekompatibilních rozhraní
  - Most (bridge): odděluje abstrakci od implementace
  - Složenina (composite): umožňuje pracovat se skupinou objektů jako s jedním objektem
- Interakce mezi objekty (behavioral):
  - Pozorovatel (observer): umožňuje objektům být informovány o změnách v jiných objektech
  - Strategie (strategy): umožňuje definovat různé varianty algoritmu a měnit je za běhu
  - Šablona (template method): umožňuje definovat kostru algoritmu a nechat podtřídy implementovat detaily

Abstraktní třídy a metody v Pythonu 3

Abstraktní třída je třída, která definuje rozhraní, ale neimplementuje všechny metody.

Abstraktní metoda je metoda, která je deklarována, ale neimplementována v abstraktní třídě.

Abstraktní třídy a metody v Pythonu 3

Abstraktní třída je třída, která definuje rozhraní, ale neimplementuje všechny metody.

Abstraktní metoda je metoda, která je deklarována, ale neimplementována v abstraktní třídě.

- Abstraktní třídy a metody se používají pro návrhové vzory jako abstraktní továrna nebo šablona
- V Pythonu 3 se abstraktní třídy a metody vytvářejí pomocí modulu `abc` (Abstract Base Classes)

Abstraktní třídy a metody v Pythonu 3 – příklad

```
1  from abc import ABC, abstractmethod

2  class Animal(ABC):
3    @abstractmethod
4    def make_sound(self):
5      pass

6  class Dog(Animal):
7    def make_sound(self):
8      print("Woof")

9  class Cat(Animal):
10   def make_sound(self):
11     print("Meow")
```

Další konstrukce a výrazy pro OOP v Pythonu

- Atributy a metody třídy: Jsou to atributy a metody, které patří k třídě samotné, nikoli k žádné instanci. Definují se pomocí dekorátorů `@classmethod` a `@staticmethod`
- Vícenásobná dědičnost: Je to vlastnost, která umožňuje třídě dědit z více než jedné rodičovské třídy. To může být užitečné pro vytváření složitých hierarchií tříd, ale může také přinést některé problémy, jako je diamantový problém nebo pořadí vyhledávání metod.

Dekorátory v Pythonu 3

- Dekorátor je funkce, která přijímá jinou funkci jako argument a vrací upravenou verzi této funkce.
- Dekorátory se používají pro přidávání nebo měnění funkcionalit existujících funkcí bez změny jejich kódu.
- Dekorátory se zapisují pomocí symbolu @ před názvem dekorované funkce.
- Dekorátory mohou být také třídní metody, které přijímají třídu nebo podtřídu jako argument a vrací upravenou verzi této třídy nebo podtřídy
- V následující příkladu uvidíme, jak použít dekorátory pro implementaci návrhového vzoru tovární metoda, který umožňuje vytvářet objekty různých typů podle nějaké podmínky.

Metody třídy a dekorátory – příklad I

```
1  class Pizza:
2    def __init__(self, ingredients):
3      self.ingredients = ingredients
4    def __repr__(self):
5      return f"Pizza({self.ingredients})"
6    # Registr podtříd a jejich voleb
7    _registry = {}
8    # Metoda třídy pro registraci podtřídy a jejího jména volby
9    @classmethod
10   def register(cls, choice):
11     def decorator(subclass):
12       cls._registry[choice] = subclass # Přidat podtřídu do registru
13       return subclass # Vrátit podtřídu nezměněnou
14     return decorator
```

Metody třídy a dekorátory – příklad II

```
15  # Metoda třídy pro vytvoření pizzy podle volby
16  @classmethod
17  def make_pizza(cls, choice):
18    if choice in cls._registry: # Zkontrolovat, zda je možnost platná
19      subclass = cls._registry[choice] # Získat podtřídu z registru
20      return subclass() # Vytvořit a vrátit instanci podtřídy
21    else:
22      raise ValueError(f"Neplatná možnost: {choice}")
```

Metody třídy a dekorátory – příklad III

```
1  # Použije třídní metodu register jako dekorátor pro registraci podtříd a jejich možností
2  @Pizza.register("cheese")
3  class CheesePizza(Pizza):
4    def __init__(self):
5      super().__init__(["cheese", "tomato sauce"])

6  @Pizza.register("pepperoni")
7  class PepperoniPizza(Pizza):
8    def __init__(self):
9      super().__init__(["cheese", "tomato sauce", "pepperoni"])

10 @Pizza.register("veggie")
11 class VeggiePizza(Pizza):
12   def __init__(self):
13     super().__init__(["cheese", "tomato sauce", "mushrooms", "olives"])

14 choice = input("Jakou pizzu chcete? ")
15 pizza = Pizza.make_pizza(choice) # volání metody třídy pro vytvoření pizzy
16 print(pizza)
```

Stránky s návrhovými vzory

https://refactoring.guru/design-patterns/python