Patří sem rozhraní???

Principy objektově orientovaného programování

Vztah k ostatním paradigmatům

Každé programovací paradigma se snaží vyřešit problém sdíleného stavu programu.

- imperativní programování - výchozí způsob - stav je libovolně měnitelný a je sdílen v globálních a lokálních proměnných
- funkcionální programování - „žádný sdílený stav neexistuje“ - stav je neměnný, mezi tzv. „pure“ funkcemi bez vedlejších účinků se posílají kopie stavu
- objektové programování - stav je rozdělen na malé části a zapouzdřen logikou programu v objektech

To znamená, že konkrétní hodnoty jsou uloženy jako atributy těchto objektů. Zpracování atributů spolu s celkovou komunikací s objekty probíhá pomocí kódu, který je obsažený v metodách objektů. Navenek pak program působí jako několik navzájem spolupracujících objektů, což umožňuje snadnější přenos kódu mezi různými projekty a jednodušší úpravu již existujícího kódu pomocí dědičnosti. Další výhodou je také větší zabezpečení dat díky viditelnosti atributů.

Nevýhodou objektově orientovaného programování je jeho větší náročnost na paměť a výpočetní rychlost, proto se používá hlavně u moderních počítačů, kde jsou tyto nevýhody téměř neznatelné. U mikropočítačů a jednoduchých jednoúčelových strojů je však výhodnější použití strukturovaného programování.

Zapouzdření

Zapouzdření není výhradně koncept OOP. Obecně v programování znamená, že nějaké data nebo logiku schováme za restriktivním rozhraním.

V OOP je tento koncept implementován třídami, kde data jsou schována v podobě privátních atributů, které můžeme manipulovat pomocí public metod. Příkladem zapouzdření je například kolekce HashMap v jazyce Java (skrývá implementaci hašovací tabulky za metodami .put() nebo .getOrDefault()).

Dědičnost

Dědičnost umožňuje tvorbu nových tříd podle již vytvořených tříd. Tyto odvozené třídy sdědí všechny public a protected atributy a metody rodičovských tříd. Odvozené třídy si mohou tato data různě

upravovat nebo přidávat další.

např:

```
class Zpevak extends Clovek {

zpivej(){};

tancuj(){}; }
```

V tomto příkladu jsme vytvořili novou třídu *Zpevak*, která je odvozena od třídy *Clovek*. Objekt třídy *Zpevak* tedy může využívat všechny atributy a metody třídy *Clovek* (např. metoda *vstan()* a atribut *barvaOci*) a zároveň nově vytvořené metody *zpivej()* a *tancuj()*.

Podle kritiků by se dědičnost měla používat velmi vzácně, protože vytváří problém přílišné abstrakce a hluboké hiearchie dědičnosti, ve které člověk nepozná, kde je logika skutečně implementována. Z těchto důvodů existuje nepsané pravidlo, že dědičnost by měla být hluboká pouze 1 vrstvu. (dceřiná třída nesmí být rodičovskou třídou pro jiné třídy)

Na druhou stranu existují případy, kde je vícevrstvá dědičnost velmi výhodná - například při implementaci datových struktur ve standardní knihovně jazyku Java nebo ve vývoji her (hierarchie typu Entity → Character → Player).

Vícenásobná dědičnost

Počet rodičovských tříd je určen použitým programovacím jazykem (např. Java umožňuje pouze jednu rodičovskou třídu, C++ umožňuje více).

Polymorfismus

Umožňuje použít jednotné rozhraní pro práci s různými typy objektů.

Compile-time polymorfismus

Přetěžování metod - metoda může fungovat více různými způsoby, které se rozliší podle druhu a počtu parametrů. (funguje pouze v jazycích s)

Programovací jazyk se statickými typy ví, jaké datové typy používá, a proto umí vybrat správnou metodu při kompilaci.

Příkladem je třída PrintStream v jazyce Java (implementuje System.out.println()):

println()	Terminates the current line by writing the line separator string.
println(boolean x)	Prints a boolean and then terminate the line.
println(char x)	Prints a character and then terminate the line.
println(char[] x)	Prints an array of characters and then terminate the line.
println(double x)	Prints a double and then terminate the line.
println(float x)	Prints a float and then terminate the line.
println(int x)	Prints an integer and then terminate the line.
println(long x)	Prints a long and then terminate the line.
println(Object x)	Prints an Object and then terminate the line.
println(String x)	Prints a String and then terminate the line.

Runtime polymorfismus

Při běhu programu se musí zjistit, který datový typ používá a tím pádem kterou metodu volat.

2 způsoby implementace:

- alespoň 2 třídy mají stejnou rodičovskou třídu
- alespoň 2 třídy implementují stejné rozhraní

```
public class OsobaGML {
    public abstract void prezujSe();
}

public class StudentGML extends OsobaGML {
    @Override
    public void prezujSe() {
        System.out.println("Jdu ke skříňkám, otevřu svou skříňku, přezuji
se.")
    }
}

public class KantorGML extends OsobaGML {
    @Override
    public void prezujSe() {
        System.out.println("Jdu do svého kabinetu, přezuji se.")
    }
}

public static void main(String[] args){
    OsobaGML[] osoby = new {new KantorGML(), new StudentGML()};
    // Mám pole všech osob a chci, aby se všichni přezuli.
    for(osoba : osoby){
        osoba.prezujSe(); // Zde musí program rozlišit, jestli je konkrétní
osoba kantorem nebo studentem.
```

```
    }
}
```

Viditelnost atributů a metod

Nastavením viditelnosti dat můžeme určit, které části programu budou mít k těmto datům přístup. Nastavit můžeme tři základní možnosti.

- private - data jsou viditelná pouze pro konkrétní objekt
- public - data jsou viditelná komukoli
- protected - data jsou viditelná pouze pro konkrétní třídu a odvozené třídy

Rozhraní

Zjistíme, že programátor umí psát na počítači a účetní také. Intuitivně cítíme, že nebudou mít mnoho dalších společných schopností a navíc tuto dovednost může mít napříč povoláními leckdo, proto nemá smysl tvořit třídu na způsob ČlovekPracujícíSPočítačem a od ní dědit Programátora a Účetní, ale je výhodnější například vytvořit rozhraní SchopenPsátNaPočítači s požadavkem na metodu napišNaPočítači() a upravit třídy Programátor a Účetní tak, aby toto rozhraní implementovaly, tedy předepsanou metodu, a to každý po svém. V definici rozhraní nemůže být obsažen kód (implementace) dané metody, ale všechny třídy, které toto rozhraní implementují, musí být schopny se s příkazem napišNaPočítači() nějak vypořádat.

Obdobně může abstraktní třída předepisovat doimplementování metod, pro které ona sama nemá vlastní kód, ale jen předpis abstraktní metody.

Design patterns (návrhové vzory)

Navrhují konkrétní způsoby řešení častých problémů v OOP. Tyto způsoby byly poprvé popsány v knize Design Patterns v roce 1994.

Výhody:

- rychlost implementace (Nemusím vymyslet způsob řešení problému.)
- komunikace v týmu (Kolega rychle pochopí, protože taky dobře zná tyto návrhové vzory-)
- údržba kódu (Tyto vzory vytváří tzv. „loose coupling“. To znamená, že můžu nahradit jednu část systému, aniž bych rozbil všechno ostatní.)

Nevýhody:

- Často vedou ke příliš komplikovanému kódu, protože byly použity, aniž by byly potřeba.

Tady je seznam nejdůležitějších z každé kategorie:

Creational

Nabízí alternativní způsoby vytvoření objektů.

- Builder
- Singleton

Structural

Vysvětlují, jak uspořádat objekty do větších struktur.

- Adapter
- Decorator

Behavioral

Vysvětlují komunikaci mezi objekty.

- Iterator
- Observer