SOFTWAROVÉ INŽENÝRSTVÍ

OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Petr Horáček

2016

Obsah

1. Úvod

Objektově orientované programování je moderní způsob (paradigma) programování, který jehož cílem je rychlý vývoj rozsáhlých (složitých) programů. Objektově orientované programování, nebo také jen zkráceně OOP nahradilo procedurální programování, protože odstraňuje některé jeho nevýhody (nedostatky).

Program napsaný v procedurálním paradigmatu se skládá ze sady funkcí, které pracují s vnitřními daty (proměnnými) programu. Nevýhodou tohoto způsobu programování je, že data nejsou v žádném vztahu s použitými funkcemi (kromě datového typu). Díky tomu je možné na data volat nesprávné funkce, které mohou způsobovat chyby v programu.

Programu napsaný v objektově orientovaném způsobu programování se skládá z objektů, které obsahují vnitřní data programu a funkce (metody), které je možné nad nimi vykonávat. Tím je výrazně omezena možnost výskytu chyby způsobená nesprávně použitou funkcí. Díky tomu se celý program snáze (výrazněji) dělí na jednotlivé části, které jsou ve vzájemném vztahu a tvoří tak výsledný program.

OOP s sebou nese i nový způsob myšlení nad problematikou návrhu (programování) software. V případě procedurálního programování programátor přemýšlí o programu jako o sekvenci operací - funkcí, které jsou postupně volány. V případě OOP programátor o programu přemýšlí jako o napodobenině reálného objektu s nějakými vlastnostmi a schopnostmi.

Objektově orientované programování stojí na třech základních pilířích:

- Zapouzdření
- Dědičnost
- Polymorfizmus

2. Vnímání reálného světa

Principem objektově orientovaného programování je napodobení reálného světa. Svět se skládá z věcí, které jsou v souvislosti s OOP obecně nazývány **objekty**. Objektem může být cokoli co se vyznačuje určitými **vlastnostmi** (atributy) a **chováním**, například auto, dům, nebo i člověk. Chováním se rozumí nějaké činnosti, které daný objekt může vykonávat. Například objekt člověk dovede chodit, jíst, číst, ... Vlastnostmi objektu se myslí jeho charakteristické rysy. Příkladem vlastností objektu člověk je barva očí, počet zubů, tvar těla, ... Chování a vlastnosti objektů je vždy vzájemně propojeno v souvislí fungující prvek. Chování objektu totiž využívá jeho vlastností k tomu, aby mohl celý objekt jako celek správně fungovat.

Při programování na principu OOP je důležité správně přemýšlet o objektech - o jejich vlastnostech a chováním, aby bylo možné správně daný objekt modelovat.

Vlastnosti objektu v prostředí softwarového inženýrství jsou proměnné. Chování objektu v prostředí softwarového inženýrství jsou funkce, které s danými proměnnými pracuje.

Rozdíl mezi objektově orientovaným programování a procedurálním programování je ve způsobu přemýšlení nad daným problémem a návrhu výsledného řešení. Objektově orientované programování seskupuje věci kolem podstatných jmen - vstupní pole, formulář, tlačítko, ... které jsou známi jako objekty Procedurální programování naproti tomu seskupuje věci okolo sloves. Sloveso je ve zkutečném světě úkolem. To znamená, že procedurální programátor se zaměřuje na úkoly prováděné programem, zatímco objektově orientovaný programátor pracuje s objektem, jeho daty a chováním.

3. Třída

Pro vytvoření daného objektu je nejprve nutné popsat jeho chování a vlastnosti pomocí třídy. Třída slouží jako šablona, "stavební plány" popisující vlastnosti a chování objektů, které je možné podle ní vytvořit. Třída není objektem, ale určuje jak bude výsledný objekt vypadat. Podle definice třídy je možné vytvářet nekonečné množství objektů se stejným chováním a vlastnostmi.

Třída je popsána **definicí třídy**, která obsahuje atributy a metody. Definice třídy se skládá ze dvou částí:

- Hlavička třídy
- Tělo třídy

Hlavička třídy obsahuje především klíčové slovo konkrétního programovacího jazyka, který definuje třídu. Tím je slovo *class*. Následuje identifikátor třídy, který slouží pro další manipulaci se vzniklou datovou strukturou. Volitelně mohou ještě následovat různá další nastavení jako například dědění, specifikace přístupových práv, ...

Tělo třídy obsahuje dvě části:

- Atributy třídy
- Metody třídy

Vnitřně je základem třídy je struktura. Struktura je typ datové struktury, která může být složena z více proměnných různého (libovolného) datového typu. Třída navíc kromě proměnných různého datového typu může obsahovat funkce (metody), které s těmito proměnnými pracují.

3.1 Metody

3.1.1 Statické metody

3.2 Konstruktor a destruktor

Uvnitř každé třídy se nacházejí dvě speciální metody, které slouží k vytvoření objektu a k jeho zrušení.

3.3 Generika

3.4 Atributy

3.5 Instance třídy

Před tím, než je možné využít atributy a metody dané třídy je nutné vytvořit její instanci. **Instance třídy** nebo také **objekt** je proměnná, která je vytvořena podle definice dané třídy (předlohy). Všechny objekty vytvořené podle dané třídy obsahují stejné atributy a metody definované ve třídě, ale každý objekt má vlastní kopii těchto atributů. Na opak všechny objekty přistupují ke stejným metodám.

3.6 Ukazatel this

Objekt je vnitřně reprezentován atributy a překladač, respektive syntaktická pravidla daného programovacího jazyka jej následně sváže s metodami dané třídy → zapouzdření. Aby mohly metody pracovat s atributy daného objektu, je nutné jí předat ukazatel na tento objekt. Ukazatel na objekt, který daná metoda přijímá jako parametr se v OOP nazývá ukazatel this.

Ukazatel this je v OOP speciální ukazatel, který je automaticky předáván každé metodě dané třídy. Ukazatel this ukazuje na atributy a metody, respektive na instanci dané třídy a umožňuje tak zapouzdřit (propojit) celou třídu. Většina programovacích jazyků jej předává jako tzv. **skrytý parametr**. To znamená, že je není nutné vkládat do hlavičky deklarace dané metody a předávat jej jako parametr při jejím volání, protože se o to automaticky stará překladač při překladu zdrojových kódů.

```
void fce(void* this); //součást každého objektu
```

Volání dané metody `obj->fce()` pak vnitřně vypadá `fce(&obj)` (s nutným dodatečným přetypováním).

Použití ukazatele this při psaní programu:

```
this->atribut = value
```

Některé programovací jazyky (například C, částečně Python) však vyžadují manuální vkládání ukazatele this a odkazu na instanci dané třídy do hlavičky metod. Takové programovací jazyky mají omezenou kompatibilitu s objektově orientovaným programovacím paradigmatem a využívají jej na nízké úrovni.

3.7 Automatická práva paměti

Objekty jsou speciálním typem proměnných, které lze stejně jako

proměnné primitivních datových typů vytvářet staticky při spuštění programu, nebo dynamicky až za běhu programu. Staticky alokované proměnné jsou automaticky zrušeny při opuštění programového bloku ve kterém byly definovány, ale dynamicky alokované je vhodné zrušit v případě, že nejsou již používány, aby mohla být jejich paměť využita jinde. Při dlouhodobém běhu programu může nastat vlivem úniku paměti pád systému kvůli jejímu nedostatku. Existují dva přístupy jak paměť dynamicky alokovaných proměnných uvolnit.

V prvním případě si musí sám programátor hlídat, aby v programu neudělal chybu a veškerou alokovanou paměť musí v okamžik již není potřeba opět uvolnit. Tento způsob je velmi rychlý, ale s rizikem výskytu problému s paměti.

Druhý případ využívá automatickou správu paměti - **garbage collection**. K tomu se používá speciální program (algoritmus) nazývaný **garbage collector**, který běží souběžně se spuštěným programem a uvolňuje paměť, kterou již daný program nepoužívá. Garbage collection šetří čas při vývoji. Automatická správa paměti osvobozuje programátora od uvolňování objektů, které již dále nejsou zapotřebí, což ho většinou stojí nezanedbatelné úsilí. Garbage collection také pomáhá předcházet některým typům běhových chyb, které se často vyskytují při ruční správě paměti

Garbage collection rozpozná nevyužívanou paměť tak, že již na ni v programu nevede žádný odkaz, nebo je vyslán příkaz na její uvolnění (zrušení objektu). Garbage colector, ale může danou paměť uvolnit v náhodný okamžik. V případě, že je nutné uvolnit nevyužívanou paměť v přesně daný okamžik, je možné v programu na danou paměť zavolat funkci garbage colectoru, který zabranou paměť okamžitě uvolnění. Nevýhodou je mírné zvýšení výpočetní náročnosti programu, ale za cenu minimalizace rizika výskytu problému s pamětí (garbage colector zároveň hlídá narušení paměti, která danému procesu nepatří) a urychlení celkový vývoj programu.

3.7.2 Využití v počítačových jazycích

Garbage collector se obvykle implementuje jako část běhového prostředí (programovacího) jazyka (Java), nebo jako přídavná knihovna, podporovaná překladačem, hardwarem, operačním systémem nebo jejich kombinací (C#).

Pro programovací jazyky, které v základu nepodporují funkci automatické správy paměti (C, C++, ...) existují nadstavby ve formě kni-

hovních funkcí, které fungují obdobným způsobem.

3.8 Identifikátory v OOP

V OOP je nepsaným pravidlem používat určitý způsob psaní názvů identifikátorů. Názvy identifikátorů v OOP lze rozdělit na identifikátory objektů a identifikátory funkcí/metod.

Protože objekty představují nějaká podstatná jména, předměty, respektive názvy předmětů z reálného světa, například formulář, popisek nebo tlačítko. Jména objektů, ale i proměnných by tedy měly tvořit **podstatná jména**. V psaném textu se jména píší s **velkým počátečním písmenem** a proto se toto pravidlo využívá i identifikátorech ve zdrojovém kódu. Například:

```
object TextLabel("text")
```

Naopak funkce a metody vyjadřují nějakou činnost, kterou je třeba vykonat nad daty objektu. Jejich identifikátor by tedy mělo tvořit **sloveso**. Názvy činností se v běžně psaném textu píší malým počátečním písmenem a proto se toho využívá i identifikátorech funkcí a metod ve zdrojovém kódu programu. Například:

```
char readLine()
{
...
}
```

4. Zapouzdření

Zapouzdření je jedním ze základních pilířů OOP. Zapouzdření umožňuje skrýt některé atributy a metody tak, aby je bylo možné použít pouze uvnitř dané třídy. Objekt se pak navenek chová jako černá skříňka (black box), která definuje nějaké komunikační **rozhraní** (interface), skrze které je možné jej ovládat. Programátor neví (nemusí vědět) jak objekt vnitřně funguje, ale ví jak se navenek chová a používá. To minimalizuje riziko, že jiný programátor udělá nějakou chybu, protože objekt používá pouze tak, jak to zamýšlel jeho tvůrce. Zapouzdření tedy donutí programátory používat objekt jen tím správným způsobem. Rozhraní pak umožní rozdělit strukturu objektu na veřejně přístupnou část (private) a soukromou část (private). Zapouzdření umožňuje snáze zajistit integritu výsledného objektu (objekt nelze tak jednoduše ovlivnit nečekaným vstupem). Výsledkem je robustnější kód.

Rozdíl mezi třídou a strukturou je zapouzdření. Zapouzdření je způsob uchování atributů třídy a jejich metod. Nevýhodou procedurálního programování je, že data a funkce nemají mezi sebou žádnou souvislost. To umožňuje například na určitá data volat nesprávné funkce, nebo měnit hodnotu proměnné nesprávným způsobem a tak způsobit chybu v programu. To se projevuje především při programování v týmu.

V případě, že jsou data a funkce (metody) společně zapouzdřeny v objektu, programátor nemá na výběr než vybrat jednu z metod, které mohou s daty uvnitř objektu pracovat. Jedná se tedy o bezpečnostní prvek objektově orientovaného programování, který nedovolí tak snadno udělat chybu. Díky tomu programátor nemusí zdlouhavě studovat vnitřní fungování funkce (metody), to ale na druhou stranu může být někdy nevhodné (občas by se programátor měl zajímat jak používané funkce vnitřně funguji).

4.1 Specifikátory přístupu

Specifikátory přístupu jsou speciální klíčová slova, která v objektově orientovaném programovacím jazyce určují, které atributy a metody jsou v objektu přístupné z vnějšku. Jedná se o způsob implementace zapouzdření ve zdrojovém kódu a vytváření komunikačního rozhraní objektů. Důležité je, implementace (použití) specifikátorů přístupu závisí na použitém programovacím jazyce. Specifikace přístupových práv k prvkům objektu je nejčastěji realizováno na úrovni programovacího jazyka, to znamená, že příkazy specifikující přístup nejsou realizovány určitou programovou konstrukcí, ale jejich podpora je vestavěna přímo do překladače jazyka. Rozlišují tři základní specifikátory přístupu:

- **public** - specifikátor veřejného přístupu
- **private** - specifikátor soukromého přístupu
- **protected** - specifikátor chráněného přístupu

4.1.1 Specifikátor přístupu public

Specifikátoru veřejného přístupu **public** umožňuje definovat veřejně přístupné metody atributy a umožňuje tak vytvořit komunikačního rozhraní objektu. To znamená, že atributy a metody označené jako public je možné volat (přistupovat) vně třídy (skrze objekt).

4.1.2 Specifikátor přístupu private

Specifikátor soukromého přístupu **private** omezuje přístup k atributům a metodám pouze z vnitřku daného objektu (třídy). Soukromé (privátní) metody slouží pro interní použití, a není vhodné, nebo potřebné je volat z vnějšku. Jedná se tedy o způsob ochrany určitých částí objektu

před nesprávným (neoprávněným) použitím. Atributy a metody označené jako soukromé nejsou viditelné ani v odvozené třídě při dědění (viz. kapitola Dědění).

4.1.3 Specifikátor přístupu protected

Specifikátor chráněného přístupu **protected** je kombinací specifikátoru public a private. Jeho využití je především v případě tříd vzniklých děděním. Specifikátor chráněného přístupu omezuje přístup k atributům a metodám pouze z vnitřku objektu stejně jako v případě specifikátoru private, ale jsou přístupné v odvozené třídě při dědění. Díky tomu je možné některé atributy a metody označit jako soukromé zvenku, ale není nutné je v odvozené třídě duplikovat.

4.2 Vnitřní reprezentace zapouzdření

Každý objekt vytvořený podle stejné třídy disponuje svými atributy, ale každý objekt přistupuje ke společným metodám, které jsou uloženy v programové části paměti. Přístup k těmto metodám je v programovacích jazycích řešen dvěma způsoby. Prvním způsobem je na úrovni jazyka, který spolupracuje vývojovým prostředím. Metody jsou fyzicky tvořeny obyčejnými funkcemi, které jsou ve vývojovém prostředí spojeny s daným objektem. Druhým způsobem je pomocí ukazatelů na funkce, které jsou uloženy v daném objektu. Tyto funkce/metody jsou následně volány skrze tyto paměťové ukazatele.

Metoda se od funkce dále liší tím, že má vždy explicitně zadaný parametr - odkaz na objekt, pro který je daná metoda určena. V některých jazycích je tento parametr automaticky vkládán do volané metody, ale u jiných jazyků je nutné tento parametr ručně zadávat.

5. Polymorfismus

Název metody a seznam jejich parametrů se souhrnně nazývá **signatura metody**. Signatura metody jednoznačně identifikuje danou metodu v programu. **Polymorfizmus** nebo také **mnohotvarost** vyjadřuje schopnost programovacího jazyka definovat dvě a více funkcí (metod) se stejným identifikátorem v různých částech programu.

Polymorfizmus se používá pro případy, kdy je třeba vytvořit několik metod, které dělají stejnou věc, ale každá jiným způsobem, například nad jiným typem dat, nebo na jiné hardwarové platformě. Dík tomu lze definovat jedno softwarové rozhraní, které je možné používat stejným způsobem. Kdyby nebyl použit polymorfizmus, bylo by nutné vytvořit a používat mnoho metod s různými jmény a programátoři by si museli pamatovat, kterou z nich mají v daném případě použít.

Polymorfizmus je realizován díky schopnosti programovacího jazyka skrýt metody jiným částem programu. Díky tomu je možné, aby překladač rozeznal, kde je která funkce volána a nastavil tak správné paměťové adresy. Typickým příkladem jsou stejnojmenné metody v různých třídách. Metody nelze použít dokud není podle nich vytvořen objekt (jsou skryty před zbytkem zdrojového kódu) a daná metoda je následně díky zapouzdření spojena s daným objektem. Objekt je následně při volání použit jako vodítko ke správné metodě.

```
class A{
  show(){
    ...
  }
}

class B{
  show(){
    ...
  }
}

main(){
  A objA()
  B objB()

  objA.show()
```

```
  objB.show()
}
```

Metody, respektive funkce jsou ve spuštěném programu volány pomocí paměťové adresy. Z toho vyplývá, že polymorfizmus je řešen čistě na straně překladače, který musí rozeznat výskyt polymorfizmu ve zdrojovém kódu programu a rozlišit dvě různé metody, pro které musí vygenerovat strojový kód (nebo je jiným způsobem interpretovat). To znamená, že polymorfizmus z větší části nelze řešit programovými konstrukcemi, ale je nutná podpora programovacího jazyka.

5.1 Přetížení

Přetížení souvisí s polymorfismem, respektive jedná se o speciální případ polymorfizmu. **Přetížení** funkcí, nebo metod znamená, že v jedné části zdrojového kódu programu je možné vytvořit více funkcí nebo metod se stejným identifikátore. Tyto metody jsou překladačem rozlišovány pouze na základě návratové hodnoty a seznamu vstupních parametrů. Může se jednat například jiný datový typ návratové hodnoty, jiný počet parametrů, jiný datový typ některého z parametrů, ...

Přetížení se využívá pro vytvoření jednotného rozhraní třídy respektive objektu, který může pracovat s různými daty, například ukládání hodnot do pole, jehož datový typ je určen až při vytváření objektu. Díky tomu celé rozhraní výrazně jednodušší na použití.

```
class cls{
  show(int a){
    ...
  }

  show(double a){
    ...
  }
}

main(){
  cls obj()

  obj.show(12)
  obj.show(3.14)
```

}

5.2 Rozhraní

Rozhraní v objektově orientovaném programování je prostředek, kterým je možné komunikovat s objektem, měnit jeho vnitřní stav a různě jej ovládat. V případě rozsáhlých tříd, může být komunikační rozhraní velmi rozsáhlé. Aby nemohlo dojít k tomu, že bude volána nějaká metoda, která v rozhraní objektu není definovaná (je pouze deklarována hlavička metody), je vhodné vytvořit deklaraci rozhraní, které zajistí, že všechny metody uvedené v deklaraci rozhraní musejí být také definovány v těle třídy, jinak program skončí chybou. **Deklarace rozhraní** obsahuje všechny veřejně přístupné metody objektu. Standardní rozhraní objektu definuje **název metody**, **seznam vstupních parametrů**, **návratové hodnoty metody** a **chování**.

5.2.1 Vazby

Stejné rozhraní může být definováno pro více tříd a následně je podle nastavené **vazby** nastavena cesta ke správnému objektu. To je výhodné v případě vysoce konfigurovatelných systémů, které dělají stejnou věc, ale různými způsoby.

K vytvoření vazeb dochází buď v době překladu a nebo za běhu programu. Vazba v době překladu se nazývá *časná vazba*, která je provedena v případě, že jsou při překladu programu známé všechny informace potřebné k volání dané metody. Vazby vznikající za běhu programu se nazývají *pozdní vazby* a jsou provedeny v pokud při překladu chybí určité informace, které budou známy až po spuštění programu.

Časné vazby se používají pro normální volání metod. Při jejich volání tak nedochází ke ztrátě času. Pozdní vazba je implementována prostřednictvím virtuální metody. Pozdní vazby potřebují nějaký čas při běhu programu, než vytvoří vazbu pro volání dané metody, ale díky tomu mohou v programu reagovat na události, které nastanou až za běhu programu (není nutné psát kód pro nečekané situace, ke kterým může za běhu programu dojít).

6. Dědičnost

Dědičnost je vlastností objektově orientovaného programování, která umožňuje jedné třídě (potomek) využívat atributů a metod třídy jiné (rodič). Díky tomu je urychlen vývoj programů a omezena délka kódu, který nutné navrhnout, napsat a otestovat. Jedná se o napodobení biologického procesu dědění, kdy potomek zdědí geny (vlastnosti a schopnosti) od jeho rodičů.

Dědičnost se používá vždy v případě, když data a chování používají dva nebo více objektů. Vlastnosti nadtřídy poté zdědí další třídy, které vyžadují totožná data a chování (plus něco navíc, čím se navzájem odlišují). Chování definované v nadtřídě se označuje jako **výchozí chování**, protože z něj vychází chování všech odvozených tříd.

6.1 Hierarchie tříd

V hierarchickém vztahu mezi třídami (*relace*) rodič-potomek zdědí potomek všechny atributy a metody rodiče. Rodič používá specifikátory přístupu k řízení dostupnosti zděděných položek pro jiné třídy.

Třída, která dědí vlastnosti od jiné třídy se nejčastěji nazývá **dceřinná třída**, **odvozená třída**, **podtřída** nebo **potomek**. Třída, která slouží jako vzor pro dědění se nejčastěji nazývá **rodičovská třída**, **základní třída**, **bázová třída**, **nadtřída** nebo **super třída**.

Při dědění by se neměly využívat více než tři vrstvy v hierarchii tříd. Jedná o praktický význam usnadňující následnou údržbu a aktualizaci kódu. Nadměrná dědičnost výrazně zvyšuje složitost celé architektury softwaru a ztěžuje následné modifikace jinými programátory. Z tohoto důvodu by se měla udržovat hranice tří vrstev hierarchie tříd pokud přidání dalších vrstev nepřinese nějaký významný přínos (řešení složitého problému, logická struktura jinak komplikované architektury).

6.2 Typy dědičnosti

Dědičnost lze do programu implementovat ve dvou odlišných podobách:

- **Jednoduchá dědičnost**
- **Vícenásobná dědičnost**

6.2.1 Jednoduchá dědičnost

Jednoduchá dědičnost využívá jednoho vztahu rodič-potomek. To znamená, že jeden potomek dědí vlastnosti pouze od jednoho rodiče.

Dědění probíhá směrem od rodiče k potomkovi. Rodičovská třída nemůže přistupovat k atributům a metodám svého potomka.

6.2.2 Vícenásobná dědičnost

Při vícenásobné dědičnosti zahrnuje relace více zdrojů a jednoho potomka. Potomek obsahuje atributy a metody každé mateřské třídy. Vícenásobná dědičnost je v základu pouze opakované jednoduchá dědičnost kdy potomek zdědí vlastnosti od rodiče a následně se sám stane rodičem. Některé programovací jazyky podporují vícenásobnou dědičnost v jednom příkazu uvedením všech mateřských tříd. U ostatních programovacích je nutné vytvořit několik tříd, které postupně prostupují stromem dědičnosti.

6.3 Volání konstruktoru bázové třídy

6.4 Abstraktní a finální třída

6.5 Překrytí metody

6.6 Virtuální metody

Slovo virtuální vyjadřuje, že se něco chová jako zkutečné i když to zkutečné není. V případě virtuální metody si počítač myslí, že je definována nějaká metoda, která ale v daném okamžiku nemusí existovat. **Virtuální metoda** tedy může být zkutečná metoda a nebo může sloužit jako zástupný znak (bod) pro zkutečnou funkci, která je následně definována po spuštění programu. Toho se využívá v případě pozdních vazeb v rozhraní, které je součástí polymorfizmu. Virtuální metody tvoří tzv. **polymorfizmus za běhu programu**. Třídě, která definovala virtuální metodu říká **polymorfická třída**.

7. Výjimky