Dědičnost

Tomáš Hruška

Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky VUT v Brně, Božetěchova 2, 612 66 Brno, Česká republika

Abstrakt

Dědičnost je jednou ze základních vlastností tříd v objektově orientovaných jazycích. V úvodu článku jsou vymezeny základní termíny. Postupným rozšiřováním pojmu dědičnosti, které vyplývá z analýzy řešených problémů a často i z potřeby efektivní implementace, jsou postupně zaváděny další rysy dědičnosti až k netriviálním schématům: *vícenásobné dědičnosti* a *tříd s virtuální bází* a jejich kombinací.[1]

1. Vymezení pojmů

Dědičnost zavádíme jako binární relaci mezi třídami. *Třída* v objektově orientovaných jazycích je kartézský součin

$$D1 \times D2 \times D3 \times \ldots \ldots \times Dm$$

Položky shora uvedeného kartézského součinu Di jsou *pojmenovanými datovými typy* (datový typ chápeme jako množinu hodnot spolu s operacemi prováděnými nad těmito hodnotami).

V jazyce může být obvykle definováno uživatelem libovolné množství tříd s obecně různými položkami. Pokud dvě třídy mají následující strukturu

$$D1 \times D2 \times D3 \times \ldots \ldots \times Dm \qquad \text{a}$$
$$D1 \times D2 \times D3 \times \ldots \ldots \times Dm \times Dm+1 \times Dm+2 \times \ldots Dn$$

můžeme definovat, že první a druhá třída jsou v relaci dědičnosti (zkráceně, že druhá třída je *zděděna* z první). První ze tříd vstupujících do dědičnosti nazýváme *nadtřídou* a druhou *podtřídou*. Podtřída obsahuje tedy všechny položky (pojmenované datové typy) nadtřídy ve stejném pořadí, se stejnými jmény a případně některé další.

Prvek třídy je uspořádaná n-tice pojmenovaných hodnot datových typů, které se vyskytují jako položky v třídě. Takovýto prvek se obvykle nazývá *objekt* a je základním stavebním kamenem programů zapisovaných v objektově orientovaných jazycích.

[1] Tato práce je podporována grantem GAČR 102/96/0986 Objektově orientovaný databázový model a grantem GAČR 102/94/1097 Metodologie vývoje informačních systémů

K popisu tříd a objektů budeme v rámci tohoto článku využívat notace jazyka C++, která je obvyklá. Třídu budeme zapisovat jako

```
class A : B {
    int i;
    void f(int);
};
```

Třída *A* má nadtřídu *B*. Jejími položkami jsou všechny položky třídy *B* a navíc položka *i* celočíselného datového typu a položka *f* datového typu funkce.

V následujících odstavcích jsou prováděny úvahy o zefektivnění některých operací s objekty. Předpokládáme, že přeložený kód je generován přímo pro cílový počítač, tj. použití generačního překladače obvyklého např. pro C++. Nepředpokládáme přímou interpretaci programů, která by za cenu větší režie výpočtu některé problémy mohla značně zjednodušit.

2. Dědičnost bez metod

Pokud datovým typem může být i typ funkce (tj. proveditelný program), např. s operacemi *kompiluj*, *spusť* a pod., docházíme k základnímu schématu dědičnosti. Každý objekt obsahuje všechny údaje i všechny operace, které je možné s těmito údaji provést. Výhodou této varianty je schopnost reprezentovat pro každý objekt vlastní operace, např. každý objekt třídy graf může být vykreslován jiným způsobem.

Nevýhodnou této varianty je nutnost ukládat v objektech hodnoty datového typu funkce. To je jednak paměťově náročné a navíc reprezentace proměnných datového typu funkce není v tradičních jazycích obvyklá a podporovaná operačním systémem. Funkce jsou obvykle přeloženy, sestaveny a nelze je během činnosti programu měnit (jsou v programu konstantní). Proto první úpravou základního schématu dědičnosti bývá omezení proměnnosti funkce. Funkce jsou přiřazeny během činnosti programu ke konstantním objektům, tj. *ke třídám*.

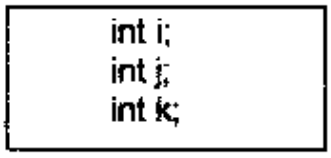
Na druhé straně nemusí tento typ dědičnosti být zcela bez významu tam, kde je nezbytné datový typ funkce užít. Např. u databází provozovaných v heterogenním prostředí více počítačů může být vhodné užít počítačově nezávislou reprezentaci funkcí. Ovládací jazyk může se zavedením datového typu funkce nabýt funkcionálních rysů, s nimiž lze algoritmy definovat deklarativně.

3. Dědičnost s metodami

Jsou-li funkce konstantní vzhledem ke všem objektům dané třídy, objektům všech jejich podtříd a navíc konstantní během provádění programu, nemá smysl je ukládat přímo do objektů. Mohou být uloženy mimo objekty jako součást přeloženého programu. Funkcím těchto vlastností budeme jim říkat *metody* dané třídy. Objekt pak obsahuje pouze datové složky. Podtřída obsahuje na svém začátku všechny datové položky nadtřídy, za něž jsou přidány všechny datové položky přidané v této podtřídě.
Nechť existuje např. následující posloupnost definicí tříd

```
class A {
        int i;
        void f(int);
};

class B : A {
        int j;
        void g(int);
};

class C : B {
        int k;
        void h(int);
};
```

Nejobvyklejší uložení objektu třídy C v paměti bude následující:



4. Virtuální metody

Neměnnost funkcí vzhledem ke všem objektům dané třídy i všem objektům podtříd bývá často omezující. Na druhé straně proměnnost funkci objekt od objektu během výpočtu nesnadno realizovatelná. Mezi těmito dvěma extrémy existuje možnost učinit funkci konstantní z hlediska jediné třídy. Od předchozího typu jednoduché dědičnosti se tedy virtuální metody liší tím, že metoda není povinně konstantní vzhledem k třídě a všem jejím podtřídám, nýbrž je možné pro každou z podtříd hodnotu metody změnit.

Navíc zde vystupuje jedna z dalších vlastností objektově orientovaných programů, tj. *zasílání zpráv*. Metody objektu jsou aktivovány zasláním zprávy a až přijímající objekt podle své třídy rozhodne o konkrétním algoritmu.

Tuto variantu lze ještě zvládnout bez uložení funkcí přímo v objektech. Vyžaduje však již jistou informaci o metodách uloženou přímo v objektu. Navíc jak uvidíme, vynucuje si tento způsob při užití generačního překladu jistou nelogičnost definice metod v tom, že je nutné v nadtřídě předem označit ty metody, které mohou být v podtřídách v budoucnu předefinovány. To je nepraktické, chceme-li dědičnostní strom tříd poskytnout uživatelům např. jako podklad pro další vývoj programu, přičemž předem nevíme, kterou z metod bude uživatel v budoucnu předefinovat. Označení předefinovávaných metod předem (např. slovem *virtual*) je vynuceno, jak uvidíme, implementací.

Problém implementace virtuálních metod (a jak uvidíme i vícenásobné dědičnosti) je dán *typovou kompatibilitou*. Objekt dané třídy je typově kompatibilní se všemi objekty dané třídy, ale rovněž s libovolným objektem všech nadtříd dané třídy. Máme-li tedy deklarován např. ukazatel na objekt jisté třídy, může tento ukazatel ukazovat nejen na objekt třídy, s níž byl explicitně svázán v deklaraci, ale i na objekt libovolné nadtřídy tohoto objektu. A

v nadtřídách (podle předchozího popisu virtuální metody) může daná virtuální metoda mít různé významy.

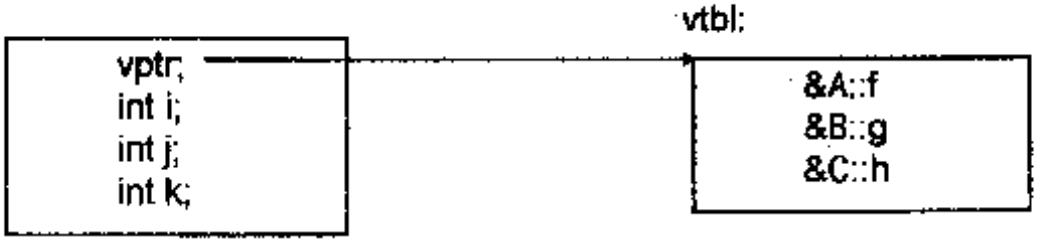
Při implementaci jsou proto všechny virtuální metody zpřístupněny tzv. *tabulkou virtuálních metod*, která může být pro každou třídu jiná. Nechť existuje následující úsek definice tříd:

```
class A {
        int i;
        virtual void f(int);
        virtual void g(int);
        virtual void h(int);
};

class B : A {
        int j;
        void g(int);
};

class C : B {
        int k;
        void h(int);
};

C* ukazatelc – new C;
```

Možné uložení objektu *C* v paměti je uvedeno na následujícím obrázku. Každá ze tříd, obsahujících virtuální metody, má definovánu tabulku virtuálních metod nazvanou *vtbl*. Tabulka obsahuje aktivační adresy všech virtuálních metod dané třídy, které mohou být uloženy v přeloženém kódu programu. Pokud v procesu dědění přibývají nové virtuální metody, jsou jejich adresy přidávány na konec tabulky. Adresa každé virtuální metody má tímto způsobem určenu neměnnou pozici v tabulce pro všechny podtřídy. Metody neoznačené jako virtuální mají pod svým jménem neměnný význam, proto je jejich uložení do tabulky zbytečné.

Aby bylo možné takovou tabulku vytvořit, musí být všechny virtuální metody předem (tj. v nejvyšší nadtřídě, v níž byly definovány) označeny. Na pevném místě v objektu leží pak ukazatel na tuto tabulku. Má-li být aktivována virtuální metoda, provádí se přístup k ní nepřímo přes ukazatel na objekt a tuto tabulku.

Na následujícím obrázku je zobrazeno uložení objektu třídy *C* v paměti. Tabulka virtuálních metod *vtbl* může být jediná pro celou třídu objektů.



Volání virtuální metody *g* s parametrem *p* zapsané v programu jako

```
ukazatelc -> g(p);
```

bude převedeno na

```
(*(ukazatelc -> vptr [1])) (ukazatelc,p)
```

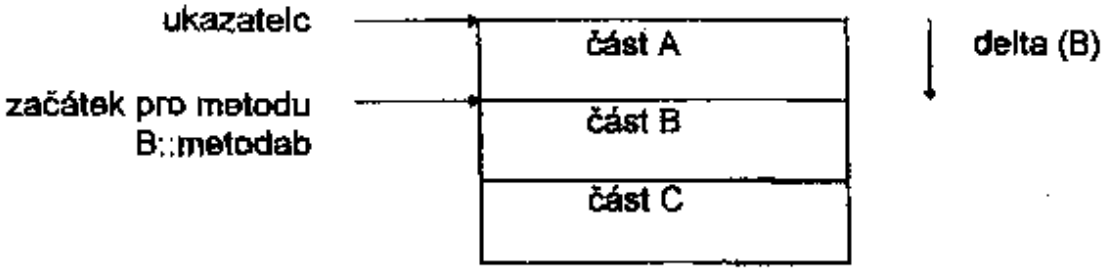
Vlastní volání funkce má dva parametry, neboť prvním parametrem metody musí být vždy ukazatel na objekt, k němuž je vázána (často označovaný v programu jako *this*). Druhým parametrem volání je skutečný parametr virtuální metody. Tabulka *vptr* je jako pole v jazyce C indexována od 0, proto má druhá metoda v pořadí index 1.

5. Vícenásobná dědičnost

Dalším požadavkem komplikujícím implementaci je zrušení binárnosti relace dědičnosti. Jinými slovy to znamená, že třída může mít více nežli jednu přímou nadtřídu. Tento požadavek není neobvyklý. Velmi praktickou programátorskou technikou může být definice skupiny tříd popisujících jisté vlastnosti budoucích objektů (např. u dokladů v ekonomickém informačním systému vlastnosti z hlediska DPH, styku s bankou, zúčtování atd.). Každá taková vlastnost je popsána v jedné třídě. U těchto tříd se nepředpokládá budoucí výskyt objektů. Jsou definovány jako *abstraktní třídy* a popisují různé pohledy na budoucí *konkrétní* objekty. Ty pak jsou kombinacemi takto definovaných vlastností, ale přirozeně ne pouze jediné vlastnosti (např. faktura může být daňovým dokladem z hlediska DPH, je zúčtovatelná, je placena přes banku atd.). Nejen z hlediska těchto technik je evidentní požadavek dědění z více nežli jediné nadtřídy nazvaný *vícenásobná dědičnost*.

```
class A {
      void metodaa(int);
};

class B {
      void metodab(int);
};

class C : A, B {
      };
```

Protože třída *C* je bezprostředně zděděna z více nežli jedné nadtřídy (zde konkrétně ze dvou nadtříd *A* a *B*), není již nadále jednoznačně zaručeno takové pořadí položek kartézského součinu, že nové položky jsou vždy přidávány za položky původní. Zdrojů je totiž více a je nutné se rozhodnout pro jedno z možných pořadí. Znovu zde vystupuje zejména požadavek kompatibility typů. Předpokládejme, že položky nadtříd jsou v objektu třídy *C* uloženy v pořadí *A*, *B*, *C*. Poznamenejme, že ani uložení jiné (např. *B*, *A*, *C*) následující problém neřeší jednodušeji



Dokud jsme nezavedli vícenásobnou dědičnost, byla adresa začátku všech objektů dané třídy i všech jejich podtříd vždy na začátku první části objektu. Nyní se toto pravidlo mění. Předpokládejme část programu

C ukazatelc = new C;*
ukazatelc->metodab(p);

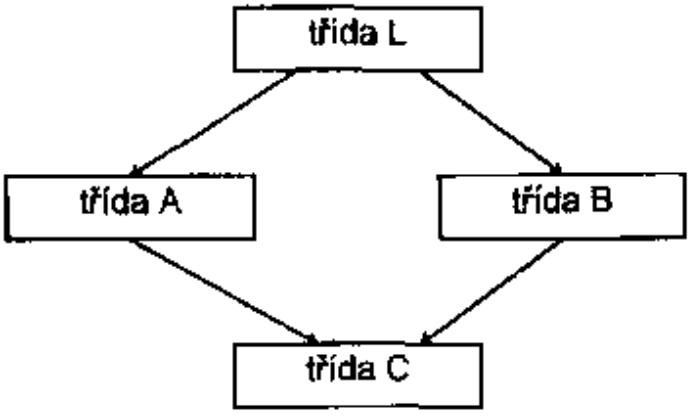
Metodab je definována v třídě *B*. Je přeložena tak, že předpokládá začátek objektu na začátku části *B* a jiné části nezná. Objekt třídy *C* je však typově kompatibilní s objekty třídy *B*. Proto musí být metody třídy *B* aplikovatelné i na objekty třídy *C*, i když zavedením vícenásobné dědičnosti již neleží část zděděná z nadtřídy nutně na začátku objektu podtřídy. Vystupuje zde jistá vzdálenost *delta(B)*, což je vzdálenost, o kterou musí být ukazatel na objekt třídy *C* změněn před voláním metody třídy B *metodab*. Tato hodnota je známa v čase překladu. Uvedené volání metody *ukazatelc->metodab(p);* bude proto převedeno na následující příkaz

((B) ((char*) ukazatelc + delta(B))) -> metodab(p);*

Poznamenejme, že doposud neuvažujeme kombinace diskutovaných problémů. Nejde tedy o virtuální metody a není nutné užívat převodní tabulku. Pro neprogramátory v jazyce C poznamenejme, že je nejdříve *ukazatelc* přetypován na ukazatel na znak, čímž se zajistí, aby ukazatelová aritmetika počítala v bytech. Potom je ukazatelovou aritmetikou přičtena celočíselná hodnota *delta(B)* uvažovaná rovněž v bytech. Nakonec je výsledek přetypován na ukazatel na objekt třídy B. Tím je umožněno volat metodu *metodab* třídy *B*.

6. Virtuální báze

Vícenásobná dědičnost vytváří další problém. Graf následnosti tříd přestal být stromem a stal se pouze acyklickým grafem. Představme si následující graf relace dědičnosti:



Zapsán textově by měl předchozí graf dědičnosti následující tvar:

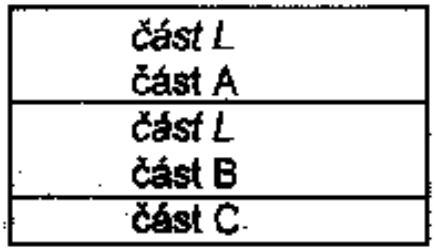
class L { ... };

class A: L { ... };

class B: L { ... };

class C : A, B { ... };

Objekt třídy *C* takto deklarovaný bude pak obsahovat část zděděnou z třídy *L* dvakrát, neboť z *L* do *C* vedou v grafu dědičnosti dvě cesty.



To v mnoha případech nevyhovuje. Obzvláště při použití abstraktních tříd je požadováno, aby ve výsledném objektu byly zdrojové části pouze jednou. Proto byl zaveden nový typ dědičnosti nazvaný *třídy s virtuální bází*. Dědíme-li část, které se má v budoucnu vyskytovat v objektech pouze jednou, i když bude zděděna více cestami, označíme ji při jejím prvním dědění atributem *virtual*. Např.

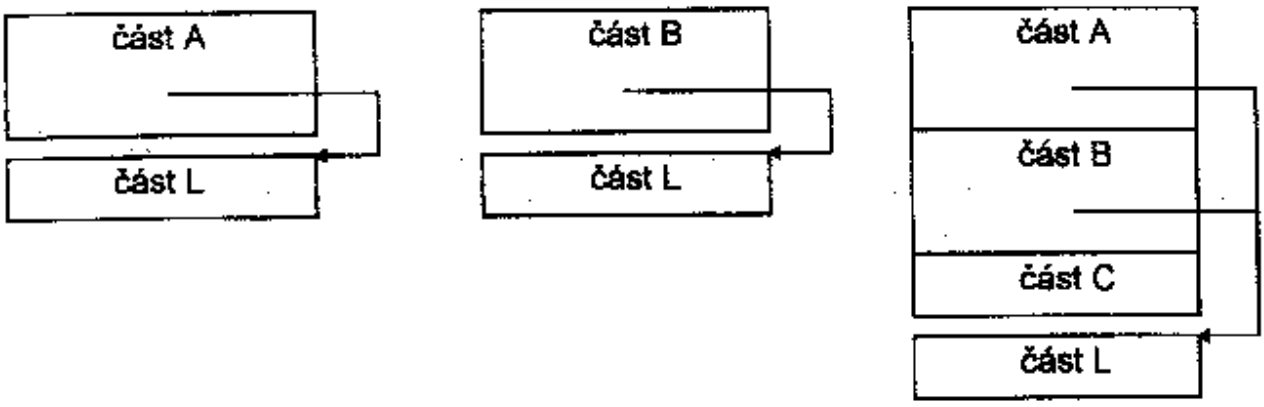
class L { ... };

class A: virtual L { ... };

class B: virtual L { ... };

class C : A, B { ... };

Jak uvidíme, opět zde z implementačních důvodů poměrně nelogicky předem určujeme, které z částí budou v budoucím dědění zpracovány tak či onak. Logické místo, kde bychom tuto vlastnost chtěli deklarovat je až konečný objekt, v němž se spojuje více větví vícenásobné dědičnosti. Implementační problémy nám v tom však zabraňují. Jsme nuceni určovat vlastnost virtuální báze jako vlastnost prvního dědění. Objekty třídy *A*, *B* a *C* budou mít následující strukturu:



Problém je tedy řešen podobně jako u virtuálních metod. Označíme-li dědění jako virtuální bázi, vytvoří se v podtřídě namísto samotných položek pomocný ukazatel na virtuálně děděnou část. Za cenu nepřímého odkazu si do budoucna připravíme půdu pro to, aby v případě vícenásobného dědění téže části mohlo více těchto pomocných ukazatelů nepřímo odkazovat tutéž část. Připomeňme, že virtualita báze v této implementaci není vlastností třídy *L*, nýbrž procesu dědění.

7. Kombinace

Předchozí typy dědičnosti nevystupují samozřejmě pouze samostatně. Kombinací uvedených typů dědičností je více. Vzhledem k rozsahu článku diskutujme zajímavou kombinaci vícenásobné dědičnosti a virtuálních metod.

7.1 Vícenásobná dědičnost a virtuální metody

Uvažujme např. následující definici tříd (příklad je převzat z [1]):

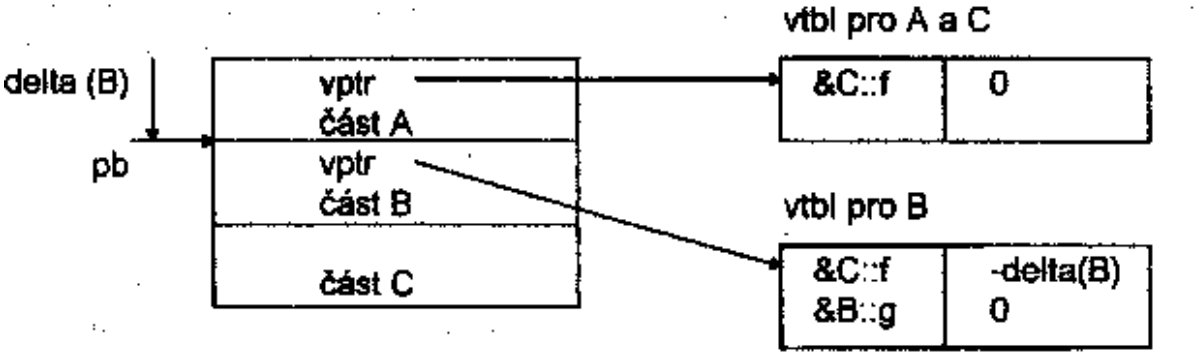
```
class A {
        virtual void f( );
};

class B {
        virtual void f( );
        virtual void g( );
};

class C : A, B {
        void f( );
        };

C* pc = new C;
B* pb = pc;

pb->f( );
```

Poslední příkaz bude aktivovat metodu *C::f*. Příkaz pro převod ukazatele *pc* na ukazatel *pb* posune hodnotu ukazatele o *+delta(B)*, aby ukazatel skutečně zpřístupňoval část *B* objektu. Jedná-li se o metodu virtuální, musí však být předchozí mechanismus posunu hodnoty ukazatele eliminován. Protože eliminace má proběhnout pouze u virtuálních metod, je nutné rozšířit tabulku virtuálních metod o hodnotu, která bude při aktivaci eliminovat předchozí posun ukazatelů na objekty nadtříd. Položka tabulky virtuálních metod bude mít proto navíc položku *delta* s touto hodnotou.

```
struct vtbl_polozka {
        void (*fct) ();
        int delta;
}
```

Uložení objektu třídy *C* v paměti je zobrazeno na následujícím obrázku. Volání metody *pb->f()* bude provedeno těmito příkazy

```
register vtbl_entry* vt = &pb -> vtbl[0];
(*vt->fct) ((B*) ((char*) pb+vt->delta));
```



Do proměnné *vt* je přiřazen ukazatel na příslušnou položku tabulky virtuálních metod. Adresa metody *vt->fct* je modifikována hodnotou *vt->delta*. Ukazatel pb musí být nejdříve přetypován na typ *(char*)*, aby ukazatelová aritmetika ve sčítání *pb+vt->delta* počítala v jednotkách bytů.

8. Závěr

Z důvodu nedostatku místa nebylo možno diskutovat kombinaci virtuální báze a virtuálních metod. Tuto i další kombinace lze nalézt např. v [1].
Všechny předchozí implementace jsou navíc založeny na tom, že objekty neobsahují žádné služební informace. To je možné u nepersistentních objektů C++, ale nikoliv u persistentních objektů užívaných v databázových aplikacích (informace o identifikaci, typu atd.). Zde nelze libovolně posouvat ukazatele, neboť služební informace leží vždy na jistém místě objektu pouze jednou. Řešení tohoto netriviálního problému je součástí projektu GAČR 102/96/0986 *Objektově orientovaný databázový model*. Tento přehled lze považovat za příspěvek k jeho studijní etapě.

Literatura:

1. Ellis,M.,A.-Stroustrub,B.: The Annotated C++ Reference Manual, Addison-Wesley Publishing Company 1990, 453 stran

2. Bertino,E.-Martino,L.: Object-Oriented Database Systems (Concept and Architectures), Addison-Wesley Publishing Company 1993, ISBN 0-201-62439-7, 264 stran.
3. Loomis,M.E.S.: Object Databases The Essential, Addison-Wesley Publishing Company 1995, ISBN – 201-56341-X, 230 stran.
4. Catell, R.,G.,G. (ed.): The Object Database Standard: ODMG-93 (Release 1.1), San Francisco, Morgan Kaufmann 1994
5. Hruška,T.: Objektově orientované databázové technologie, in: Sborník konference Tvorba software 95, Tanger Ostrava 1995, str. 113-124
6. Jacobson,I.: Object-Oriented Software Engineering - A Use case Driven Approach, Addison Wesley ACM Press 1992,.524 stran