Objektově orientované programování

Úvod do dědičnosti

2023/24

Osnova přednášky

- Co řeší dědičnost?
- Příklad.
- Dědičnost – základní princip.

Co řeší dědičnost?

Co se řeší?

- Znovu-použitelnost
  - Nechceme znovu opisovat (kopírovat) kód, který jsme už jednou napsali a odladili.
- Rozšiřitelnost
  - Chceme rozšířit (pozměnit) kód, který jsme už…

Role třídy?

- Znovu-použitelnost a rozšiřitelnost v kontextu použití tříd můžeme chápat jako:
  - Kombinování s jinými třídami, skládání
  - Rozšíření o nové chování
  - Pozměnění existujícího chování

Skládání x dědičnost

- Skládáním docílíme toho, že objekt jedné třídy je složen z objektů jiné třídy.
    - **Jedná se o vztah „MÁ“.**
- Dědičností docílíme toho, že nová třída je rozšířením nebo speciálním případem existující třídy (nebo více tříd).
    - **Jedná se o vztah „JE“.**

Příklad

```cpp
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;
    Client *partner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);
    Account(int n, Client *o, Client *p);
    Account(int n, Client *o, Client *p, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    Client *GetPartner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

Co je na třídě *Account* špatně?

- Účet s partnerem je rozšířením účtu bez partnera.
- Účet s partnerem **JE** účet
- Můžeme využít dědičnost.
- Jak?

Deklarace předka

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

Deklarace potomka

```
class PartnerAccount : public Account
{
private:
    Client *partner;

public:
    PartnerAccount(int n, Client *o, Client *p);
    PartnerAccount(int n, Client *o, Client *p, double ir);

    Client *GetPartner();
};
```

Implementace konstruktorů

```
Account::Account(int n, Client *o)
{
    this->number = n;
    this->owner = o;
    this->balance = 0;
    this->interestRate = 0;
}

Account::Account(int n, Client *o, double ir)
{
    this->number = n;
    this->owner = o;
    this->balance = 0;
    this->interestRate = ir;
}
```

Potomek *PartnerAccount* použije pro inicializaci členských položek konstruktor rodiče *Account*, kterému předá potřebné inicializační hodnoty.

```
PartnerAccount::PartnerAccount(int n, Client *o, Client *p) : Account(n, o)
{
    this->partner = p;
}

PartnerAccount::PartnerAccount(int n, Client *o, Client *p, double ir) : Account(n, o, ir)
{
    this->partner = p;
}
```

Použití (zastupitelnost)

```cpp
int main()
{
    Account *a;
    PartnerAccount *pa;
    pa = new PartnerAccount(0, new Client(0, "Smith"), new Client(1, "Jones"));
    a = pa;

    cout << a->GetOwner()->GetName() << endl;
    //cout << a->GetPartner()->GetName() << endl;

    cout << pa->GetPartner()->GetName();

    getchar();
    return 0;
}
```

```
Smith
Jones
```

Banka s účty dvou typů

```
class Bank
{
private:
    Client * * clients;
    int clientsCount;

    Account** accounts;
    int accountsCount;

public:
    Bank(int c, int a);
    ~Bank();

    Client* GetClient(int c);
    Account* GetAccount(int n);

    Client* CreateClient(int c, string n);
    Account* CreateAccount(int n, Client *o);
    Account* CreateAccount(int n, Client *o, double ir);
    PartnerAccount* CreateAccount(int n, Client *o, Client *p);
    PartnerAccount* CreateAccount(int n, Client *o, Client *p, double ir);

    void AddInterest();
};
```

Mělo by fungovat? A proč?

```
int main()
{
    Account *a;
    PartnerAccount *pa;

    Bank *b = new Bank(100, 1000);
    Client *o = b->CreateClient(0, "Smith");
    Client *p = b->CreateClient(1, "Jones");
    a = b->CreateAccount(0, o);
    pa = b->CreateAccount(1, o, p);

    cout << a->GetOwner()->GetName() << endl;
    cout << pa->GetPartner()->GetName() << endl;

    cout << b->GetClient(1)->GetName() << endl;
    //cout << b->GetClient(1)->GetPartner() << endl;

    getchar();
    return 0;
}
```

Dědičnost – základní princip

Terminologie

- Předek – potomek, přímý předek – potomek
- Rodič – dcera (syn)
- Nadřazená (super, bázová) třída – pod (sub) třída

Vztahy v dědičnosti

A

B

C

- A je bázová třída třídy B, A je rodič B, A je předek C
- B je bázová třída třídy C, třída B dědí z třídy A, B je rodič C
- C dědí z B a A, C je dceřiná třída třídy B, C je potomek A a přímý potomek B.

Příklady

- Vozidlo – kolo, motorka, osobní auto
- Osoba – uživatel, správce
- Kolekce – seznam, množina

Špatně?

- Auto – Škoda
  - Škoda je ZNAČKA *osobního auta*.
- Strom – smrk
  - Smrk je DRUH *jehličnatého stromu*.

Generalizace - specializace

- Nezaměňovat vztah *„je instancí“* a *„dědí z“*.
  - „je instancí“ je vztah mezi třídou a objektem
  - „dědí z“ je vztah mezi třídami.
- Vztah dědičnosti definuje vztah *OBECNÝ – SPECIÁLNÍ*
- Potomek by tedy měl reprezentovat speciální případ předka…
- … a předek by měl reprezentovat zobecnění svých potomků.

Jinak řečeno…

- Předek definuje společné chování všech svých potomků.
- Potomci mohou toto chování rozšířit či pozměnit.
- Potomci se nemohou tohoto chování zbavit.
- A tedy:
  - **Dědí se vše bez výjimky!!!**
  - **Dědí se i míra skrývaní informace…**

Vztah skládání a dědičnosti

- Skládání – „MÁ“ x dědičnost „JE“

- Nicméně:
  - Dědičnost můžeme chápat jako důsledek skládání.
  - Instance třídy potomka obsahuje vše, co má instance třídy předka.

Hierarchie

- Při použití dědičnosti vznikají hierarchie tříd.
- V našem případě pracujeme s jednoduchou dědičností.
  - Každý potomek má právě jednoho přímého předka.
  - Předek může mít více přímých potomků.
- V případě jednoduché dědičnosti je touto hierarchií strom.
- Nezaměňovat *hierarchii objektů* (kompozice) a *hierarchii tříd* (dědičnost).

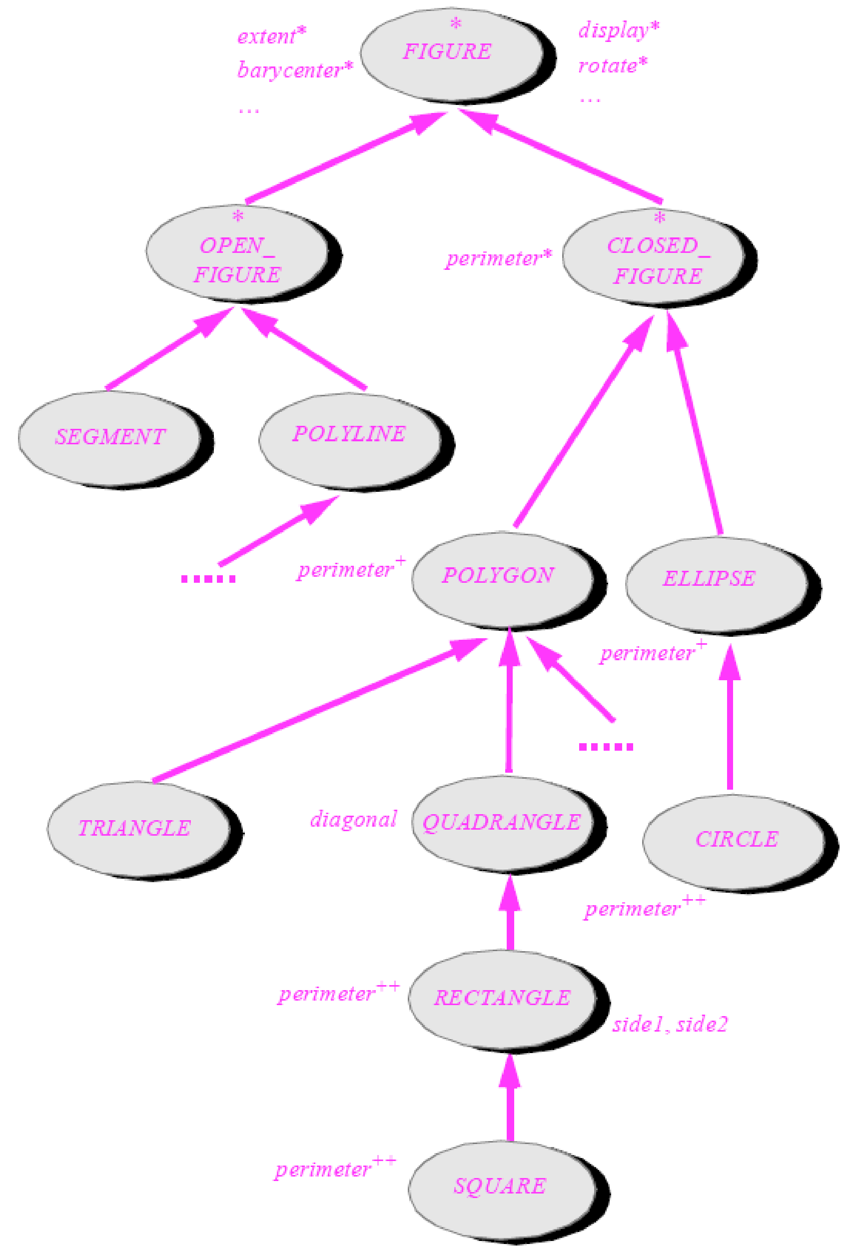

Figure type hierarchy

Liskové substituční princip

- Barbara Liskov 1987. *Data abstraction and hierarchy*.
- Bertrand Meyer. *Invarianty chování*.
- Potomek může vždy zastoupit předka…
  - … a to proto, že mají společné chování.
- Opačně to neplatí…

Vznik potomka

1. Volání konstruktoru objektu.
2. Volání konstruktoru přímého předka.
3. Vykonání konstruktoru přímého předka.
4. Vykonání konstruktoru objektu.

Úkoly na cvičení

- Implementuje příklad z přednášky a vytvořte banku s mnoha klienty a účty. Zaměřte se na pochopení principu zastupitelnosti a na to, jak fungují konstruktory v dědičnosti.

- Navrhněte a implementujte další jednoduché příklady dědičnosti s rozšířením společného stavu a chování, jako například *Auto, Osobní auto, Nákladní auto*.

Kontrolní otázky

- Které dva klíčové požadavky řešíme pomocí dědičnosti?
- Jaké návrhové požadavky máme na použití tříd (co s nimi můžeme dělat)?
- Jaký je rozdíl mezi dědičností a skládáním? Co mají společného?
- V jakých rolích vystupují třídy v dědičnosti? Použijte správnou terminologii.
- Vysvětlete v jakém obecném vztahu je třída, ze které se dědí, se třídou, která dědí.
- Co všechno se dědí, co ne a proč?
- Co rozumíme jednoduchou dědičností a jak s tím souvisí hierarchie tříd v dědičnosti?
- Co je Liskové substituční princip a jak se projevuje v dědičnosti?
- V jakém pořadí se volají a vykonávají konstruktory při použití dědičnosti?

Ke studiu

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [459-467]