Objektově orientované programování

Dědičnost – změna chování

2023/24

Osnova přednášky

- Rozšíření chování.
- Změna chování.
- Příklad.

Rozšíření chování

Když rozšiřujeme chování…

- Můžeme bezpečně použít to, co už máme.
- Nehrozí žádný problém s pochopením, jak se objekt chová.
- Objekt vystupuje sám za sebe…
- …nebo za některého ze svých předků.

Paradox specializace a rozšíření

- Vztah dědičnosti je vztahem *obecný - speciální*.
- Potomek je tedy *speciálním případem* předka.
- Paradoxní je, že při rozšíření dochází k tomu, že *potomek umí více, než kterýkoli jeho předek*.

…a tedy

- **Čím bohatší chování uvažujeme, tím méně tříd ho poskytuje.**

- V dědičné hierarchii je nejmenší společné chování definováno ve společném předkovi.

- Koncové třídy této hierarchie mají nejbohatší chování (každá trochu jiné).

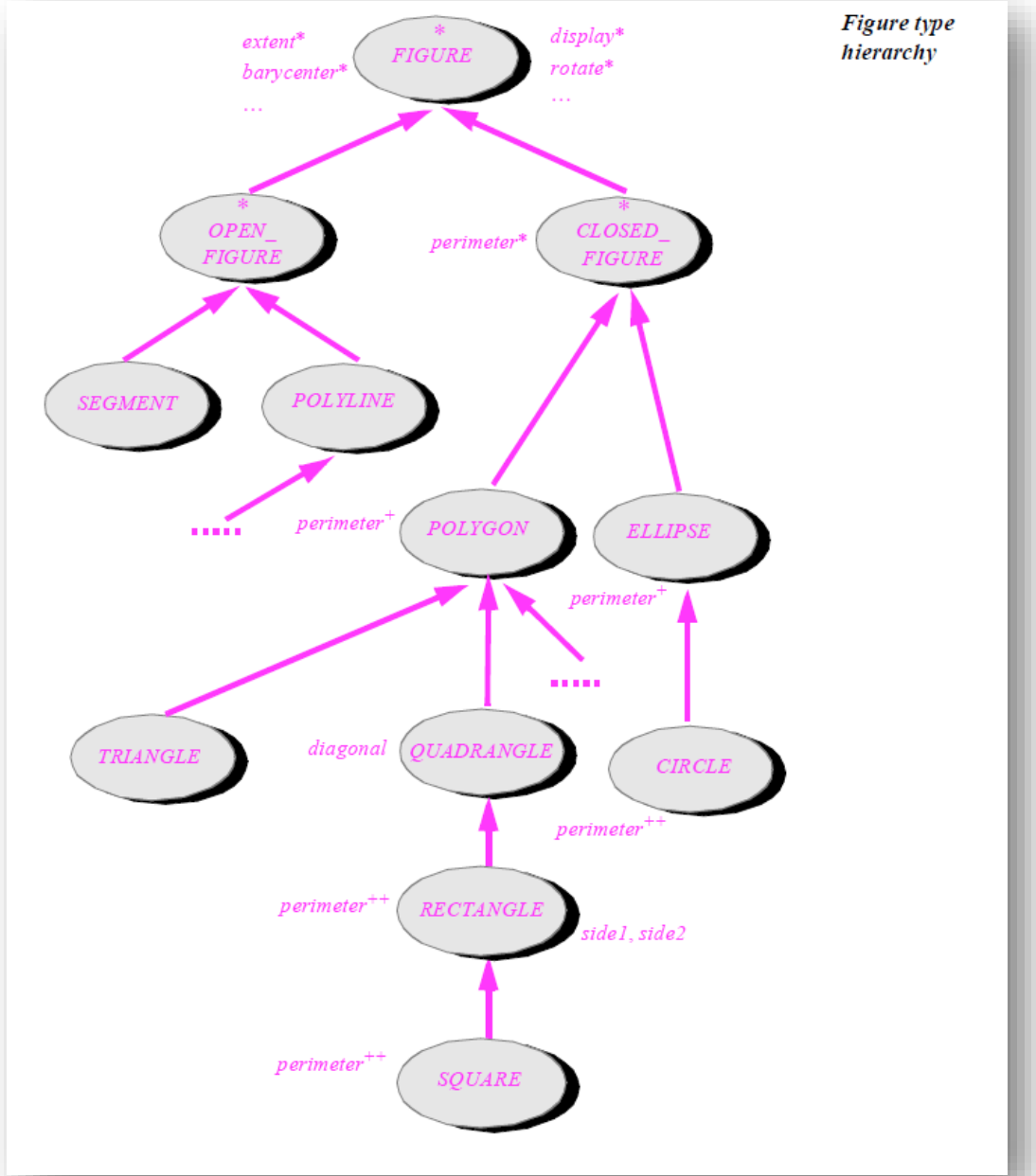


Figure type hierarchy
FIGURE
extent*
barycenter*
...
display*
rotate*
...
OPEN_ FIGURE
CLOSED_ FIGURE
perimeter*
SEGMENT
POLYLINE
perimeter+
POLYGON
ELLIPSE
perimeter+
TRIANGLE
diagonal
QUADRANGLE
CIRCLE
perimeter++
perimeter++
RECTANGLE
side1, side2
perimeter++
SQUARE

Špatný příklad

- Potřeba rozšíření sama o sobě není dostačující pro použití dědičnosti.
- Např. vztah bodu a kružnice, mohli bychom potřebovat rozšířit bod o práci s poloměrem (nové chování).
- Je to dostatečné, abychom se rozhodli použít dědičnost?

Ne!!!

- Není splněna podmínka **specializace** (kružnice není speciálním případem bodu).

Změna chování

Změna chování

- Pokud je chování deklarováno v předkovi, můžeme ho v potomkovi deklarovat znovu.
  - Existuje pak více metod stejného jména.
- Deklarované chování pak musíme v potomkovi implementovat (aby bylo proveditelné).
  - *Deklarované chování nemusí být implementované v předkovi.*

Přetížení jako rozšíření chování

Přetížení

- Přetížením rozumíme situaci, kdy daná metoda má stejné jméno, ale má:
  - jiné parametry,
  - jiné typy parametrů,
  - jiný typ návratové hodnoty.
- Přetížení nicméně **není změna chování**, a to i přesto, že metoda má stejné jméno.

Typy přetížení

- **Jméno metody zůstává stejné.**
- Jiný počet parametrů.
- Jiné datové typy parametrů.
- Jiná návratová hodnota (ne v *C++*).
- Lze kombinovat.

Překrytí

- Překrytím rozumíme situaci, kdy metoda potomka má stejnou deklaraci, jako metoda předka (stejnou signaturu).

- Potomek dědí i metodu předka. Má tedy dvě metody se stejnou deklarací.

Kdy použít překrytí?

- Typickým příkladem použití *přetížení* jsou konstruktory.
- Typickým příkladem použití *překrytí* je opravdová změna chování potomka.
- Příkladem může být metoda na výběr peněz z různých typů účtů v bance.

Příklad

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();
};
```

Překrytí

- Deklarujeme třídu *CreditAccount*.
- Překryjeme metodu *CanWithdraw*.
- Má stejnou signaturu, ale jinou definici.
- Důsledek: Ve třídě *CreditAccount* bude instanční metoda *CanWithdraw* dvakrát!!!

Deklarace potomka

```
class CreditAccount : public Account
{
private:
    double credit;

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

    bool CanWithdraw(double a);
};
```

Definice

```
bool Account::CanWithdraw(double a)
{
    return (this->balance >= a);
}
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

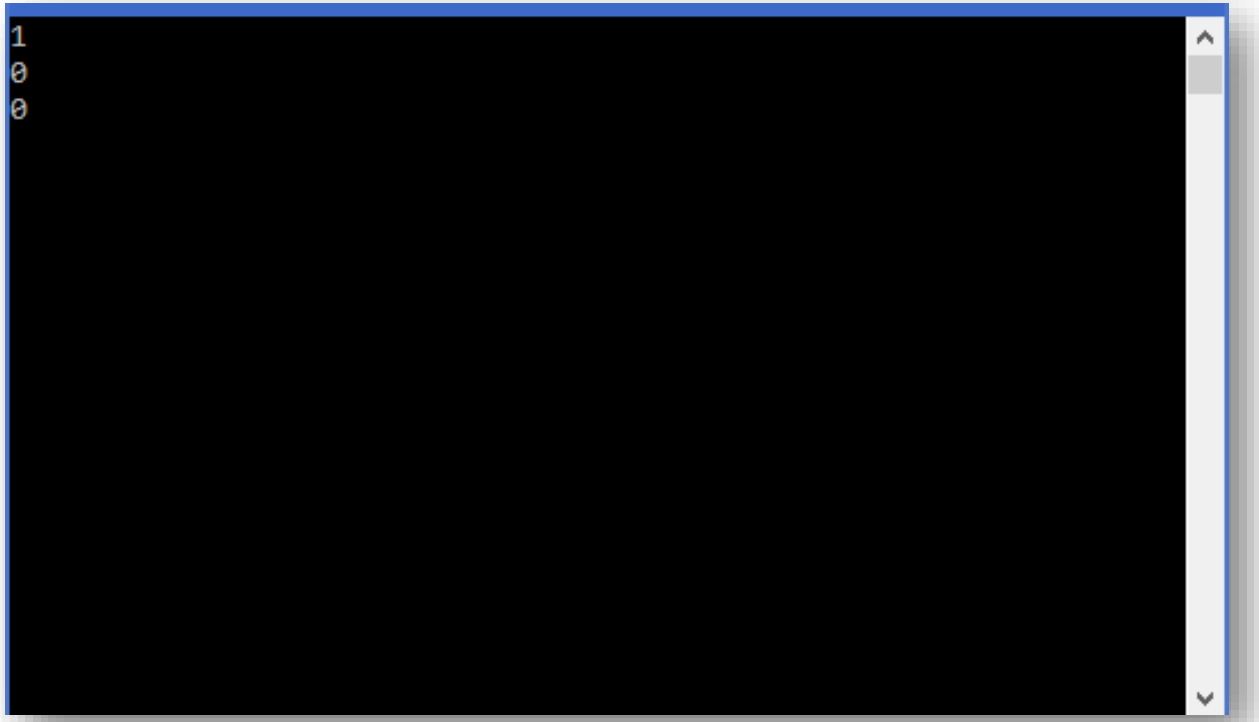
Definice (připomenutí)

```
bool Account::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

Použití

```
int main()
{
    Client *o = new Client(0, "Smith");

    CreditAccount *ca = new CreditAccount(1, o, 1000);
    cout << ca->CanWithdraw(1000) << endl;

    Account *a = ca;
    cout << a->CanWithdraw(1000) << endl;

    cout << ca->Withdraw(1000) << endl;

    a = nullptr;
    delete ca;

    getchar();
    return 0;
}
```

Výsledek?



Jsme hotovi?

- NE!!!
- Jak vybereme z účtu (pokud můžeme), když nemáme přístup k členské proměnné *balance*?
- Jaké máme možnosti?

Vlastní metoda?

```
class CreditAccount : public Account
{
private:
    double credit;

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

    bool CanWithdraw(double a);
    bool Withdraw(double a);
};
```

Máme problém…

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

Jaké tedy máme možnosti?

- *Public* přístup k datové položce?
- Porušení zapouzdření?
- Nebo jinak?

Nový předek

```
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();
};
```

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

    Client *owner;

protected:
    double balance;

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();
};
```

Funguje, jenže…

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

- Máme stejný kód dvakrát.
- Porušujeme zapouzdření.
- Při zastoupení předka potomkem se použijí různé metody

Výsledek

Porušení zapouzdření

- Při změně chování může vzniknout potřeba pracovat se soukromou částí předka.
- Jedná se samozřejmě o porušení zapouzdření a toho si musíme být vědomi…
- …nicméně každé rozumné pravidlo má nějaké výjimky.

Dá se zavolat metoda předka?

- Je to stejné jako volání statické metody ☹
- Z potomka voláme originální metodu.
- *Account::CanWithdraw(a);*

Úkoly na cvičení

- Implementujte příklady z přednášky. Zaměřte se na překrytí, vyzkoušejte použití „protected“.

- Navrhněte a implementuje jednoduchou dědičnou hierarchii geometrických objektů, které budou mít společné metody „Obsah“ a „Obvod“. Využijte překrytí a rozeberte chování při využití substitučního principu.

Kontrolní otázky

- Co rozumíme paradoxem specializace a rozšíření?
- Uveďte správné a špatné příklady vztahu "generalizace-specializace".
- Co rozumíme v dědičnosti změnou chování?
- Co rozumíme přetížením? Jedná se o rozšíření nebo změnu chování?
- Uveďte různé typy přetížení.
- Co rozumíme překrytím? Jedná se o rozšíření nebo změnu chování?
- Jaký princip porušujeme, použijeme-li „protected“ a proč?
- Jaký problém přináší potřeba změny chování v dědičnosti?
- Popište, jak se prakticky projevuje různá míra přístupu k položkám třídy.
- Jak se použití „protected“ projeví ve vztahu předka a potomka?

Ke studiu

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