

PB161 Programování v jazyce C++

Přednáška 11

Návrhové principy a vzory
Sémantika objektů v C++ a jinde

Nikola Beneš

30. listopadu 2015

Zápočtový příklad na cvičení

- tento týden (30. 11. – 4. 12.) nanečisto
- poslední týden (14. – 18. 12.) naostro
- náhradní termín začátkem ledna
 - bude upřesněno podle počtu lidí

Zvané přednášky

- příští týden (7. 12.) – Jiří Weiser: C++11 podrobněji
- poslední týden (14. 12.) – Juraj Michálek & Martin Halfar (YSoft): refaktORIZACE, návrhové antivzory

Návrhové principy a vzory

Jak správně navrhnout hierarchii v OOP?

- co umístit do jedné třídy?
- jaký vztah mezi třídami?

Jak se pozná špatný návrh?

- malé změny vyžadují velké úpravy
- změny způsobí nečekané problémy
- velká provázanost (s konkrétní aplikací, s jiným kódem)

Základní návrhové principy pro OOP

- **S**RP – *The Single Responsibility Principle*
- **O**CP – *The Open/Closed Principle*
- **L**SP – *The Liskov Substitution Principle*
- **I**SP – *The Interface Segregation Principle*
- **D**IP – *The Dependency Inversion Principle*

- [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))
(odkazy v části *References*)

The Single Responsibility Principle

- „Třída by měla mít jediný důvod ke změně.“
- obecnější princip: každý se stará o jednu věc

Důsledky porušení

- změna jedné z funkcionalit vyžaduje rekompilaci
- úprava kódu poruší více funkcionalit

Příklad porušení SRP

- třída `Rectangle` zároveň vykresluje čtverec a počítá jeho obsah
- někteří uživatelé potřebují obě tyto funkcionality, někteří jen jednu

[ukázka]

- <http://www.objectmentor.com/resources/articles/srp.pdf>

Lepší řešení

- rozdělit funkcionalitu `Rectangle` do dvou tříd
- o počítání obsahu se postará `GeometricRectangle`

The Open/Closed Principle

- „Třídy by mělo být možné rozšiřovat, ale bez jejich modifikace.“
- nejen třídy (moduly, jiné entity)
- *open for extension*: možnost přidávat chování
- *closed for modification*: možnost použití jiným kódem
- dosahuje se použitím dědičnosti a abstrakce

Související

- nepoužívat globální proměnné
- soukromé atributy objektů
- používat `dynamic_cast` opatrně

Důsledek porušení: kaskáda změn v kódu

The Liskov Substitution Principle

- (už jsme viděli)
- „Potomek může zastoupit předka.“
- funkce očekávající objekt typu předka musí fungovat s objekty typu potomka, aniž by o tom věděly
- úzce souvisí s OCP

Důsledek porušení: neočekávané chování (špatné předpoklady)

Příklad porušení LSP

- mějme třídu `Rectangle` a třídu `Square`, která z ní dědí
- obě mají metody `setWidth` a `setHeight` (a odpovídající `get`)

```
void f(Rectangle & r) {  
    r.setWidth(5);  
    r.setHeight(6);  
    assert(r.getWidth() * r.getHeight() == 30);  
}
```

- kde je problém?

Příklad porušení LSP

- mějme třídu `Rectangle` a třídu `Square`, která z ní dědí
- obě mají metody `setWidth` a `setHeight` (a odpovídající `get`)

```
void f(Rectangle & r) {  
    r.setWidth(5);  
    r.setHeight(6);  
    assert(r.getWidth() * r.getHeight() == 30);  
}
```

- kde je problém?
- čtverec je pravoúhelník, ale objekt typu `Square` není objektem typu `Rectangle` (mají jiné *chování*)

The Interface Segregation Principle

- „Vytvářejte specifická rozhraní pro specifické klienty.“
- více malých rozhraní je lepší než jedno obrovské
- klienti by neměli být nuceni záviset na rozhraních, které nepoužívají
- souvisí s SRP

Jak dosáhnout?

- rozhraní s mnoha metodami: kdo je používá?
- opravdu používají všichni všechny metody?
- rozdělit metody do samostatných rozhraní
- použití vícenásobné dědičnosti

The Dependency Inversion Principle

- obecné třídy by neměly záviset na specializovaných třídách; všichni by měli záviset na abstrakcích
- abstrakce by neměly záviset na detailech, ale naopak
- souvisí s OCP a LSP
- použití rozhraní (čistě abstraktních tříd)

Důsledky porušení:

- přidání podpory nového typu vede ke změně v obecné třídě
- obecná třída použitelná jen pro co byla implementována

Příklad porušení:

- třída `Worker` s metodou `work()` a třída `Manager`, která ji používá
- závislost `Manager` → `Worker`
- co když budeme chtít přidat novou třídu `SuperWorker`?

Řešení:

- vytvoření abstrakce (rozhraní) `IWorker`
- závislosti `Manager` → `IWorker`, `Worker` → `IWorker`, ...

Keep It Simple, Stupid!

Princip KISS

- jednoduché, postačující řešení může být lepší než komplikované a rafinované
 - snazší pochopení (dalšími vývojáři, námi samotnými)
 - menší riziko chyby
- kompromis mezi aktuálními a budoucími požadavky
 - příliš mnoho budoucích požadavků návrh komplikuje
 - omezení na aktuální požadavky vadí budoucí rozšiřitelnosti
- průběžný vývoj, refaktORIZACE (nebát se!)

Motivace

- opakující se programátorské problémy
- opakující se způsoby řešení
- vhodné konstrukce pro řešení: návrhové vzory
- kniha *Design Patterns: Elements of Reusable Object-Oriented Software* (23 vzorů)
- autoři *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* (tzv. Gang of Four, GoF)
- existují i jiné návrhové vzory
- kde začít?
https://en.wikipedia.org/wiki/Software_design_pattern

Co je návrhový vzor?

- (jednoduchý) způsob, jak řešit skupiny podobných problémů
- opakovatelný, (víceméně) nezávislý na konkrétní aplikaci či jazyce

Pozitiva

- řeší běžné problémy
- mohou zlepšovat kód a jeho udržitelnost

Kritika

- zbytečně složité na jednoduché problémy
- nadužívání může vést ke komplikovanějšímu kódu
- často řeší jen nedokonalost používaného jazyka

Sémantika objektů v C++ a v jiných jazycích

Sémantika objektů v C++ a v jiných jazycích

aneb Jak se předávají objekty funkcím?

Sémantika objektů v C++ a v jiných jazycích

aneb Jak se předávají objekty funkcím?

aneb Co znamená =?

Sémantika objektů v C++ a v jiných jazycích

aneb Jak se předávají objekty funkcím?

aneb Co znamená =?

aneb Proč není `const` jako `final`?

Sémantika objektů v C++ a v jiných jazycích

aneb Jak se předávají objekty funkcím?

aneb Co znamená =?

aneb Proč není `const` jako `final`?

„Kolik jazyků umíš, tolikrát jsi programátorem.“

Předávání parametrů funkcím (volání)

Volání hodnotou (call by value)

- funkce dostane kopii parametru
- změny parametru ve funkci se navenek neprojeví
- podporuje velká řada jazyků, minimálně pro primitivní typy
- ve funkcionálním programování: striktní vyhodnocování

Předávání parametrů funkcím (volání)

Volání hodnotou (call by value)

- funkce dostane kopii parametru
- změny parametru ve funkci se navenek neprojeví
- podporuje velká řada jazyků, minimálně pro primitivní typy
- ve funkcionálním programování: striktní vyhodnocování

Volání odkazem (call by reference)

- parametr ve funkci je alias předaného parametru
- jakékoli změna parametru ve funkci se projeví navenek

Předávání parametrů funkcím (volání)

Volání hodnotou (call by value)

- funkce dostane kopii parametru
- změny parametru ve funkci se navenek neprojeví
- podporuje velká řada jazyků, minimálně pro primitivní typy
- ve funkcionálním programování: striktní vyhodnocování

Volání odkazem (call by reference)

- parametr ve funkci je alias předaného parametru
- jakékoli změna parametru ve funkci se projeví navenek

Volání sdílením (call by sharing)

- něco mezi voláním hodnotou a odkazem
- nazýváno mnohdy různě (zdroj hádek)
- Java (pro objekty), Python, Ruby, ...

Předávání parametrů funkcím (volání)

Volání hodnotou (call by value)

- funkce dostane kopii parametru
- změny parametru ve funkci se navenek neprojeví
- podporuje velká řada jazyků, minimálně pro primitivní typy
- ve funkcionálním programování: striktní vyhodnocování

Volání odkazem (call by reference)

- parametr ve funkci je alias předaného parametru
- jakékoli změna parametru ve funkci se projeví navenek

Volání sdílením (call by sharing)

- něco mezi voláním hodnotou a odkazem
- nazýváno mnohdy různě (zdroj hádek)
- Java (pro objekty), Python, Ruby, ...

A jiné (volání jménem, vstupem/výstupem, ...)

Hádka o předávání parametrů

Volá Python hodnotou nebo odkazem?

```
def f(s):  
    s.append(3)  
  
t = [1, 2]  
f(t)  
print t # [1, 2, 3]
```

AHA! Python volá odkazem!

Hádka o předávání parametrů

Volá Python hodnotou nebo odkazem?

```
def f(s):  
    s.append(3)
```

```
t = [1, 2]  
f(t)  
print t # [1, 2, 3]
```

AHA! Python volá odkazem!

```
def f(s):  
    s = [3]
```

```
t = [1, 2]  
f(t)  
print t # [1, 2]
```

AHA! Python volá hodnotou!

Hádka o předávání parametrů

Volá Python hodnotou nebo odkazem?

```
def f(s):  
    s.append(3)
```

```
t = [1, 2]  
f(t)  
print t # [1, 2, 3]
```

AHA! Python volá odkazem!

```
def f(s):  
    s = [3]
```

```
t = [1, 2]  
f(t)  
print t # [1, 2]
```

AHA! Python volá hodnotou!

Skutečnost? Python volá sdílením (call by sharing).

- na nejnižší úrovni (assembler) je každé volání hodnotou

Hodnotová sémantika v jazyce C

```
void f(int x) {  
    x += 7;  
}
```

```
int main() {  
    int a = 3;  
    f(a);  
    return a; // 3  
}
```

Jazyk C a struktury

```
typedef struct { int x, y; } MyObject;
```

```
int f(MyObject a) {  
    MyObject b = { 7, 10 };  
    a = b;  
    a.x += 11;  
    return a.x + b.x;  
}
```

```
int main() {  
    MyObject o = { 0, 0 };  
    int z = f(o);  
    return z + o.x;  
}
```

Hodnotová sémantika

- C nemá jinou sémantiku, ale může ji emulovat

Ukazatele v C

- jejich *hodnota* je adresa v paměti
- emulace volání odkazem:

```
void f(int *x) {  
    *x += 7;  
}  
  
int main() {  
    int a = 3;  
    f(&a);  
    return a; // 10;
```

- pozor na rozdíl mezi $*x = a$ a $x =$

Ukazatele v C a const

- dvě možnosti:
 - ukazatel
 - to, na co se ukazuje

```
int main() {  
    int x, y, z;  
    int *px = &x;  
    const int *py = &y;  
    int *const pz = &z;  
    *px = 0;  
    px = &y;  
    *py = 0; // CHYBA!  
    py = &x;  
    *pz = 0;  
    pz = &x; // CHYBA!  
}
```

Odkazová (referenční) sémantika

C++ a reference (připomenutí)

- reference nemůžou být `nullptr`
- reference nemůžou být přesměrovány
- `int & rx` je *něco jako* `int * const px`
- (s tím rozdílem, že místo `rx` si dosadíme `*px`)

```
// C++: reference
void f( int &x ) {
    x += 7;
}
```

```
// C (nebo C++): ukazatel
void f( int *const px ) {
    *px += 7;
}
```

Odkazová (referenční) sémantika

C++ a reference (připomenutí)

- reference nemůžou být `nullptr`
- reference nemůžou být přesměrovány
- `int & rx` je *něco jako* `int * const px`
- (s tím rozdílem, že místo `rx` si dosadíme `*px`)

```
// C++: reference  
void f( int &x ) {  
    x += 7;  
}
```

```
// C (nebo C++): ukazatel  
void f( int *const px ) {  
    *px += 7;  
}
```

- `const int & rx` je *něco jako* `const int * const px`

Odkazová (referenční) sémantika

Jazyk C++ a objekty (volání hodnotou)

```
class MyObject {  
    int x, y;  
public:  
    MyObject(int, int);  
    int getX() const;  
    int getY() const;  
    void setX(int);  
    void setY(int);  
};  
  
int f(MyObject a) {  
    MyObject b( 7, 10 );  
    a = b;  
    a.setX( a.getX() + 11 );  
    return a.getX() + b.getX();  
}  
  
int main() {  
    MyObject o( 0, 0 );  
    int z = f(o);  
    return z + o.getX();  
}
```

- přiřazení je kopírování

Odkazová (referenční) sémantika

Jazyk C++ a objekty (volání odkazem)

```
class MyObject {  
    int x, y;  
public:  
    MyObject(int, int);  
    int getX() const;  
    int getY() const;  
    void setX(int);  
    void setY(int);  
};  
  
int f(MyObject & a) {  
    MyObject b( 7, 10 );  
    a = b;  
    a.setX( a.getX() + 11 );  
    return a.getX() + b.getX();  
}  
  
int main() {  
    MyObject o( 0, 0 );  
    int z = f(o);  
    return z + o.getX();  
}
```

- přiřazení je stále kopírování (reference se nedají přesměrovat)

Sémantika volání sdílením (call by sharing)

Volání sdílením

- Java, Python, Ruby, Scheme, OCaml ...
 - v Javě nazýváno volání hodnotou reference
 - v Ruby nazýváno volání referencí
- poprvé popsáno 1974 Barbarou Liskov
- objekty jsou reference (odkazy)
- reference se mohou přesměrovat
 - přiřazení je přesměrování, ne kopírování

Podobnost s voláním odkazem

- objekty předané funkcím mohou funkce modifikovat (pomocí metod, příp. přetížených operátorů – např. Python)
- ale přiřazení (přesměrování) objekt nemění

Dá se emulovat/implementovat pomocí ukazatelů

Porovnání sémantik

// C++ hodnotou

```
void f(MyObject a) {  
    MyObject b( 10, 12 );  
    a.setX( 100 );  
    a = b;  
    a.setY( 50 );  
    b.setX( 20 );  
}
```

// C++ odkazem

```
void f(MyObject &a) {  
    MyObject b( 10, 12 );  
    a.setX( 100 );  
    a = b;  
    a.setY( 50 );  
    b.setX( 20 );  
}
```

// Java: sdílením

```
void f(MyObject a) {  
    MyObject b = new MyObject( 10, 12 );  
    a.setX( 100 );  
    a = b;  
    a.setY( 50 );  
    b.setX( 20 );  
}
```

// C++: emulace sdílení ukazatelem

```
void f(MyObject *a) {  
    MyObject *b = new MyObject( 10, 12 );  
    a->setX( 100 );  
    a = b;  
    a->setY( 50 );  
    b->setX( 20 );  
}
```

Konstantní hodnoty (primitivní typy)

- Javovské `final int x` je v C++ `const int x`

Konstantní reference

- Javovské `final MyObject o` je jakoby `MyObject * const o` v C++
- **není** to `const MyObject * o`
- `final` zakazuje měnit (přesměrovat) referenci, nezakazuje měnit odkazovaný objekt

Výhody a nevýhody hodnotové sémantiky objektů

Výhody

- rychlost
- objekty se mohou alokovat na zásobníku
- objekty mohou obsahovat jiné objekty (ne jen odkazy na ně)

Nevýhody

- velikost objektu musí být známa při kompilaci kódu, který jej používá
- důsledek: nutnost překompilovat vše i při změně soukromých částí
- obtížnější pro pochopení, kopírování je zdroj chyb

Ještě jednou zpět k Pythonu

- Python umí přetěžovat operátory
- + v Pythonu řetězí seznamy

```
def f(s):  
    s += [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

```
def f(s):  
    s = s + [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

Ještě jednou zpět k Pythonu

- Python umí přetěžovat operátory
- + v Pythonu řetězí seznamy

```
def f(s):  
    s += [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

```
def f(s):  
    s = s + [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

- výsledky nebudou stejné

Ještě jednou zpět k Pythonu

- Python umí přetěžovat operátory
- + v Pythonu řetězí seznamy

```
def f(s):  
    s += [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

```
def f(s):  
    s = s + [7, 9]
```

```
t = [1, 2]  
f(t)  
print t
```

- výsledky nebudou stejné
- kombinace volání sdílením a přetěžování operátorů může být neintuitivní

Návrhové principy

- je dobré o nich vědět a snažit se je dodržovat

Návrhové vzory

- užitečná řešení častých problémů
- dobrý nástroj, ale nepřehánějte to s nimi

Sémantiky objektů v různých jazycích

- existují různé sémantiky
 - mají své výhody a nevýhody
- souvisí se způsobem volání funkcí
- syntakticky *podobné* jazyky mohou mít velmi *různou* sémantiku
- zdroj častých nedorozumění a překvapení

Není žádný jeden dokonalý programovací jazyk.