PB161 – Programování v jazyce C++
Objektově Orientované Programování

Šablony, Návrhové principy a
(anti-)vzory

Organizační

- Zápočtový příklad nanečisto
  - na vašem cvičení, nácvik

Organizační - zvané přednášky

- 26.11. Juraj Michálek (SinusGear)
  - There is much more to C(++)
- 3.12. Ondřej Krajíček (YSoft)
  - Software Developer and his role in the process of Software Quality
- 10.12. Šimon Tóth (Cesnet)
  - C++11
  - + diskuze o předmětu

Šablony

Šablony funkcí - motivace

- Máme kód, který chceme použít s hodnotami různých typů
  - např. prohoď hodnoty dvou argumentů A a B
- Kód (tělo funkce) je pro různé typy hodnot shodný
  - `temp = A; A = B; B = temp;`
  - liší se právě jen typem používaných hodnot
- Musíme vytvářet pro každý typ samostatnou funkci?
- Nemusíme, známe přece makra!

Přehazování prvků – využití makra

```
#include <iostream>
using std::cout;
using std::endl;
void myswap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
void myswap(float& a, float& b) {
    float temp = a;
    a = b;
    b = temp;
}
int main() {
    // Swap two integers
    int a = 5;
    int b = 7;
    cout << a << " " << b << endl;
    myswap(a, b);
    cout << a << " " << b << endl;

    return 0;
}
```

```
#include <iostream>
using std::cout;
using std::endl;

#define MYSWAP(A,B) {\
    A = A + B;\
    B = A - B;\
    A = A - B;}

int main() {
    float af = 5;
    float bf = 7;
     // Swap two by macro...
    cout << af << " " << bf << endl;
    MYSWAP(af, bf);
    cout << af << " " << bf << endl;


    return 0;
}
```

Přehazování prvků – problémy s makrem

- Nejasné chybové hlášky při syntaktické chybě
  - nevidíme přesně řádek chyby
- Nelze krokovat debuggerem
- Často obtížně odhalitelná logická chyba
  - nevidíme kód po rozvinutí makra
- Často spoléháme na existenci některých operátorů
  - např. + a – u SWAP
  - nemůžeme využít dodatečnou proměnnou `temp =`
- Šablony funkcí výše uvedené problémy řeší

Šablony funkcí - ukázka

```
void myswap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
void myswap(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```

Klíčovým slovem template zavedeme šablonu

Klíčovým slovem typename nebo class označíme parametrizovatelný typ

```
void myswap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Zatím nelze zkompilovat, typ T není známý

```
template <typename T>
void myswap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Do <> uvádíme datový typ nahrazující T při potřebě využití v kódu

```
std::string as = "Hello";
std::string bs = "World";
myswap<std::string>(as, bs);
myswap<int>(ai, bi);
```

Šablony funkcí – základní vlastnosti

- Jsou to vlastně „vylepšená“ makra
- Rozvíjí se do kódu stejně jako makra
  - mluvíme o tzv. instanci šablon
  - `myswap<int>(ai, bi);`
  - instanci šablon už znáte z STL kontejnerů
  - `vector<int> myVect;`
- Instance je vytvářena během překladu
  - výsledek je rychlý
  - samotná šablona v přeloženém kódu není
  - nepoužité šablony (== bez instance) v přeloženém kódu vůbec nejsou

Šablony funkcí – více typů parametrů

- Šablona může mít víc parametrů pro typ

```
template <typename T, typename U>
void myswap(T& a, T& b, U& c, U& d) {
    T temp1 = a;
    a = b;
    b = temp1;
    U temp2 = c;
    c = d;
    d = temp2;
}
```

```
// Swap two integers and two strings by template function
int main() {
  int ai = 5; int bi = 7;
  std::string as = "Hello";
  std::string bs = "Happy";
  myswap<int, std::string>(ai, bi, as, bs);
  return 0;
}
```

- Můžeme využít i pro funkci bez parametrů nebo parametrizovat návratovou hodnotu
  - `T foo() { … }`

Šablony funkcí – přetěžování

- Můžeme přetěžovat stejně jako běžné funkce

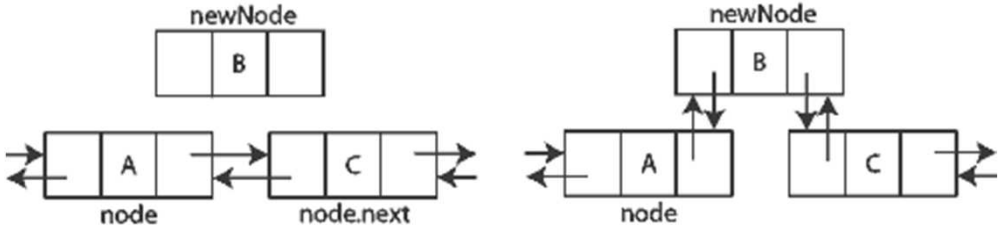
```
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
void myswap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

template <typename T>
void myswap(T& a, T& b, T& c) {
    T temp = a;
    a = b;
    b = c;
    c = temp;
}
```

```
int main() {
    std::string as = "Hello";
    std::string bs = "Happy";
    std::string cs = "World";

    // Swap two strings by template function
    cout << as << " " << bs << endl;
    myswap<std::string>(as, bs);
    cout << as << " " << bs << endl;


    // Swap three strings by template function
    cout << as << " " << bs << " " << cs << endl;
    myswap<std::string>(as, bs, cs);
    cout << as << " " << bs << " " << cs << endl;


    return 0;
}
```

Šablony funkcí – priority volání

- U instance šablon nemusíme uvádět typ, pokud bude jednoznačně určitelná cílová šablona
  - `myswap (as, bs);` místo `myswap<std::string>(as, bs);`
- Co v případě, že existuje i běžná funkce pro tento typ?
  - tj. funkce `void myswap(string&,string&) {}`
- Priority volání
  1. Existuje-li pro daný typ parametrů funkce, použije se
  2. Neexistuje-li, hledá se šablona pro daný typ
  3. Neexistuje-li, chyba při překladu

Šablony tříd



- Šablona pro vytváření tříd parametrizovaných konkrétním datovým typem
- Např. položka dynamického seznamu nese hodnotu konkrétního typu

```
class CLinkedItem {
    CLinkedItem* m_pNextItem;
    CLinkedItem* m_pPrevItem;
    int           m_value;
  …
};
```

```
template <typename T>
class CLinkedItem {
    CLinkedItem* m_pNextItem;
    CLinkedItem* m_pPrevItem;
    T            m_value;
public:
    CLinkedItem(T value) :
      m_pNextItem(0), m_pPrevItem(0), m_value(value) {}
    void setValue(T value) { m_value = value;}
    T getValue() { return m_value;}
};
```

Šablony tříd - ukázka

```
template <typename T>
class CLinkedItem {
    CLinkedItem* m_pNextItem;
    CLinkedItem* m_pPrevItem;
    T            m_value;
public:
    CLinkedItem(T value) :
      m_pNextItem(0), m_pPrevItem(0), m_value(value) {}
    void setValue(T value) { m_value = value;}
    T getValue() { return m_value;}
};
```

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
    CLinkedItem<int>  linkInt(10);
    CLinkedItem<std::string>  linkString("Hello world");

    cout << linkInt.getValue() << endl;
    cout << linkString.getValue() << endl;

    return 0;
}
```

Šablony tříd - implementace metody třídy

- Bez problémů, pokud implementujeme metodu zároveň s deklarací
  - viz. předchozí příklad
- Pokud implementujeme mimo deklaraci třídy
  - implementace metody u šablony třídy je také šablona
  - použijeme syntaxi známou ze šablony funkce
  - v případě implementace v rámci stejného *.h souboru jako deklarace šablony třídy bez problémů
- Umístění implementace metody třídy do separátního souboru (*.cpp)
  - (běžný způsob pro nešablonové třídy a metody)
  - u šablonových metod typicky problematické
  - překladač „nevidí“ zároveň šablonu třídy a metody
  - preferujte umístění implementace šablonových funkcí/tříd do *.h
  - http://www.parashift.com/c++-faq-lite/templates.html#faq-35.12

Šablona metod třídy - ukázka

```
class CLinkedItem {
public:
    // …
   void setValue(int value);
};
void CLinkedItem::setValue(int value) {
     m_value = value;
}
```

Nyní máme šablonu s typem T

```
template <typename T>
class CLinkedItem {
public:
    // …
   void setValue(T value);
};

template <typename T> void CLinkedItem<T>::setValue(T value) {
     m_value = value;
}
```

I metoda třídy je šablona s typem T

A jmenný prostor CLinkedItem je šablona s typem T

Šablony tříd – částečná specializace

- Šablonou vytváříme obecnou implementaci pro všechny vhodné datové typy
  - např. CLinkedItem očekává od typu pouze operátor =
- Specializace šablony nám umožňuje tuto obecnou implementaci pro konkrétní typ změnit
  - např. CLinkedItem se strcpy namísto operátoru = pro typ char*
- Částečná specializace šablon
  - upřesníme podmnožinu typů
  - typické použití pro ukazatele
  - `template <typename T> class CLinkedItem<T*> …`
- Pokud vytváříme instanci s ukazatelovým typem, bude mít šablona `CLinkedItem<T*>` přednost před `CLinkedItem<T>`

Šablony tříd – úplná specializace

- Úplná specializace šablon zafixuje konkrétní typ
  - např. char*
  - `template <> class CLinkedItem<char*> {`
- Při překladu se použije nejspecializovanější šablona
  - stejně jako pro šablony funkcí
  - char* vs. T* vs. T (sestupná priorita)
- Specializace šablon tříd nesouvisí s dědičností
  - jde o samostatnou třídu bez dědického vztahu k obecnější

Úplná specializace - ukázka

```
// Full specialization for char* type
template <> class CLinkedItem<char*> {
    CLinkedItem* m_pNextItem;
    CLinkedItem* m_pPrevItem;
    char*        m_value;
public:
    CLinkedItem(char* value) : m_pNextItem(0), m_pPrevItem(0) {
        m_value = 0;
        setValue(value);
    }
    void setValue(char* value);
    const char* getValue() const { return m_value;}
    ~CLinkedItem() { if (m_value) delete[] m_value; }
};
void CLinkedItem<char*>::setValue(char* value) {
    if (m_value) delete[] m_value;
    m_value = new char[strlen(value) + 1];
    strncpy(m_value, value, strlen(value)+1);
}
```

Šablona není parametrizovaná žádným volitelným datovým typem

Typ šablony je zafixován na char*

Jmenný prostor CLinkedItem použijeme ve specializaci s char*

Výhody šablon

- Přehledný kód s typovou kontrolou
  - stejně jako běžná funkce/třída
  - pouze doplníme parametrizaci datovým typem
- Je rozvinuto přímo do kódu => rychlost
- Nepoužité šablony nejsou zahrnuty do kódu
- Typicky můžeme krokovat debuggerem
  - můžeme hledat snáze logické chyby
- Plnohodnotná náhrada maker s výhodami

Návrhové principy

Jak navrhnout správně OO hierarchii?

- Co umístit do jedné třídy?
- Jaké třídy vytvořit a jaký mají mít mezi sebou vztah?
- Zapouzdření, dědičnost… jsou jen nástroje
  - dobrý objektový návrh je schopnost je dobře použít

Jak se pozná špatný návrh?

1. I malá změna vyžaduje velké množství úprav v existujícím kódu
2. Změna způsobuje nečekané problémy v jiných částech kódu
3. Je příliš svázán s konkrétní aplikací
   - je obtížné podčást kódu použít jinde

- Společným prvkem je velká provázanost kódu
  - OOP se snaží kód rozdělit do samostatných částí

Principy pro návrh OO architektury

- Pro OOP existuje 5 základních principů (a řada dalších)
  - **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
  - (jednotlivé principy se někdy částečně překrývají)
  - *http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)*
  - *http://www.objectmentor.com/resources/publishedArticles.html*
- Postup
  1. Porozumějte principům a s jejich vědomím navrhněte hierarchii
  2. Zkontrolujte, zda principy nejsou porušeny a případně opravte

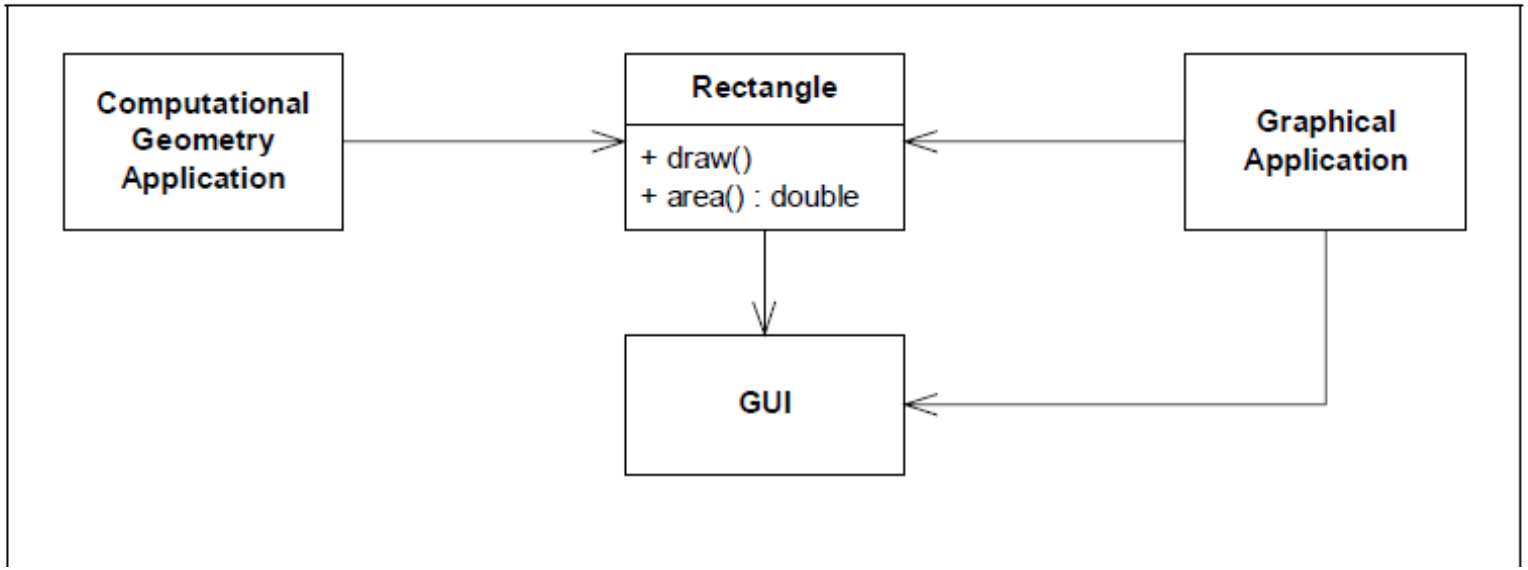
SRP - The Single Responsibility Principle

- Cílem je podpořit robustní rozšiřovatelnost aplikace
- Třída by měla být zodpovědná za plnění jediného funkčního požadavku
  - při změně požadavků na aplikaci by důvod pro změnu konkrétní třídy měl být právě jeden
  - při změně požadavků není zasažena nesouvisející funkčnost třídy (protože žádná další není ☺)
- http://www.objectmentor.com/resources/articles/srp.pdf

SRP – problémy při porušení

- Porušení principu může vést k
  - při použití jen jedné plněné funkčnosti se musí zbytečně kompilovat ostatní kód
  - při úpravě kódu přestane fungovat jiná plněná funkčnost třídy

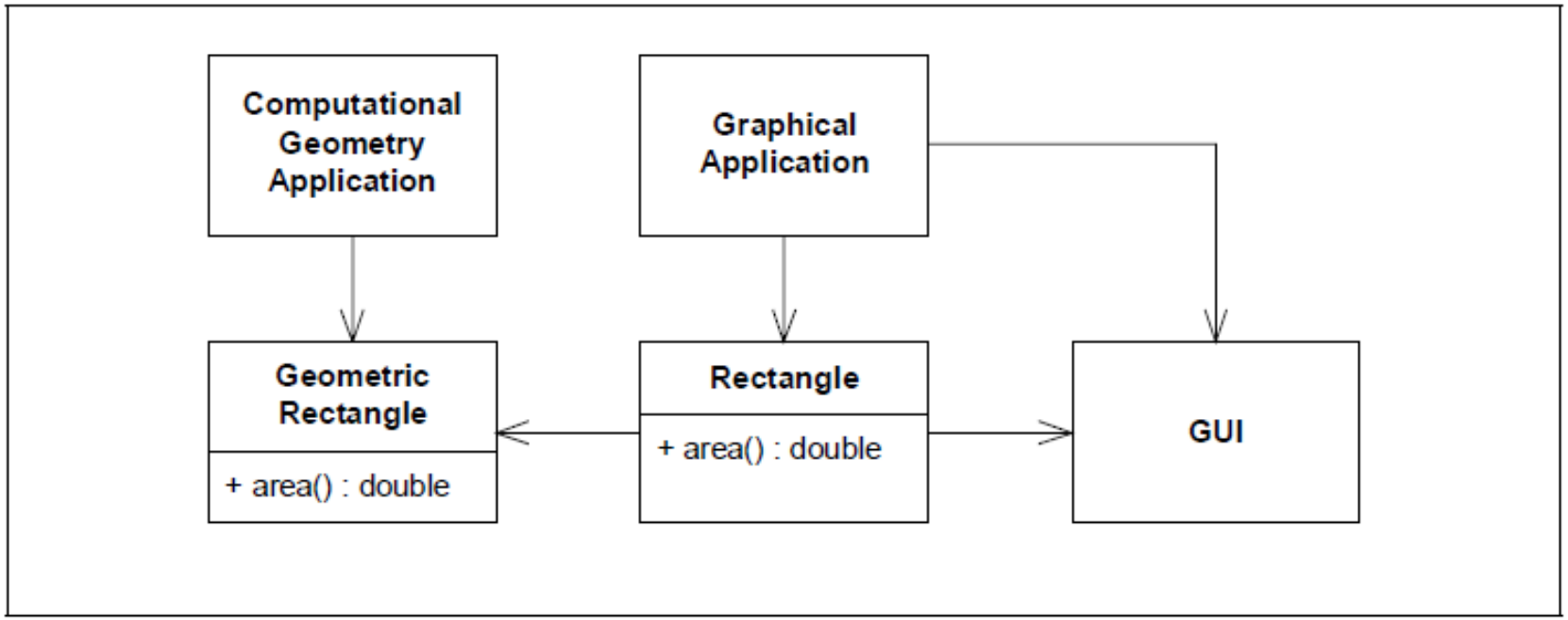
SRP – negativní ukázka



http://www.objectmentor.com/resources/articles/srp.pdf

- Musíme i pro Geometry App vkládat kód GUI
- Při změně požadavků Graphical App musíme překompilovat Geometry App

SRP – pozitivní ukázka



http://www.objectmentor.com/resources/articles/srp.pdf

- Třída Rectangle využívá Geometric Rectangle pro výpočet plochy `area()`

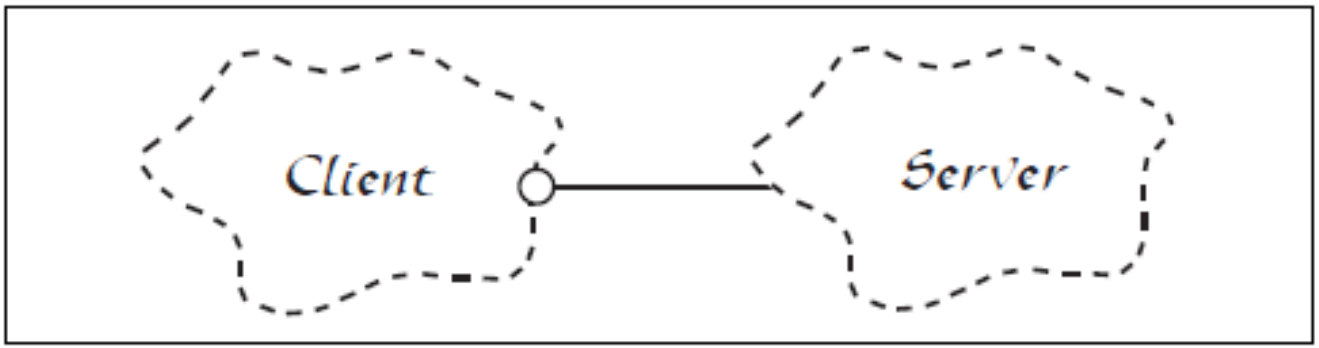
OCP - The Open Closed Principle

- Klíčový princip pro využití abstrakce a polymorfismu
- Třída X by měla být navržena tak, aby:
  - umožnila snadnou modifikaci své funkčnosti vytvořením nového potomka Y v reakci na nové požadavky (Open)
  - pro výše uvedené nebylo nutné měnit kód třídy X (Closed)
- Dosahuje se pomocí abstrakce
  - objekty mají metody navržené pro manipulaci s abstraktními třídami, nikoli konkrétními implementacemi
  - lze tedy přidat nového potomka abstraktní třídy bez změny původního kódu
- http://www.objectmentor.com/resources/articles/ocp.pdf

OCP - The Open Closed Principle (2)

- Porušení principu může vést k
  - změna požadavku vede ke kaskádě změn v kódu
  - přidání nového typy objektu vyžaduje změnu stávajících
- Obecně nelze vždy zajistit 100% část principu týkající se uzavřenosti
  - je potřeba dobře vybrat typ změn požadavků, vůči kterým design uzavře (tzv. strategické uzavření)
  - a tyto požadavky vhodně abstrahovat

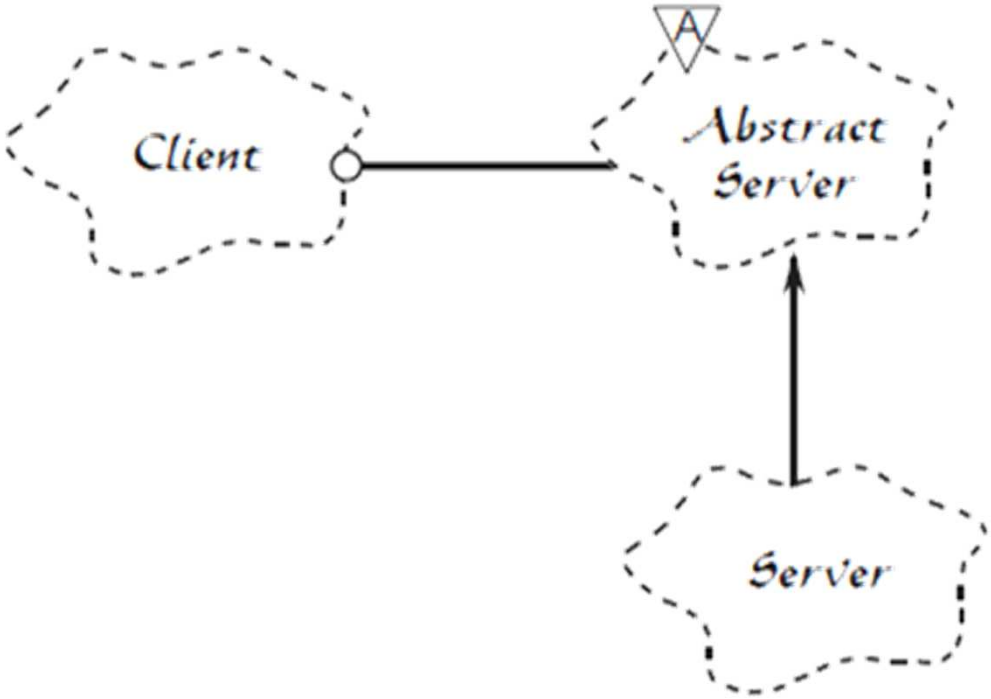
OCP – negativní ukázka



http://www.objectmentor.com/resources/articles/ocp.pdf

- Třída Client používá konkrétní třídu Server
- Nelze použít jinou třídu ServerX bez změny kódu třídy Client

OCP – pozitivní ukázka



http://www.objectmentor.com/resources/articles/ocp.pdf

- Client využívá třídu Abstract Server
- Konkrétní implementace Server je potomek Abs. Server
- Dalším dobrým příkladem je istream a ostream z STL

LSP - The Liskov Substitution Principle

- Pokud třída X používá ukazatel nebo referenci na základní třídu B, pak musí být schopná pracovat beze změny i s potomky třídy B
  - aplikace může nahradit v instanci třídu B za její potomky a X tím nesmí být dotčená
  - princip se týká i potomků třídy B – nesmí očekávat víc, než nabízí B
- Souvisí úzce s OCP
  - pokud aplikace splňuje OCP, tak by měla splňovat i LSP
  - porušení LSP způsobí problémy s dodržením OCP
- http://www.objectmentor.com/resources/articles/lsp.pdf

LSP - The Liskov Substitution Principle (2)

- Někdy označováno jako „Design by contract“
- Dosahuje se pomocí dědičnosti, přetížení a vhodného návrhu
- Porušení principu může vést ke
  - kaskádě změn v kódu při přidání nové třídy
  - chybnému chování aplikace, protože X předpokládá něco o B, co je v potomcích B porušeno

LSP – negativní ukázka

```
#include <typeinfo>

class Shape {};
class Circle : public Shape {};
class Square : public Shape {};

class GraphicScene {
public:
    void DrawShape(Shape& s)  {
       if (typeid(s) == typeid(Square))
              DrawSquare(dynamic_cast<Square&>(s));
       else if (typeid(s) == typeid(Circle))
              DrawCircle(dynamic_cast<Circle&>(s));
    }

    void DrawSquare(Square& s) {
              // draw square
    }
    void DrawCircle(Circle& s) {
              // draw circle
    }
};
```

GraphicScene umí pracovat s objekty typu Shape, ale musí si zjistit pro vykreslení jejich reálný typ

Přidání nového potomka třídy Shape znamená nutnost úpravy i třídy DrawScene

LSP – pozitivní ukázka

```
class Shape {
public:
        virtual void Draw() const = 0;
};
class Circle : public Shape {
public:
        virtual void Draw() {
                // draw circle
        }
};
class Square : public Shape {
public:
        virtual void Draw() {
                // draw square
        }
};

class GraphicScene {
public:
        void DrawShape(Shape& s)  {
                s.Draw();
        }
};
```

Potomci třídy Shape by neměli měnit očekávané chování slíbené na úrovni třídy Shape (např. by neměli začít něco načítat z `cin`)

GraphicScene umí pracovat i s potomky třídy Scene, aniž by byly tyto známy v době psaní GraphicScene

ISP - The Interface Segregation Principle

- Navazuje na DIP a postihuje problém příliš velkých rozhraní
  - takových, kde klientský objekt reálně využívá pouze malou podčást metod, ale je nucen záviset na všech
  - při změně v nevyužívaných částech nutná rekompilace i klienta
- Klientské objekty by neměly být nuceni záviset na rozhraních, které nepoužívají
- Dosahuje se vytvářením více rozhraní
- http://www.objectmentor.com/resources/articles/isp.pdf

ISP - The Interface Segregation Principle

- Zkontrolujte třídy, jejichž rozhraní má mnoho metod
- Udělejte si seznam tříd, které toto rozhraní využívají
  - opravdu používá třída A všechny metody?
- Oddělte metody pro A do samostatného rozhraní
- Např. postupem času vznikne třída s metodami umožňujícími
  - přijímat zprávy (využívá jiný objekt typu producent)
  - odesílat zprávy (využívá jiný objekt typu konzument)
  - zobrazovat zprávy (využívá jiný objekt typu UI)
- Lze rozdělit alespoň do 3 separátních rozhranní
  - např. konzument používá pouze rozhraní produkující zprávy

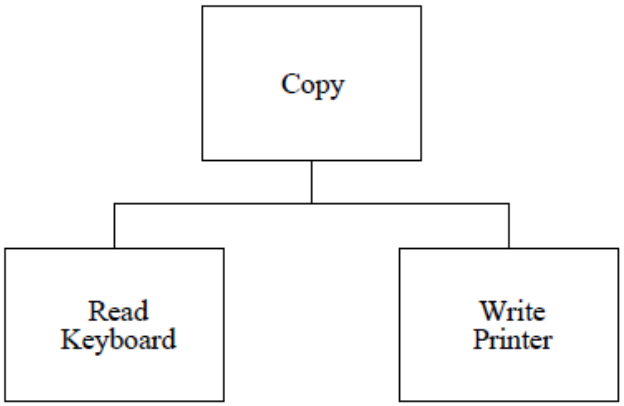
DIP - The Dependency Inversion

- Je výsledkem kombinace OCP a LSP
- Třída by měla být závislá na třídách s obdobnou nebo větší úrovní abstrakce
  - pokud obecná třída závisí na specializované třídě, je obtížné ji použít pro jiný účel
  - abstrakce by neměla záviset na detailech, ale naopak
- Dosahuje se využitím rozhraní
  - v C++ čistě abstraktními třídami

DIP - The Dependency Inversion (2)

- Porušení principu může vést ke
  - k nutnosti změny v obecné třídě pro přidání podpory nového typu
  - snížené flexibilitě obecné třídy – použitelná jen pro to, s čím byla původně implementována
- http://www.objectmentor.com/resources/articles/dip.pdf

DIP – negativní ukázka




```
void Copy() {
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

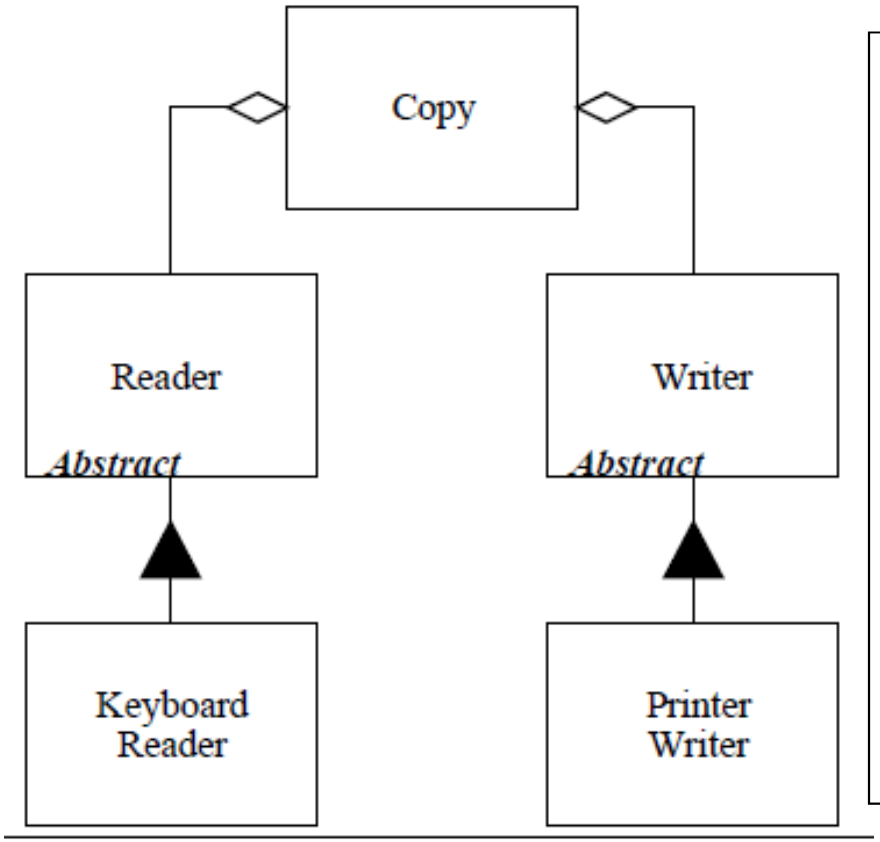
```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev) {
  int c;
  while ((c = ReadKeyboard()) != EOF)
  if (dev == printer)
    WritePrinter(c);
  else
    WriteDisk(c);
}
```

http://www.objectmentor.com/resources/articles/dip.pdf

- Obecný algoritmus kopírování závisí na konkrétní implementaci printer nebo disk
- Změna výstupního zařízení (*disk*) vede ke změně Copy

DIP – pozitivní ukázka

http://www.objectmentor.com/resources/articles/dip.pdf




```
class Reader {
public:
  virtual int Read() = 0;
};
class Writer {
public:
  virtual void Write(char) = 0;
};
void Copy(Reader& r, Writer& w) {
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

- Obdobně funguje např. STL copy algoritmus
  - *std::copy(istream_iterator,istream_iterator,ostream_iterator);*

KISS – Keep It Simple, Stupid!

- Jednoduché, postačující řešení může být lepší než designově rafinované, ale komplikované
  - menší riziko chyby v tom, co snáze chápeme
  - snažší pochopení od dalších vývojářů
- Je nutné hledat kompromis mezi aktuálními a budoucími požadavky
  - předpoklad budoucích požadavků návrh typicky komplikuje
  - přílišné omezení návrhu na aktuální požadavky typicky komplikuje budoucí rozšiřitelnost
- Snažte se průběžně validovat svůj návrh vůči požadavkům a nebojte se refaktorizace

Návrhové vzory

Návrhové vzory - motivace

- Většina programátorských problémů není nová a způsob jejich řešení se opakuje
  - konkrétní implementace jsou lepší a horší
  - postupem vykrystalizovaly vhodné konstrukce pro jejich řešení
- Označováno souhrnným pojmem návrhové vzory
  - není vůbec omezeno pouze na C++
  - většinou jde o jazykově nezávislé konstrukce
- Termín vešel v širokou známost díky knize *Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides (1994)*
  - podle čtyř autorů označováno často jako Gang of Four (GoF)

Návrhové vzory – jak začít?

- GoF kniha popisuje 23 návrhových vzorů
- GoF kniha není organizována z hlediska výuky
  - neřadí od nejsnazších pro nejobtížnější
  - neřadí ani od nejčastějších po nejobskurnější
- Použijeme jiné řazení a omezíme se pouze na vybrané vzory
  - http://mahemoff.com/paper/software/learningGoFPatterns/
- Detailní popis všech vzorů včetně zdrojáků
  - http://sourcemaking.com/design_patterns
  - http://www.vincehuston.org/dp/

Návrhové vzory

- Co považovat za návrhový vzor?
  - principiálně jednoduchý způsob, jak řešit skupiny podobných problémů
  - důležitá je opakovatelnost použití dané konstrukce
- Pozitiva návrhových vzorů
  - existující způsob řešení běžných problémů
  - snížení množství nutných změn později v kódu
  - zřejmě opravdu reálně zlepšují kód a jeho udržovatelnost
    - http://www.labsoftware.com/flahdo/Javapro/DoPatternsHelp.pdf
- Kritika návrhových vzorů
  - návrhové vzory mohou být zbytečně složité na jednoduché problémy
  - potřeba návrhových vzorů může vznikat z nedokonalosti použitého programovacího jazyka
    - http://www.paulgraham.com/icad.html

Formát popisu jednotlivých vzorů

- Popis každého vzoru je dělen do těchto částí:

1. Záměr (Intent)
   - shrnutí účelu použití a přínosu
2. Řešený problém (Problem)
   - shrnutí problému, které je vzorem řešen
3. Discussion
   - co má vzor za cíl a co naopak nemá
4. Structure
   - technický popis realizace vzoru
5. Check list
   - jak postupovat při implementaci konkrétního vzoru
6. Rules of thumb
   - ověřená pravidla, která je vhodné dodržovat
   - uvedení vzoru do kontextu s dalšími vzory (velice přínosné)

Demo: Návrhový vzor ‘Adapter’ - proč

- http://sourcemaking.com/design_patterns/adapter
- Máme existující komponentu A s nekompatibilním rozhraním pro použití jinou naší komponentou B
  - nenabízí metody ve správném formátu (např. jako operátory)
  - má jinou filozofii použití (např. předpokládá předalokovanou paměť)
  - A ani B nechceme (nemůžeme) měnit

Adapter – jak?

- Vytvoříme novou komponentu, která poskytuje rozhraní kompatibilní pro naše použití
- Metody komponenty adapteru volají metody původní komponenty a případně upravují chování
  - např. tvorba operátoru + a jeho mapování na existující metody add()
  - např. alokace potřebné paměti před voláním původních metod
- Implementováno pomocí dědičnosti
  - pokud je rozhraní stejné, potřebujeme jen upravit vnitřní funkčnost
- Nebo agregace
  - pokud měníme rozhraní

Návrhové vzory – jak dál

- Projděte si materiály na webu
  - http://sourcemaking.com/design_patterns
  - http://www.vincehuston.org/dp/
  - a další
- Přečtete si knihy
  - *Design Patterns: Elements of Reusable Object-Oriented Software* a další
- Rozmyslete před další implementací, zda se některý vzor nehodí
  - http://www.vincehuston.org/dp/applicability.html
- Seznamte se s knihovnami, které vzory využívají
  - např. ACE http://www.cs.wustl.edu/~schmidt/ACE.html

Návrhové anti-vzory

```
if(server.is_file_in_database(path)){
        server.set_licence_data_from_database(path);
        char type;
        char constrain;
        bool right_input = false;
        permissions new_permissions = {{FULLY, 0, {0,0,0}}, {FULLY, 0, {0,0,0}}, {FULLY, 0, {0,0,0}}, {FULLY, 0, {0,0,0}}};
        do{
            cout<<endl<<"Enter type of file (t)text/(m)music/(e)executable: ";
            cin>>type;
            switch (type){
              case 't':
                    right_input = true;
                    // display
                    cout<<"Enter constrain for display (n)no/(p)partially/f(fully): ";
                    cin>>constrain;
                    switch (constrain){
                        case 'n':
                            new_permissions.display.constricted = NO;
                            new_permissions.display.count = -1;
                            new_permissions.display.interval.year = -1;
                            break;
                        case 'p':
                            int count;
                            new_permissions.display.constricted = PARTIALLY;
                            cout<<"Count of display (-1 for not set): ";
                            cin>>count;
                            if(count > -1){
                                new_permissions.display.count = count;
                            }
                            else{
                                new_permissions.display.count = -1;
                            }
                            int year, month, day;
                            cout<<"Enter year (-1 for not set): ";
                            cin>>year;
                            if(year > -1){
                                new_permissions.display.interval.year = year;
                                cout<<"Enter month: ";
                                cin>>month;
                                new_permissions.display.interval.month = month;
                                cout<<"Enter day: ";
                                cin>>day;
                                new_permissions.display.interval.day = day;
                            }
                            else{
                                new_permissions.display.interval.year = -1;
                            }
                            break;
                        case 'f': //f is default value
                            break;
                        default:
                            cerr<<"Wrong input type. Please insert n/p/f."<<endl;
                            right_input = false;
                            break;
```

Návrhové antivzory (antipatterns)

- Typické konstrukce, jejichž výskyt signalizuje (budoucí) problémy
  - http://sourcemaking.com/antipatterns
- Mohou vznikat nevhodným návrhem
- Mohou vznikat i postupně s rostoucím projektem
- Je dobré je znát a rozpoznat jejich výskyt
  - a umět odstranit

Návrhový antivzor – špagetový kód

- Vzniká velice často a snadno
  - vyznačuje se rozsáhlým kódem v jediné funkci
  - vyznačuje se častým opakováním kódem s minimální změnou (pokud vůbec)
  - vyznačuje se dlouhou řadou vnořených podmínek
- A určitě jej nelze vzít a použít jinde
- Pomůže pravidelný Refactoring
  - opakované části kódu přesouváme do nové funkce
  - dlouhý kód rozdělujeme do více funkcí
  - …
  - http://sourcemaking.com/refactoring

Shrnutí

- Šablony umožňují typově nezávislé programování
  - snažší na pochopení, než např. OOP
  - pokud používáte často makra, naučte se i šablony
- Návrhové principy a vzory
  - časem prověřené postupy, které vám mohou ušetřit práci při rozšiřování programu
  - seznamte se s nimi a kontrolujte porušení v kódu