KIV/ZSWI 2003/2004
Přednáška 7

V klasickém životním cyklu vývoje SW pod pojem ”návrh” spadají dvě aktivity, které jsou umístěny mezi analýzou požadavků a kódováním programu:

* návrh architektury systému - rozděluje systém do podsystémů či komponent; o návrhu architektury jsme mluvili na minulé přednášce
* podrobný návrh systému - každá součást systému je popsána natolik podrobně, aby to postačovalo pro kódování

V dalším textu se budeme věnovat podrobnému objektově orientovanému návrhu a jeho implementaci.

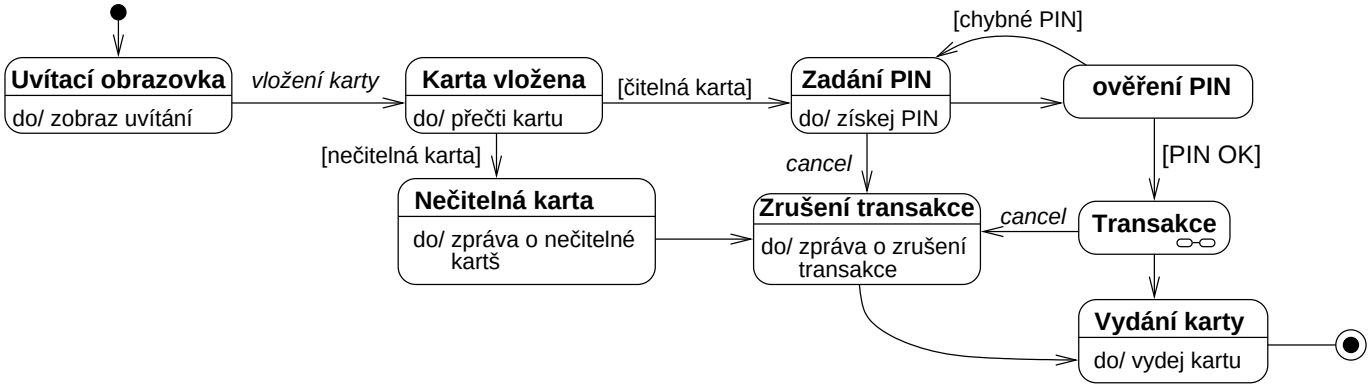
Objektově orientovaný návrh

Přehled:

* vstupem jsou analytické třídy představující role, které mohou být pokryty jednou nebo více třídami návrhu; analytická třída se v návrhu může stát jednou třídou, částí třídy, agregovanou třídou, skupinou spřízněných tříd, asociací, naopak asociace třídou apod.
* vytvoření návrhových tříd (design classes)
* definice operací
* definice asociací, agregací a kompozic

* pro analytické třídy budeme vytvářet jednu nebo více návrhových tříd
 - návrh hraničních tříd
  . pokud jsou k dispozici nástroje pro návrh GUI, pak jedna hraniční třída bude odpovídat jednomu oknu nebo formuláři
  . jednu třídu pro každé API nebo protokol
 - entitní (datové) třídy
  . jsou často pasivní a perzistentní, musíme pro ně vybrat způsob implementace: v souboru, v relační databázi (z perzistentních tříd vytvoříme tabulky atd.); nejsou-li třídy perzistentní, pak budou implementovány v paměti
 - řídící třídy
  . obsahují aplikační logiku

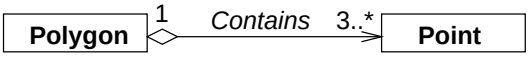
* vyplňování tříd 1: definice operací
 - standardní konstruktory se předpokládají => z návrhu je vynecháváme
 - operace které čtou a nastavují veřejné atributy se (většinou) předpokládají
 - hledáme operace - prvním vodítkem může být již vytvořený seznam sloves
 - další způsob - získat operace z popisu interakce objektů
  . nakreslíme diagramy spolupráce nebo sekvenční diagramy (resp. upravíme diagramy získané ve fázi analýzy, doplníme zasílané zprávy)
  . z nich zjistíme jaké stimuly musí být objekt schopen přijmout, navrhneme odpovídající operace
 - další možnosti, co může být operace:
  . inicializace nově vytvořené instance včetně propojení s asociovanými objekty
  . případné vytvoření kopie instance (pokud je zapotřebí)
  . případný test na ekvivalenci instancí (pokud je zapotřebí)
 - operace popíšeme: název, parametry, návratová hodnota, krátký popis, viditelnost
 - pokud je algoritmus složitý, popíšeme metodu (= implementaci operace) nebo nakreslíme stavový diagram objektu nebo operace; např. stavový diagram pro bankomat:



- základní kritérium: každá metoda by měla dělat jednu věc dobře

* vyplňování tříd 2: definice atributů
- při hledání atributů můžeme vycházet z logických atributů (produkt analýzy), popisujících co je potřebné pro uchování stavu objektu
- další možnost - jaké atributy jsou třeba pro implementaci operací
- atributy v návrhu mají být "jednoduché" (int, boolean, float, ...) nebo mají mít sémantiku hodnoty (tj. být nezměnitelné, např. String)
  . jinak použijeme asociaci
- atributy popíšeme: jméno, typ, počáteční hodnota, viditelnost
  . měli bychom se snažit o skrývání informací - soukromé atributy (viditelné pouze pro potomky = protected "#")
- ověříme, že všechny atributy jsou potřebné
- pokud zjistíme, že se atributy a operace dělí do dvou tříd, které spolu příliš nesouvisejí, pak se pravděpodobně jedná o dvě různé třídy - měli bychom třídu rozdělit

* definice asociací, agregací a kompozic
- stejně jako v analýze, máme ale podrobnější informace o chování
- navíc můžeme přidat tzv. průchodnost (navigability) - od kterého ke kterému objektu budeme chtít snadno přecházet
  . označuje se šipkou v asociačním vztahu



* pokud třída obsahuje více než cca 10 atributů, 10 asociací nebo 20 operací, pak asi není dobře navržená a potřebuje rozdělit

* definice zobecnění (hierarchie dědičnosti)
- stejně jako v analýze, tj. společné vlastnosti vyjmeme do nadtříd
- pokud má některá třída sjednocení atributů jejích podtříd (místo průniku), zřejmě spolu podtřídy ve skutečnosti nemají nic společného
- hierarchie by měla být vyvážená, tj. neměly by být třídy pro které je hierarchie neobvykle plochá nebo naopak hluboká

* kontrola modelu
- ověřit realizace případů použití, v návrhu nám nesmí chybět chování potřebné pro některý případ použití

* volitelně vytváření balíčků
- buď pro lepší srozumitelnost (strukturování), nebo pro izolaci částí které se budou častěji měnit (případně obojí)
- do balíčku vkládáme funkčně příbuzné třídy; např. pokud by změna chování jedné třídy způsobila změnu chování druhé, jsou třídy funkčně příbuzné
- třídy uvnitř balíčku mohou být veřejné (public) nebo soukromé (private)
  . veřejná třída může mít asociace s libovolnou jinou třídou; veřejné třídy tvoří rozhraní balíčku
  . soukromá třída může být asociována pouze se třídami uvnitř stejného balíčku
- pokud má třída v jednom balíčku asociaci se třídou v jiném balíčku, pak balíčky na sobě navzájem závisejí - modeluje se vztahem závislosti (přerušovaná šipka)

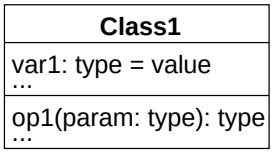
- v jednom diagramu můžeme zakreslit i závislosti mezi třídami apod.

* tím končí návrh a můžeme přejít k implementaci

Objektově orientovaná implementace

Poznámka: Obrázky v této části nejsou platné UML, ale snaží se naznačit způsob implementace návrhu.

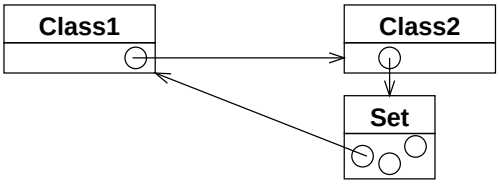
Kostru implementace v objektově orientovaném programovacím jazyce můžeme snadno vytvořit z diagramu tříd:



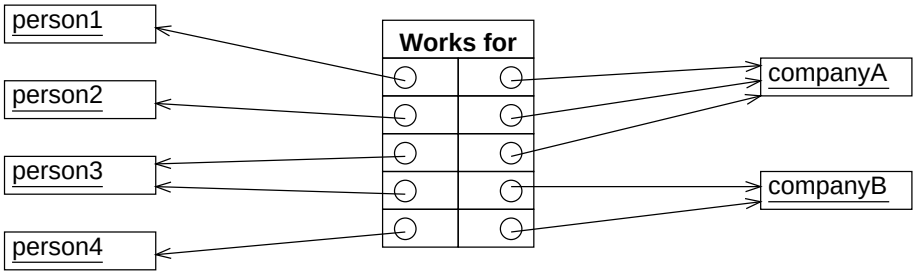

```
public class Class1 {
    // konstruktory
    public Class1() {};
    public Class1(type param) { setVar1(param) };

    // veřejné operace get a set, soukromá proměnná
    private type  var1 = value;
    public type getVar1()           { return var1; }
    public void setVar1(type param) { var1 = param; }

    // operace pro každou metodu
    public type op1(type param) { ... }
}
```

* převod asociací - asociace nejsou přímo podporovány OO programovacími jazyky
 - pokud je násobnost 1 a průchodnost jen jedním směrem (tj. jedním směrem přístup podstatně častěji), implementujeme asociaci jako odkaz na objekt (referenci objektu, v některých jazycích jako ukazatel na objekt)

 - pokud je násobnost 1 a průchodnost oběma směry - implementujeme jako vzájemné odkazy
 - pokud je násobnost 1:M - množinu odkazů udržujeme např. ve stromu apod.



 - pokud je k asociaci přidán nebo z ní odebrán link, je třeba upravit oba odkazy
 - pokud jsou změny asociací častější než vyhledávání, je vhodná implementace pomocí asociačního objektu (asociačního objekt implementuje např. hashovanou tabulku, prvek tabulky obsahuje dvojici (ObjA, ObjB))



- asociace M:N nemohou být přiřazeny jako atribut ani jednomu konci, tj. pro ně musíme také vytvořit asociační třídu
- ternární a vyšší asociace buď rozložíme na binární (pokud to jde) nebo opět vytvoříme asociační třídu (N-ární asociace naštěstí nejsou v praxi časté)

* převod agregátů a kompozic
 - agregáty implementujeme stejně jako asociace, označení v diagramu je pouze ”komentář” ukazující, že vztah je bližší než u ostatních asociací
 - kompozice = existenční závislost součástí
  . v destruktoru zrušíme součásti
  . pokud mají součásti stejnou dobu života, vytvoříme je v konstruktoru

* dědičnost se na konstrukce OO programovacího jazyka mapuje přímočaře, například v jazyku Java:

```
public class Podtřída extends Nadtřída {
  ...
}
```

Implementace objektového návrhu v neobjektovém jazyce

Výše uvedené koncepce lze implementovat i v neobjektovém jazyce, jako je např. C nebo Pascal - implementace ovšem nebude tak přímočará.

Níže uvedeným způsobem byly implementovány některé SW systémy před příchodem spolehlivých překladačů jazyka C++.

Implementace probíhá v následujících krocích:

* překlad tříd do datových struktur
* vytvoření konstruktorů
* implementace metod

Překlad tříd do datových struktur

* třídu obvykle implementujeme jako blok atributů - typ ”struct” v jazyku C, typ ”record” v Pascalu, například:

```
struct Čára {
    int x1, y1;
    int x2, y2;
};
```

 - proměnná reprezentující objekt musí reprezentována ukazatelem na záznam, aby odkaz bylo možné předávat a sdílet, např.:

```
struct Čára *čára;
```

Vytvoření konstruktorů

* objekty mohou být alokovány staticky nebo dynamicky (na hromadě)
 - staticky - pro globální objekty, pokud předem víme, jaký počet objektů daného typu bude zapotřebí
 - dynamicky (pomocí ”malloc()” v C nebo ”new” v Pascalu) - pokud už není

zapotřebí, musí být explicitně dealokován (”free()” v C, dispose v Pascalu)

```
struct Čára *create_čára(x1, y1, x2, y2)
{
    struct Čára *čára;

    čára = (struct Čára*) malloc(sizeof(struct Čára));
    čára->x1 = x1;
    čára->y1 = y1;
    čára->x2 = x2;
    čára->y2 = y2;

    return čára;
}
```

Metody
.....

* pro pojmenování metod se obvykle používá konvence, která popisuje, které třídě metoda patří (jméno třídy + dvě podtržítka + jméno metody)
* každé metodě je třeba předat odkaz na objekt, nad kterým je operace prováděna (”self” nebo ”this” v OO jazycích); konvence je předávat ”self” jako první argument

```
void Čára__posun(struct Čára *self, int posun_x, int posun_y)
{
    self->x1 += posun_x;
    self->y1 += posun_y;
    self->x2 += posun_x;
    self->y2 += posun_y;
}
```

Dědičnost
....

V neobjektovém jazyce je nejlepší se dědičnosti vyhnout, tj. přestrukturovat diagram tříd nebo jeho části tak, aby dědičnost nebyla zapotřebí. Abychom se dědičnosti vyhnuli, můžeme také použít následující dvě možnosti:

1. ”zděděnou” operaci implementujeme znovu jako samostatnou metodu
   - v tomto případě ovšem dochází k duplikaci kódu
2. místo dědění z nadtřídy jsou ”děděné” atributy a operace implementovány samostatným objektem
   - podtřída bude obsahovat odkaz na tento nový objekt a všechny ”děděné” operace deleguje tomuto novému objektu

Poznámka pro zajímavost (implementace dědičnosti v neobjektovém jazyce)

Pokud musíme implementovat dědičnost a potřebujeme polymorfismus, pak se třídy implementují pomocí dvou záznamů:

* první záznam obsahuje atributy objektu (podobně jako ve výše uvedeném příkladu)
  - jako první položku navíc obsahuje ukazatel na druhý záznam, obsahující odkazy na metody třídy (tento druhý záznam je sdílen všemi instancemi třídy)
  - počáteční položky prvního záznamu jsou v potomkovi stejného typu a ve stejném pořadí jako v nadtřídě (to umožňuje, aby nad nimi pracovaly i metody nadtřídy)

```
struct Shape {  /* abstraktní třída Shape (česky ”tvar”) */
    struct ShapeClass *class;  /* odkazy na metody třídy */
    int x, y;                  /* umístění tvaru */
};

struct Line { /* konkrétní třída Line (česky ”čára”) */
    struct LineClass *class;   /* odkaz na metody třídy */
    int x, y;                  /* opakuje atributy z abstraktní třídy */
```

```
        int x2, y2;                /* nové atributy */
    };
```

* pro zjištění skutečných metod máme druhou datovou strukturu, obsahující ukazatele na metody dané třídy:

```
    struct ShapeClass {
        void (*move)(int x, int y);  /* přesun objektu */
        void (*scale)(int newsize);  /* změna velikosti objektu */
    };

    struct LineClass {
        void (*move)(int x, int y);  /* přesun objektu */
        void (*scale)(int newsize);  /* změna velikosti objektu */
        void (*setColor)(int color); /* nová metoda - změna barvy objektu */
    };
```

- vytvoříme a inicializujeme proměnnou, obsahující odkazy na metody:

```
    struct LineClass LineClass =
    {
        Shape__move,   /* dědíme po rodičovi */
        Line__scale,   /* překrytí metody vlastní metodou */
        Line__setColor /* nová metoda */
    };
```

- metodu bychom pak mohli vyvolat následovně:

```
     line->class->scale(5); /* vyvolání metody */
```

* konstruktor třídy ”Line” by mohl vypadat takto:

```
    struct Line *create_line(int x1, int y1, int x2, int y2)
    {
        struct Line *line;

        line = malloc(sizeof(struct Line));
        line->class = &LineClass;
        line->x = x1;
        line->y = y1;
        line->x2 = x2;
        line->y2 = y2;

        return line;
    }
```

* použití:

```
     struct Line *line = create_line(1, 1, 2, 2); /* vyvolání konstruktoru */
```

[]

Návrhové vzory

* ”vzory” (patterns) jsou řešení problémů, které se při vytváření SW systémů často opakují
* vzory existují na různých úrovních:
  - návrhové vzory - dokumentují řešení problémů na úrovni návrhu
  - analytické vzory - dokumentují řešení doménových problémů (o nich zde nebudeme z časových důvodů mluvit)
  - architektonické vzory - řešení problémů vysokoúrovňového návrhu

V následujícím textu uvedu několik často používaných návrhových vzorů.

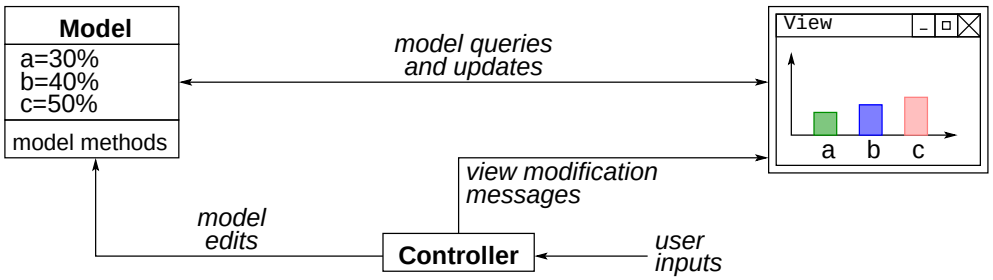
Termín návrhové vzory (design patterns) byl zaveden stejnojmennou knihou, která byla přeložena také do češtiny:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns.
Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.
(český překlad: Návrh programů pomocí vzorů. Grada 2003.)

Co je návrhový vzor

Jako příklad pro ilustraci uvedu trojici tříd nazývaných
Model/View/Controller (MVC) pro návrh interaktivních aplikací.

* myšlenka pochází z prostředí jazyka Smalltalk (Goldberg a Robson 1983)
* fčnost aplikace rozdělíme na 3 typy objektů, mezi objekty je vztah
  vydavatel-předplatitel (publisher-subscriber)
 - ”model” je objekt aplikace, zapouzdřuje data která mají být zobrazena
 - ”view” (česky ”náhled”) je prezentace modelu na obrazovce
  . ve chvíli kdy dojde ke změně stavu modelu, oznámí to model všem ”view”
    které na něm závisejí; view zjistí od modelu nové hodnoty a znázorní je
  . způsobů prezentace může být více, je možné ho změnit např. na koláčový
    graf nebo tabulku aniž by to ovlivnilo model nebo controller
 - ”controller” definuje způsob jak uživatelské rozhraní reaguje na vstup
  . způsob reakce na uživatelský vstup je možné změnit aniž by to ovlivnilo
    model nebo view; například místo klávesových zkratek použijeme výběr z
    menu



* v tomto příkladu je vidět několik dalších návrhových vzorů:
 - návrhový vzor vydavatel-předplatitel (angl. publisher-subscriber nebo
   subject-observer)
  . vydavatel = instance, v níž může událost vzniknout
  . předplatitel = instance, která má zájem reagovat na určitou událost
  . předplatitel si u vydavatele zaregistruje metodu, která má být v důsledku
    události vyvolána
  . předplatitelů může být i více
  . nastane-li událost, vydavatel vyvolá registrované metody předplatitelů
 - vztah mezi ”view” a ”controller” je příklad návrhového vzoru ”stategie”
   atd.

Iterátor

* jeden z nejjednodušších a nejčastěji používaných návrhových vzorů je
  iterátor
* problém: agregovaný objekt (jako je např. seznam) by měl umožňovat
  procházení, aniž by k tomu uživatel musel znát vnitřní strukturu agregátu
  (která se může změnit)
 - iterátor udržuje informaci o aktuálním objektu v agregátu, umí se posunout
   na následující prvek agregátu a prvek poskytnout uživateli
 - moderní OO programovací jazyky obsahují podporu iterátorů; např. v jazyce
   Java se používá rozhraní Iterator:

```
public interface Iterator
{
    boolean hasNext(); // Vrací ”true” pokud jsou další prvky.
    Object next();     // Vrací další prvek agregátu.
}
```

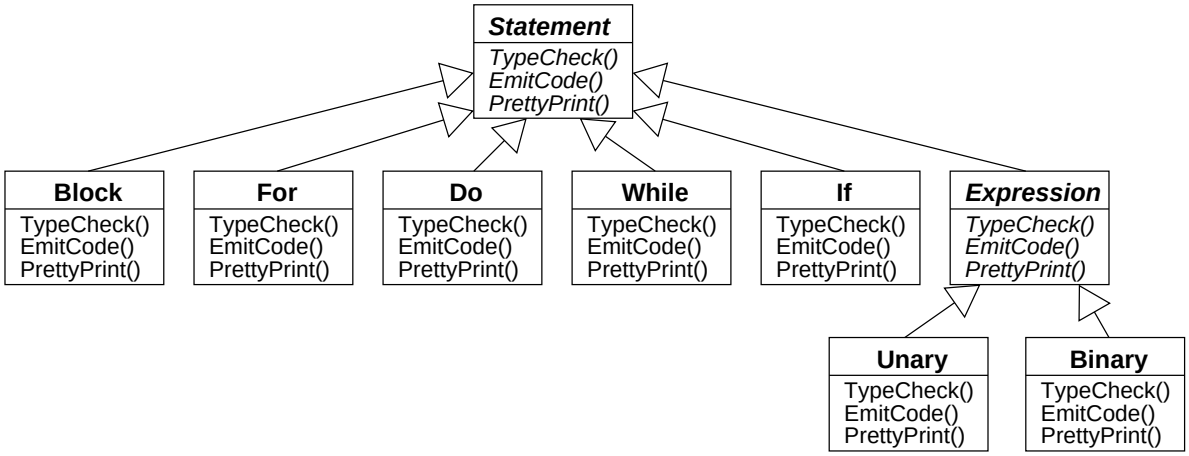
- v Javě poskytují iterátor třídy Set, List, Map, atd.; lze psát např.

```
for (Iterator e = v.iterator(); e.hasNext();) {
    System.out.println(e.next());
}
```

Návštěvník
.....

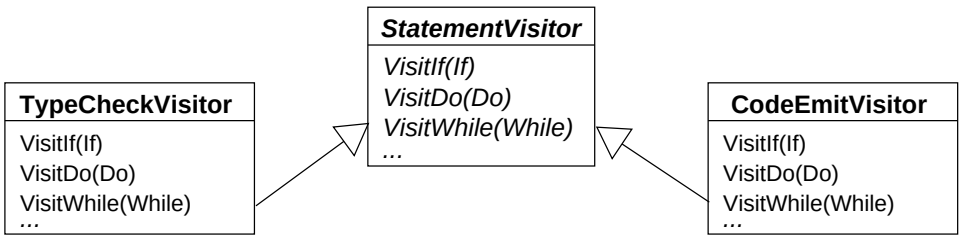
* představme si program, který vytváří strukturu objektů, nad kterými se provádějí další operace
 - například překladač vytváří syntaktický strom odpovídající zdrojovému programu
 - nebo formátovací program vytváří strom objektů odpovídající struktuře vstupního textu

* nad strukturou objektů budeme chtít provádět různé operace
 - například překladač - kontrola syntaktické správnosti, optimalizace, generování kódu, měření vlastností zdrojového textu apod.



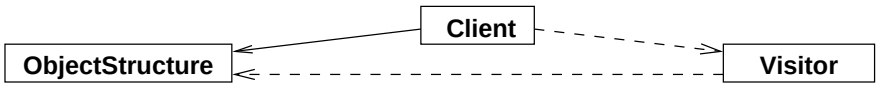
- pokud potřebujeme nad objekty struktury provádět mnoho různých operací, bylo by lepší, kdybychom mohli nové operace mohli přidávat samostatně a kdyby původní strom objektů byl nezávislý na operacích, které nad ním budou prováděny (abychom pro každý typ prvku nemuseli implementovat všechny operace - obtížnější udržovatelnost)

* řešení - příbuzné operace každé třídy zabalíme do samostatného objektu, nazývaného návštěvník (visitor)
 - při průchodu stromem objektů předáváme návštěvníka jednotlivým prvkům
 - navštívený prvek pošle návštěvníkovi požadavek, v názvu volané metody je zakódována třída prvku, prvek předá sám sebe jako argument
 - například TypeCheckVisitor bude při procházení stromu objektů volat jejich metody Accept()
  . metoda Accept() vyvolá odpovídající metodu návštěvníka, např. objekt třídy If vyvolá VisitIf, objekt třídy While vyvolá VisitWhile apod.
  . návštěvník může při průchodu strukturou akumulovat stav, čímž eliminujeme potřebu globálních proměnných



- jinými slovy - definujete dvě hierarchie tříd, jednu pro objekty nad kterými se provádějí operace (If, While...) a jednu pro návštěvníky, kteří provádějí operace nad objekty (TypeCheckVisitor, CodeEmitVisitor,

PrettyPrintVisitor...)
- kromě návštěvníka a struktury objektů hraje v systému často roli ještě klient, který např. vyvolá vytvoření hierarchie objektů a žádá zpracování této hierarchie



Jedináček (singleton)
....

* v některých případech potřebujeme třídu, která bude mít jedinou instanci
 - např. pokud celý systém sdílí jednoho správce oken, jednu tiskovou frontu
 - třída může zajistit, bude vytvořena její jediná instance následujícím způsobem:
  . skryjeme konstruktor třídy (bude protected)
  . pro vytvoření instance poskytneme operaci třídy
  . pokud instance existuje, vrátí ji; pokud neexistuje, vytvoří a vrátí ji

```
public class Singleton {
    protected static Singleton instance = null;

    private Singleton() {}

    public static Singleton instance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

* klienti si vytvoří instanci pomocí Singleton.instance()
* pozor, jedináček de facto zavádí globální proměnné (i když s možností vytvářet potomky apod.), proto by jeho použití mělo být uvážené

Výše uvedená kniha (Design patterns) uvádí katalog 23 základních návrhových vzorů, se kterými se můžete často setkat, proto jí doporučuji k prostudování.

❇