KIV/ZSWI 2003/2004
Přednáška 4

Úvod do OO terminologie

Za počátek objektově orientovaného programování se považuje vznik jazyka Simula67 (Norwegian Computing Centre, asi 1967). Prvním čistě objektově orientovaným jazykem byl jazyk Smalltalk-80, vyvinutý v Xerox PARC (čistě objektově orientovaný jazyk = všechno je objekt, včetně základních datových typů). Později se objevily další OO jazyky (C++, Objective C, Eiffel, Python, Java (čti [džava]), C# (čti [sí šarp] nebo česky [cis])...

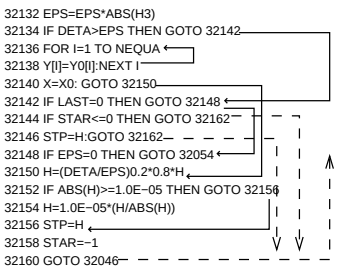
Objektová orientovanost se postupně rozšířila do všech fází vývoje SW - vznikly a stále ještě vznikají objektově orientované metodiky pro analýzu, návrh, implementaci, testování, zpětné inženýrství (reverse engineering) a přepracování kódu (refactoring) atd.

V dalším textu si nejprve uvedeme základní motivaci, terminologii a poté metodiku OO analýzy a návrhu.

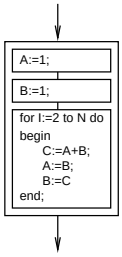
Proč objektově orientované programování?

Vývoj nástrojů pro tvorbu SW (např. programovacích jazyků) odráží nutnost vytvářet a zacházet se stále rozsáhlejšími celky. Nemáme-li k dispozici přiměřenou technologii, nazýváme takový stav ”krize”; řešením krize je změna paradigmatu (např. postupně rozšíření strukturovaného, objektově orientovaného a komponentově orientovaného programování).

* historicky první programovací jazyky (asseblery a FORTRAN) byly nestrukturované
 - základní řídící konstrukcí byl skok - příkaz GOTO (spustí provádění instrukcí od zadaného návěští)
 - srozumitelnost silně klesá s velikostí projektu
 - proto snaha vnést strukturu - tzv. strukturované programování



a) nestrukturovaný kód v jazyku BASIC



b) strukturovaný kód v jazyku Pascal

* strukturované programování podporují všechny moderní imperativní programovací jazyky
 - ve strukturovaném kódu snadněji vidíme, které části jednoho podprogramu se k sobě vztahují a jak
 - s rostoucí velikostí programů ale opět problém: příliš mnoho věcí a vztahů, které musíme brát do úvahy najednou
  . ”věci” = podprogramy, funkce a data
  . ”vzájemné vztahy” = které podprogramy na sobě závisejí, jaké používají datové struktury
 - proto sdružování souvisejících podprogramů a dat do modulů tak, aby změna jednoho modulu měla minimální dopad na obsah ostatních modulů

a) jeden celek b) totéž rozděleno do modulů

```
* strukturovaně navržené systémy se ale časem obtížně udržují
 - při strukturovaném programování se zabýváme zvlášť funkcemi a zvlášť daty
   . fce jsou aktivní, mají chování
   . data jsou pasivní, jsou zpracovávána funkcemi
   . problém: fce musejí znát strukturu dat
   . pokud se změní struktura dat (změna požadavků), je třeba změnit všechny
     fce pracující s datovou strukturou
   . pokud máme zpracovávat podobná data, do všech fcí musíme přidat podmínky
   . tím vzrůstá "entropie" (neuspořádanost) SW; pokud dosáhne určitého
     limitu, cena modifikací naroste tak, že je ekonomicky neobhajitelné
     systém nadále udržovat => je nutné systém přestrukturovat

* související problém: sémantická mezera mezi externím a interním pohledem
  na systém (tj. mezi aplikační doménou a vytvářeným systémem)
 - potřebujeme, aby malá změna požadavků vedla k malé změně systému
   . pozorování: "co" má systém dělat se mění méně než "jak" to má systém dělat
   . řešení - mapování mezi skutečností, konceptuálním modelem a systémem by
     mělo být co nejjednodušší (např. faktura -> datový typ "faktura")

* další problém - znovupoužitelnost kódu v dalších aplikacích
 - proč knihovny procedurálních programovacích jazyků (C, Pascal, FORTRAN...)
   obsahují pouze fce poměrně nízké úrovně?
 - v klasických procedurálních jazycích nelze vysokoúrovňovou funkčnost
   snadno přenést z jednoho kontextu do druhého - nutno vytvářet znovu
 - řešení: principy abstrakce, zapouzdření a dědičnosti podporované
   programovacím jazykem = objektově orientované programování (viz dále)

* další krok po OO stejným směrem bude pravděpodobně komponentově orientované
  programování a objektově orientované aplikační rámce


Základní koncepce OO technologie
--------------------------------

* pojem "objektově orientovaný" znamená přibližně to, že SW je organizován
  jako množina objektů, které spolu komunikují
 - objekty slučují data a nad nimi pracující fce => změny budou lépe
   lokalizovány
 - tím, že problém strukturujeme do objektů existujících v aplikační doméně
   => menší sémantická mezera mezi aplikační doménou a modelem
 - některé objekty mohou sloužit jako znovupoužitelné komponenty (např.
   univerzální kontejner)
 - pokud celý vývoj probíhá objektově orientovaným (dále jen "OO") způsobem,
   můžeme pro všechny fáze vývoje používat stejnou terminologii a grafickou
   notaci (i když objekty budou na různé úrovni abstrakce)

* na začátku analýza požadavků = zabývá se vytvářením konceptuálního modelu
  studovaného systému
* při postupu k implementaci zjemňujeme předchozí model přidáváním detailů a
  nových objektů poskytujících funkčnost
* protože informace je ukryta v objektech, může být rozhodnutí o konkrétní
  realizaci dat odloženo až na implementaci
* v některých případech můžeme obdobně odložit rozhodnutí o distribuci
  objektů a zda budou sekvenční nebo paralelní - může záviset na prostředí
  kde budou běžet

* např. v analýze požadavků modelujeme systém jako množinu interagujících
  objektů aplikační domény, tj. entit a operací sdružených s řešeným
  problémem (např. nemocnice: pacienti, sestry, lékaři, výkony...)
* v OO návrhu vytváříme OO model SW systému, který bude požadavky
  identifikované v analýze implementovat
 - některé objekty aplikační domény budou odpovídat objektům implementace,
```

   ale při postupu k implementaci je třeba vytvářet další objekty
* při OO implementaci realizujeme systém v OO jazyce jako je např. Java

Poznámka (modelovací jazyk UML)

 Podobně jako se v elektrotechnice používají schémata, používají se při
 návrhu SW systémů modely. Různé modely zachycují různé klíčové vlastnosti
 studovaného systému a vynechávají detaily (podobně jako el. schémata
 vynechávají velikost a rozmístění prvků). Téměř všechny modely mají nějakou
 grafickou notaci s textovým popisem.

 Pro popis objektově orientovaných modelů budeme používat notaci UML (Unified
 Modeling Language, 1996), která se stala de-facto standardem pro modelování
 OO systémů. UML podporuje mnoho CASE nástrojů (Rational Rose, Together,
 MagicDraw, Fujaba...).

 Před vznikem UML existovalo nejméně 50 různých notací, některé z nich můžete
 ještě najít např. v nových překladech starší literatury.

[]

V literatuře existují určité neshody v tom, co přesně znamená pojem
”objektově orientovaný”, nicméně většina autorů se shoduje na těchto
základních principech:

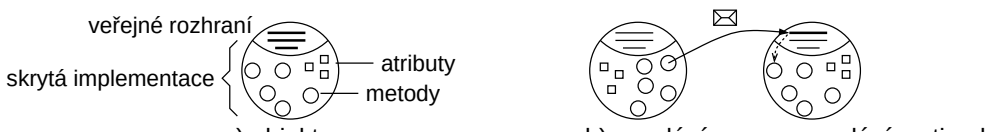
1. abstrakce
2. zapouzdření
3. dědičnost
4. polymorfismus

Abstrakce
....
[Rumbaugh: OMT, Sigfied: Understanding OOSE, Cox: OOP]

* abstrakce = zaměření se na podstatné vlastnosti entity a zanedbání
  detailů, které jsou pro daný účel nepodstatné
 - při OO modelování umožňuje zaměřit se nejprve na to, co objekt je a co
   dělá
 - detaily (např. návrh a implementaci) doplníme až budeme problému lépe
   rozumět
 - nakonec se sice musíme detaily zabývat, ale můžeme je např. vhodně
   uspořádat za použití vrstvené abstrakce (vrstvená abstrakce =
   vysokoúrovňová abstrakce je popsána pomocí nízkoúrovňových abstrakcí)

Zapouzdření
....
[Gosling-McGilton: The Java Language Environment. 1996]

* zapouzdření (encapsulation) = entita má dobře definované rozhraní a ukrytou
  vnitřní reprezentaci
* v OO řešení problému je jednotkou zapouzdření objekt
 - zveřejňuje rozhraní - množinu nabízených operací
 - skrývá implementaci
 - objekt můžeme používat podle jeho specifikace nezávisle na vnitřní
   implementaci (která se může změnit)



a) objekt b) vyvolání operace zasláním stimulu

Příklad:

 Příkladem zapouzdření v reálném světě je např. auto: ovládá se pomocí
 volantu, spojky, brzdy, plynu atd. - téměř všechna auta ovládáme podobně
 nezávisle na jejich vnitřní implementaci.

Objekt
......

* objekt je entita která má vlastní identitu, stav a množinu operací, které pracují se stavem (mění stav, zjišťují stav, vyvolají určité chování)
* každý objekt má jedinečnou identitu, její pomocí se na objekt odkazujeme po celou dobu existence objektu
* stav objektu je reprezentován množinou atributů objektu
* operace sdružené s objektem poskytují služby jiným objektům (klientům)
* klienti vyvolají operaci zasláním stimulu (stimuly se často nazývají ”zprávy”, i když nemusejí nutně nést informaci)

Příklad (objekty na vysoké úrovni abstrakce):

Objekty mohou odpovídat objektům reálného světa. Objektem na určité úrovni abstrakce může být např. ”Honza Vomáčka”, jeho atributy budou např. ”věk 18 let”, ”výška 177 cm”, operacemi může být ”jez”, ”pij” apod.

[]

Třída
.....

* třída objektu popisuje skupinu objektů stejnou strukturou (atributy), chováním (operacemi), vztahy k ostatním objektům a významem
 - objekty dané třídy mají obdobný význam - např. i když kůň i stáj mají atributy ”stáří” a ”cena”, pravděpodobně budou patřit do různých tříd
 - sdružováním objektů do tříd abstrahujeme společné vlastnosti objektů

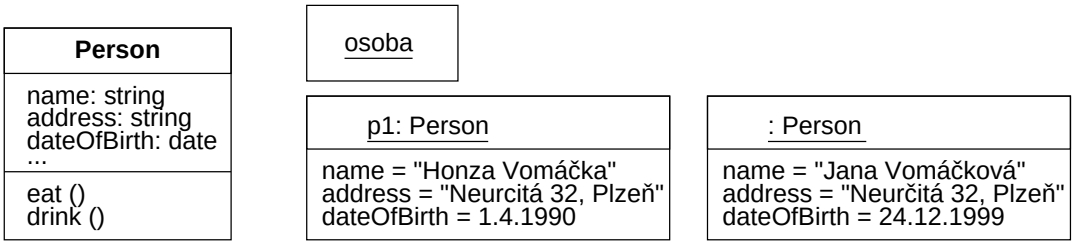
* objekty stejné třídy = instance třídy, každá instance má jedinečnou identitu
* atributy i operace objektů jsou deklarovány jejich třídou
* v OO jazycích se objekty vytvářejí podle definice třídy

Příklad (třída na vysoké úrovni abstrakce):

Objekty ”Honza Vomáčka” a ”Kateřina Drsná” mohou být instancemi třídy Člověk. I když se ”Kateřina Drsná” vdá a bude mít nové jméno, její identita se tím nezmění (identita je nezávislá na stavu objektu).

[]

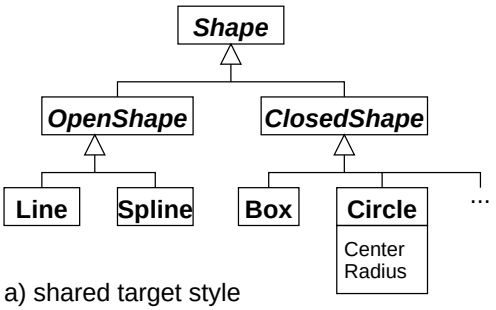
* notace pro třídy a instance v UML:
 - třídy - znázorňují se jako obdélník se jménem třídy (jméno tučně a vycentrovat) a dvěma volitelnými sekcemi
  . střední sekce = atributy objektu (”attributes”)
  . spodní sekce = operace objektu (”operations”)
 - instance - podobně, jméno ve tvaru: ”objekt” : ”Třída” (podtržené), lze zkrátit i jen na ”objekt” nebo : ”Třída” (viz obrázek)
  . případně atributy které nás zajímají a jejich hodnota



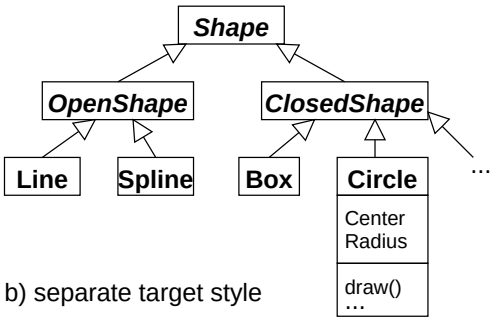
a) třída b) instance třídy

* termín ”operace” se používá pro akci, termín ”metoda” pro implementaci operace
* v OO jazycích se metody se spouštějí zasláním zprávy buď samotné třídě, aby vytvořila objekt, nebo již vytvořenému objektu
* zprávu zašleme / metodu vyvoláme např.

```
        adresát_zprávy.metoda(argumenty);     // Java, C#
  nebo
        [ adresát_zprávy metoda argumenty ]   // Objective C

* v některých distribuovaných systémech je zaslání zprávy implementováno
  přímo odesláním textové zprávy - příjemce ve zprávě najde požadovanou
  operaci a data, podle názvu operace určí metodu, vyvolá jí a předá jí data
* pokud objekty koexistují ve stejném programu, vyvolání metod je
  implementováno obdobně jako volání procedury v Pascalu nebo v C -
  komunikace je synchronní
* pokud jsou objekty implementovány pomocí paralelních procesů nebo vláken,
  může být operace asynchronní => volající může pokračovat zatímco se služba
  vykonává, popíšeme později

Dědičnost
.........

* angl. inheritance
* některé třídy budou mít společné vlastnosti
* např. třídy Muž a Žena budou mít mnoho společných vlastností
  (operace "jez" a "pij", atributy "jméno", "adresa", "datum narození")

* společné vlastnosti můžeme sdílet tak, že je vyjmeme a vložíme do
  samostatné třídy Osoba (používají se termíny generalizace, zobecnění)
 - ostatní třídy tyto společné vlastnosti (atributy a operace) mohou sdílet
    mechanismem dědění
 - ve třídách Muž a Žena můžeme popsat jenom nové vlastnosti
 - pokud je zapotřebí modifikovat společné chování, stačí změnit v definici
    Osoby

* někdy provádíme specializaci existující třídy - najdeme třídu která
  poskytuje operace a atributy potřebné pro novou třídu, nová třída je zdědí
  a přidá nové vlastnosti
 - např. specializací třídy Muž může být Programátor, který bude mít další
    atributy ("programovací jazyk") a operace ("programuj")

* zobecňováním a specializací vzniká hierarchie tříd
 - předkové, nadtřídy (ancestors, super-classes) - jsou obecnější, jednodušší
 - potomci, podtřídy (descendants, sub-classes) - specializovanější, složitější

* v UML se dědičnost znázorňujeme šipkou, která vede k rodičovské třídě
 - tři tečky (...) znamenají, že část modelu není zobrazena
 - znázorněny jsou dvě varianty pro kreslení šipek
```

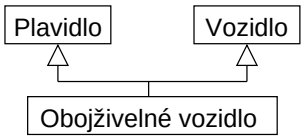


a) shared target style

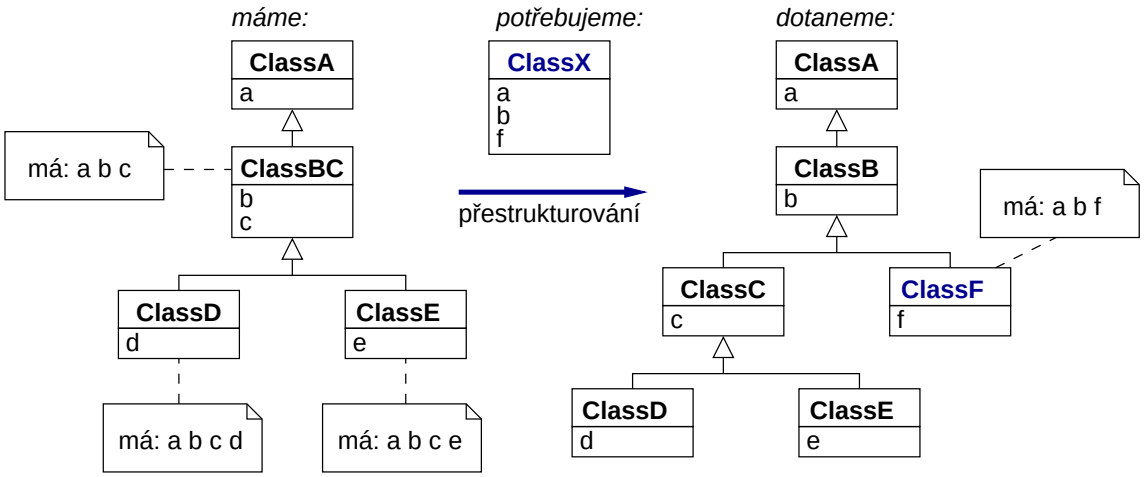


b) separate target style

```
* předkové vytvoření pouze pro účel dědění ostatními se nazývají
  abstraktní třídy - nevytvářejí se z nich instance
 - v UML se název abstraktní třídy zobrazuje kurzívou

* dědičnost může být jednoduchá a vícenásobná (multiple inheritance)
 - jednoduchá = každá třída dědí pouze z jedné rodičovské třídy
 - vícenásobná = třída může mít více než jednoho rodiče
  . vícenásobná dědičnost zvyšuje složitost hierarchie tříd, pravděpodobnost
    koncepčních chyb, je obtížně srozumitelná atd., proto jí mnoho OO jazyků
    nepodporuje (např. Java; toto omezení je možné v případě potřeby obejít
    vytvářením tzv. složených tříd)
  . v UML vícenásobnou dědičnost znázorňujeme:
```



* někdy musíme hierarchii tříd přestrukturovat, abychom získali vhodnou třídu ze které bychom mohli dědit
 - například v následujícím obrázku máme hierarchii tříd s vlastnostmi ”a”, ”abc”, ”abcd”, ”abce” a potřebujeme třídu s vlastnostmi ”abf”
 - ze současné hierarchie tříd není možné takovou třídu získat děděním, musíme přestrukturovat



* ve výše uvedeném obrázku také ukázka UML notace pro poznámky:
 - obdélník s přehnutým pravým horním rohem, připojen přerušovanou čarou

* použití dědičnosti v praxi:
 - sdílení kódu - při návrhu mohu sdílet kód mezi podobnými třídami
 - mohu si přizpůsobit existující třídu (např. z minulého projektu)
 - koncepční zjednodušení - zmenšení počtu nezávislých vlastností systému

* tradiční pravidla pro použití dědičnosti:
 - nová třída musí obsahovat všechny atributy původní třídy, lze přidávat nové atributy
 - nová třída musí rozumět stejným zprávám jako původní třída, lze přidávat nové operace
 - implementace metody může být v podtřídě změněna, neměl by se ale změnit její význam (pak už by nebylo čisté zobecnění/specializace)

* nesprávné použití dědičnosti - pokud mezi třídami není vztah zobecnění/specializace ale např. vztah kompozice (okno není specializace domu, ale dům může kromě jiného obsahovat okna)
 - jinými slovy: dědičnost bychom neměli používat pouze jako mechanismus pro ”vypůjčení si” kódu bez ohledu na cokoli, protože tím zaneseme do modelu konceptuální problémy a do kódu skryté předpoklady

Polymorfismus

* pokud bychom v klasickém procedurálním programovacím jazyku potřebovali vytisknout dva geometrické obrazce (reprezentované datovými strukturami typu ”čtverec” a ”obdélník”), nejspíše bychom pro to měli k dispozici samostatné procedury:

```
tiskni_čtverec(a);
tiskni_trojúhelník(b);
```

* v OO jazycích najdeme mechanismus polymorfismu = možnost zasílat stejnou zprávu různým objektům, které na ni odpoví každý svým způsobem

```
 - např. ve výše uvedeném případě může každý objekt mít metodu ”vytiskni_se”,
   vyvoláme:

  a.vytiskni_se;  // vytiskne se čtverec
  b.vytiskni_se;  // vytiskne se trojúhelník

 - příjemce zprávy může patřit do libovolné třídy implementující metodu
   ”vytiskni_se”, vysílající nemusí znát konkrétní třídu příjemce
 - pokud bychom měli např. seznam skládající se ze čtverců a trojúhelníků,
   mohli bychom napsat něco jako:

   foreach anObject in list  // projdi všechny objekty v seznamu ”list”
      anObject.vytiskni_se;  // vyvolej jejich metodu ”vytiskni_se”

 - například každý objekt může mít metodu ”ulož se do souboru” apod.
 - zvýšení flexibility

* většinou se polymorfismus omezuje (omezený polymorfismus)
 - například zprávu ”vykresli se na obrazovku” má smysl posílat pouze
   grafickým objektům (čára, obdélník...)


Paralelní objekty
.................

* objekty koncepčně žádají o provedení služby zasláním ”zprávy”, v tom není
  explicitní požadavek na sériové vykonávání
 - např. pokud bychom chtěli vytisknout soubor (a.printFile(f)), bylo by
   přirozené, kdyby volající nemusel čekat na dokončení tisku
 - obecný model dovoluje objektům běžet paralelně (např. na stejném počítači
   nebo jako distribuované objekty na různých strojích)
 - kdyby se ale jednalo např. o zavření dveří výtahu (a.closeDoors), mohlo by
   mít paralelní provedení nepříjemné důsledky
 - většinou předpokládáme návrat ve chvíli, kdy je to bezpečné
 - v UML se paralelismus operace označuje připojením vlastnosti
  { concurrency = concurrent }
```

Printer
printFile() {concurrent}

```
* většina jazyků implementuje vyvolání metody stejně jako volání fce, tj.
  volající objekt je pozastaven do dokončení operace
* některé jazyky (např. Java) umožňují vytvářet paralelně běžící objekty

* existují dva druhy implementace paralelně běžících objektů:
 - servery - objekt je implementován jako paralelně běžící proces, metody
   odpovídají operacím
  . metoda se spustí jako odpověď na příchozí požadavek, může běžet paralelně
    s metodami sdruženými s ostatními objekty
  . po dokončení se objekt pozastaví a čeká na další zprávu = požadavek
 - aktivní objekty - stav objektu se může měnit interní operací samotného
   objektu
  . proces představující objekt běží neustále nebo svůj stav upravuje v
    určitých intervalech, např. v RT systému zjišťuje stav okolí
  . např. v Javě - udržuje informaci o pozici letadla pomocí satelitního
    navigačního systému:

        class Transponder extends Thread {

                Position currentPosition;
                Coords c1, c2;
                Satellite sat1, sat2;
                Navigator navigator;

                public Position givePosition() {
                        return currentPosition;
```

```
                }

                public void run() { // zavolá se po vytvoření vláken
                        while (true) {
                                c1 = sat1.position();
                                c2 = sat2.position();
                                currentPosition = navigator.compute(c1, c2);
                        }
                }
        } //Transponder
```

Modelovací jazyk UML

* modelování = návrh aplikace před kódováním
* model hraje stejnou roli při vývoji SW jako studie a projektová dokumentace při stavbě domu
  - tj. slouží pro vizualizaci návrhu, kontrolu a dokumentaci před zahájením realizace (kódování)
* model abstrahuje základní detaily skutečnosti nebo vytvářeného systému
* standardní notace pro OO modelování je UML

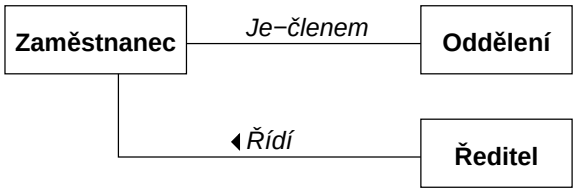
* UML definuje 12 typů diagramů rozdělených do 3 kategorií:
  - strukturální diagramy (diagram tříd, diagram objektů, diagram komponent a diagram nasazení - všechny popisují statickou strukturu systému)
  - diagramy chování (diagram případů použití - některé metodiky ho používají pro sběr požadavků, diagram spolupráce a sekvenční diagram - pro popis komunikace, stavový diagram a diagram činností)
  - diagramy pro správu a strukturování modelů (balíčky, podsystémy a modely)

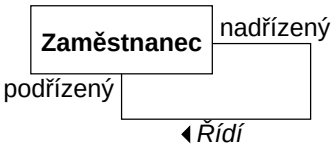
Diagram tříd a diagram objektů

* diagram tříd ukazuje třídy a vztahy mezi nimi (zobecnění, asociace, agregace)
  - vztahy zobecnění (dědičnosti) už byly popsány, ukážeme si další typy vztahů

Asociace

* asociace říká, že objekty které jsou instancemi jedné třídy mohou mít vztah s objekty jiné nebo stejné třídy
* např. objekty třídy Zaměstnanec budou mít asociaci k objektům třídy Oddělení
* v UML se znázorňují plnou čarou mezi třídami, u čáry volitelně název vztahu
  - u názvu volitelně malý černý trojúhelníček ukazující kterým směrem se má název vztahu číst
  - asociace může být i rekurzivní

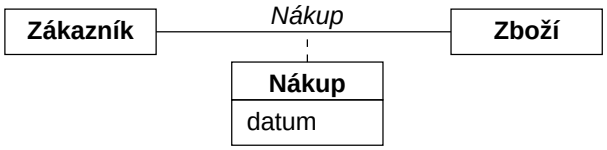




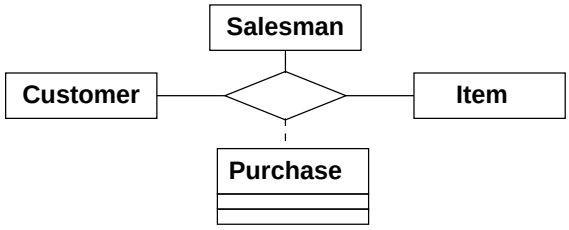
  - konce asociace mohou být volitelně popsány rolemi ve vztahu
    . role popisující vzdálené konce asociací stejné třídy mají být jedinečné
    . u rekurzivních vztahů (nadřízený řídí podřízeného) by role měla být uvedena vždy
    . pojmenování rolí užitečné pokud je více než jedna asociace mezi stejným párem tříd

* pokud má asociace vlastnosti jako atributy, operace a další asociace, můžeme pro ní vytvořit tzv. asociační třídu (analogie ”asociativního indikátoru typu” v ERA diagramech)
  - příklad: zákazník nakoupil zboží
    . asociace ”nakoupil” sdružuje zákazníky a položky zboží

. pokud bychom chtěli uchovat informaci o datu nákupu atd., nepatří ani
k zákazníkovi, ani ke zboží



* všechny dosavadní asociace byly binární, tj. do vztahu vstupovaly dvě strany
 - binární asociace jsou zdaleka nejčastější, občas se může vyskytnout
   ternární atd.
 - n-ární asociaci můžeme znázornit pomocí prázdného kosočtverce
 - následující obrázek ukazuje ternární asociaci která je zároveň asociační
   třídou:



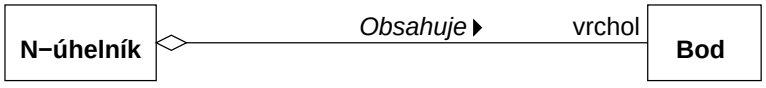
* asociace je velmi obecný typ vztahu, v UML se často používá pro označení že
  některý atribut je asociovaný objekt nebo že implementace některé metody
  závisí na asociovaném objektu
 - v počátečních fázích návrhu je vhodné nezahrnovat asociace, které
   bezprostředně nepotřebujeme
 - v konečných fázích návrhu je dobré konkretizovat (např. vyjádřit vztah
   agregace a kompozice)

Agregace
.....

* jednou z nejčastějších binárních asociací je agregace
* objekt je vytvořen z dalších objektů = je agregátem množiny objektů
 - např. objekt Stádo je agregátem Ovcí, Les je agregátem Stromů, Rodina
   bude agregátem objektu typu Muž, objektu typu Žena a množiny objektů Dítě
 - nebo Předmět se může skládat z Přednášek, Cvičení, Zápočtové_úlohy,
   Zkoušky atd.

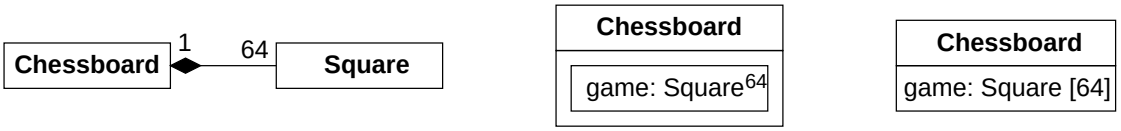
* agregace je více než pouze součet svých částí, vzniká něco nového (agregát)
 - agregát může vystupovat v některých operacích jako samostatná jednotka
 - části mohou existovat samostatně, mohou být součástí dalších agregací

* v UML se agregace znázorňuje prázdným kosočtvercem na straně agregátu:



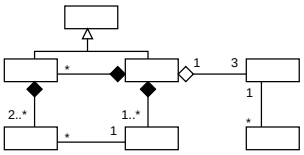
Kompozice
.....

* kompozice je silná asociace - součást náleží právě jednomu složenému objektu
 - součást nemůže existovat samostatně (políčko nemůže existovat bez šachovnice)
 - při zániku celku tedy zaniknou i jeho části
 - v UML se kompozice znázorňuje plným kosočtvercem nebo grafickým vnořením:



- kompozice je jediný vztah, který smíme modelovat ”pohřbením” objektů jako atributů jiného objektu (objekt silně patří jediné třídě)
  . kromě tohoto případu se reference na objekty nemají vyskytovat jako atributy objektů, ale mají být vždy modelovány jako příslušné asociace

* v jednom diagramu mohou vystupovat různé typy vztahů:



* v příkladu uvedena také násobnost (kardinalita) vztahu = počet elementů, které mohou vstoupit do vztahu; uvádí se intervalem nebo výčtem
- intervalem: 0..1 = žádný nebo jeden, 1..* jeden a více, * je totéž co 0..*
- výčtem: 1..3,7,10
- pokud násobnost není uvedena, nelze o ní nic předpokládat (kromě kompozice, kde plný kosočtverec zároveň značí násobnost 1)

Rozhraní a jeho realizace

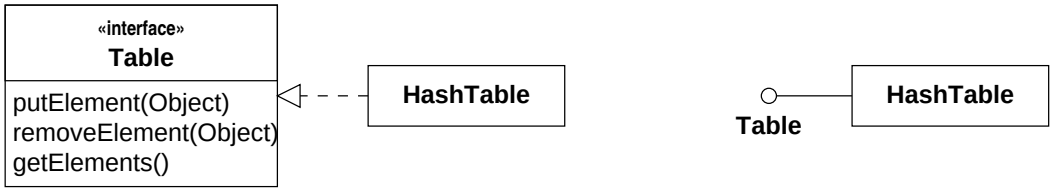
* specifikace atributů a operací ve třídě:

Class Name
attribute attribute: type attribute: type [multiplicity] attribute: type = init_value ...
operation() operation(arg_list) operation(arg_list): return_type ...

ExampleClass
size boundingBox: Rectangle colors: Color [3] userName: String = "root" ...
hide() display(win: Window) setColor(c: Color): Boolean ...

- specifikace atributů:
  <atribut> : <typ> [<násobnost>] = <počáteční hodnota>
- specifikace operací:
  <operace> (<seznam parametrů>) : <typ návr.hodnoty>
  kde parametr má tvar: <in|out|inout> <parametr> : <typ> = <default.hodnota>
- defaultní je ”in”; volba ”in”, ”out” nebo ”inout” zvýrazněně
- typy jsou závislé na implementačním jazyce
- před atributem nebo operací může být uvedena <viditelnost>
  . ”+” public = viditelné (přístupné) pro všechny třídy
  . ”#” protected = viditelné pouze uvnitř třídy a pro potomky
  . ”-” private = viditelné pouze uvnitř třídy, není viditelné ani pro potomky
  . ”~” package = viditelné pouze uvnitř balíčku

* rozhraní (interface) je specifikace navenek viditelných operací třídy, komponenty apod.
- specifikuje pouze rozhraní pro operace bez jejich implementace
- nemůže mít atributy, stav nebo asociace
- dva způsoby zakreslení:
  . podobně jako třída, nad jménem rozhraní klíčové slovo <<interface>>
  . část s atributy můžeme vynechat (je vždy prázdná)
  . pokud třída rozhraní realizuje (tj. pokud implementuje všechny operace rozhraní), znázorňujeme to přerušovanou šipkou s trojúhelníkovou hlavou, která směřuje od třídy k rozhraní
- rozhraní se může kreslit také jako kroužek spojený plnou čarou s třídou která ho implementuje, pod kroužkem název rozhraní



* rozhraní mají v mnoha OO jazycích přímou realizaci (v Javě klíčová slova
  interface a implements)
 - stejné rozhraní mohou implementovat jinak nepříbuzné třídy

Stereotypy
....

* výše uvedené klíčové slovo <<interface>> je příklad tzv. stereotypu
 - stereotyp = rozšíření UML o nové prvky, které mají stejnou podobu jako
   prvky existující, ale odlišný účel
 - klíčové slovo stereotypu se píše nad jméno prvku, např. nad jméno třídy
   do francouzských uvozovek (= znaky ”guillemotleft” a ”guillemotright”),
   pokud tyto nejsou k dispozici pak mezi dvojice znaků << a >>

❇