

REPREZENTACE INFORMACE V PAMĚTI - OBJEKTOVĚ ORIENTO VANÝ NÁVRH



Kurz: **Datové struktury a algoritmy**

Lektor: Doc. Ing. Radim Burget, Ph.D.

Autor: Doc. Ing. Radim Burget, Ph.D.



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098
Evropského sociálního fondu a státním rozpočtem České republiky.

Cíl přednášky

- Jak reprezentovat informaci v počítači = navrhovat software
- Co je to objektově orientovaný návrh (OON)?
 - Příklad návrhu směrovače
- Jaké fáze má návrh software? (Téměř identické s návrhem sítí, ITIL)
- Jak lze graficky modelovat znalost (= datový model), jazyk UML
 - Specifikaci požadavků ... UML diagram případů užití
 - Návrh software ... UML diagram tříd
- Existují některé zažité a časem ověřené návrhy (Návrhové vzory)
- Příklady řešení problémů pomocí OON
 - Teorii si předvedeme na příkladu návrhu směrovače

V čem je problém?

- Investor přišel s úžasnou myšlenkou pro nový produkt / službu
- Zbývá už „jen“ realizace...
- V čem je problém ?

Co zákazník objednal



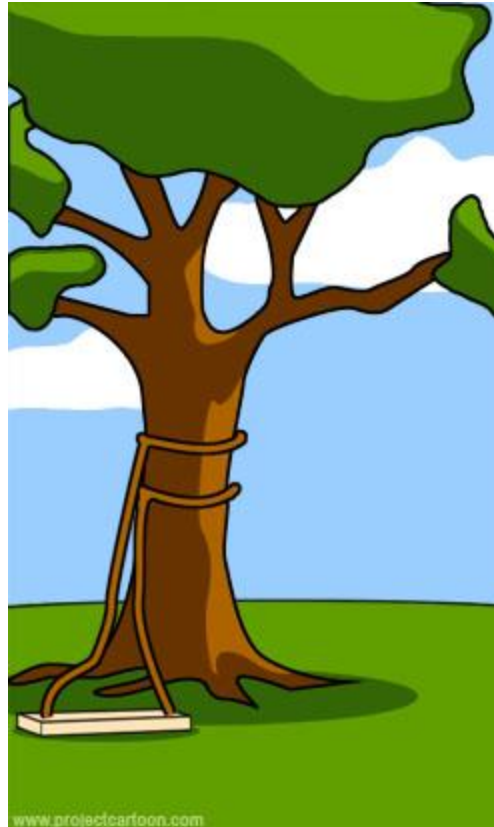
Jak zadání porozuměl obchodník



Jak navrhl řešení analytik



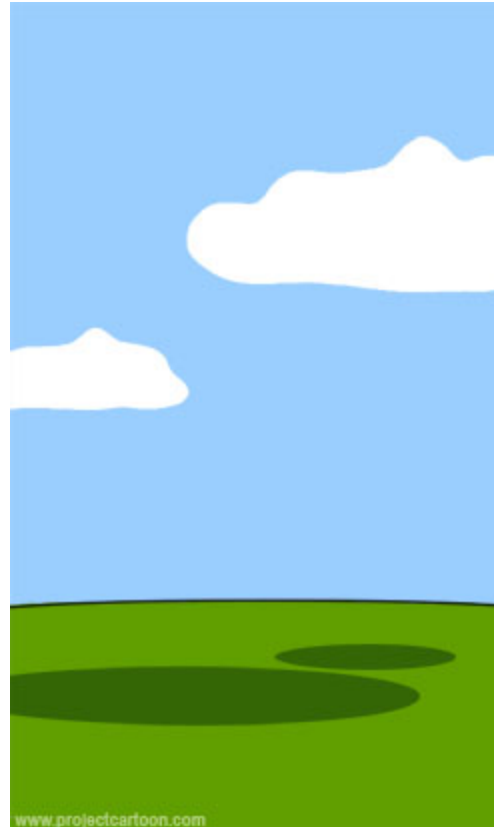
Co dodali programátoři



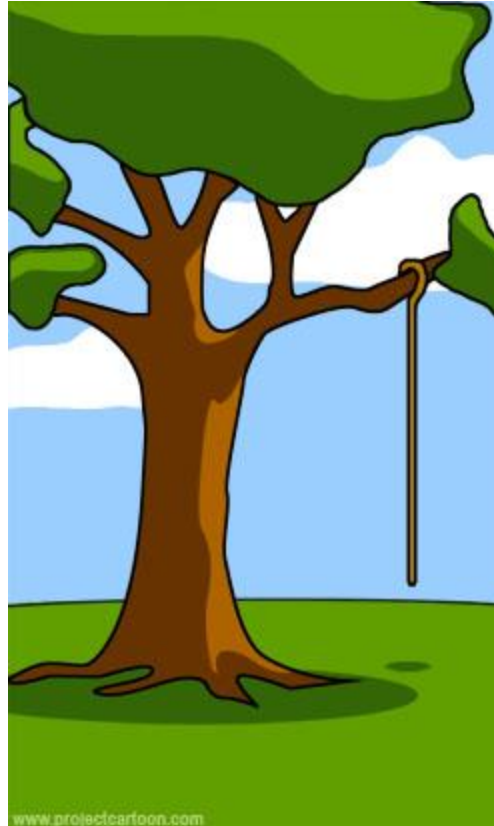
Jak bylo řešení zákazníkovi prezentováno



Co bylo zdokumentováno



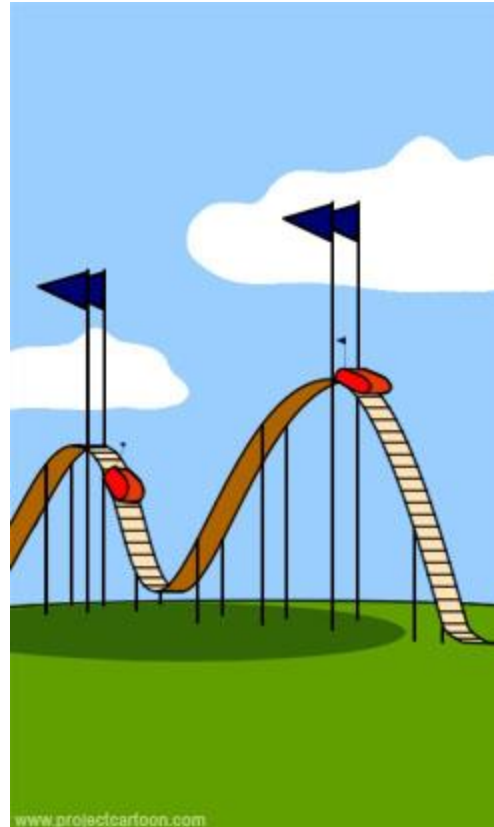
Co bylo zákazníkovi dodáno



Kdy byl projekt dodán



Co bylo zákazníkovi vyfakturováno



Co zákazník opravdu potřeboval



Modelování informace = Návrh software

- Návrh software
 - modelování informace
 - uživatelského rozhraní
- **Není triviální záležitost a vyžaduje**
 - Systematický postup
 - Porozumění dané problematice
 - Často nelze naplánovat detailně od A do Z
 - Požadavky se mohou měnit v čase
 - Komunikace v rámci vývojového týmu, se zadavatelem

Paradigmata pro vývoj software

- Procedurální (C, Pascal, ...)
- Funkcionální (Lisp, Haskell, ...)
- Objektově orientované programování (C++, JAVA, SmallTalk, ...)
- Mnoho dalších
 - Založené na teorii automatů
 - Založené na agentech
 - ...

Objektově orientovaný návrh

- Jeden ze způsobů jak reprezentovat informaci
- Metodika návrhu software
- Vychází z principů reálného světa, jsou blízké jak je chápe člověk
- Nezávisí na prog. jazyce (musí být OO)
- Umožňuje zapouzdřit složité myšlenky a poskytnout k nim rozhraní
 - Ovládání televize (víte na které frekvenci je který program?)
 - Řízení auta (víte, veškeré podrobnosti ohledně řízení auta?)
 - Umělá inteligence (neuronové sítě, SVM, k nejbližších sousedů, ...)
- Výhoda – nižší náklady na údržbu aplikací
- Není pravda, že OON přináší horší výkonnost – většina her je již řadu let vyvíjena pomocí OON

Objektově orientovaný návrh

- Bližší věcem z reálného světa jak je známe
- Tj. není to nový způsob myšlení!
- **Snazší** pro porozumění
- Nižší náklady pro údržbu kódu (méně chyb)
- Není méně výkonný
 - Používá jej i herní průmysl,
 - kriticky náročné části mohou být implementovány v Assembleru a vloženy do OON (herní průmysl)
- Již po mnoho let trendem a jen ve výjimečných případech není vhodný



Objektově orientovaný návrh

- Kdy není vhodný OO přístup:
 - **Na cílové platformě neexistuje překladač OO jazyka**
 - Potřebuji to v konkrétním jazyce, který nepodporuje OO programování
 - Existuje mnoho stávajícího kódu nekompatibilního s OON a vzhledem k životnosti projektu se již nevyplatí jej přepisovat
 - Vývojáři neznají OOP?
 - => z dlouhodobého hlediska horší správa projektu
 - => vyšší náklady
 - => nižší konkurenceschopnost
 - X naučit se myslet objektově není náročné ani složité

Objektově orientovaný návrh

- Dobře napsaný kód v procedurálním jazyce se blíží OON
 - I v jazycích s podporou OO programování lze psát špatně, dá to ale více práce
- Vlastnosti
 - Více typovost
 - Dědičnost vlastností
 - Polymorfismus
 - Asociativnost
 - ...

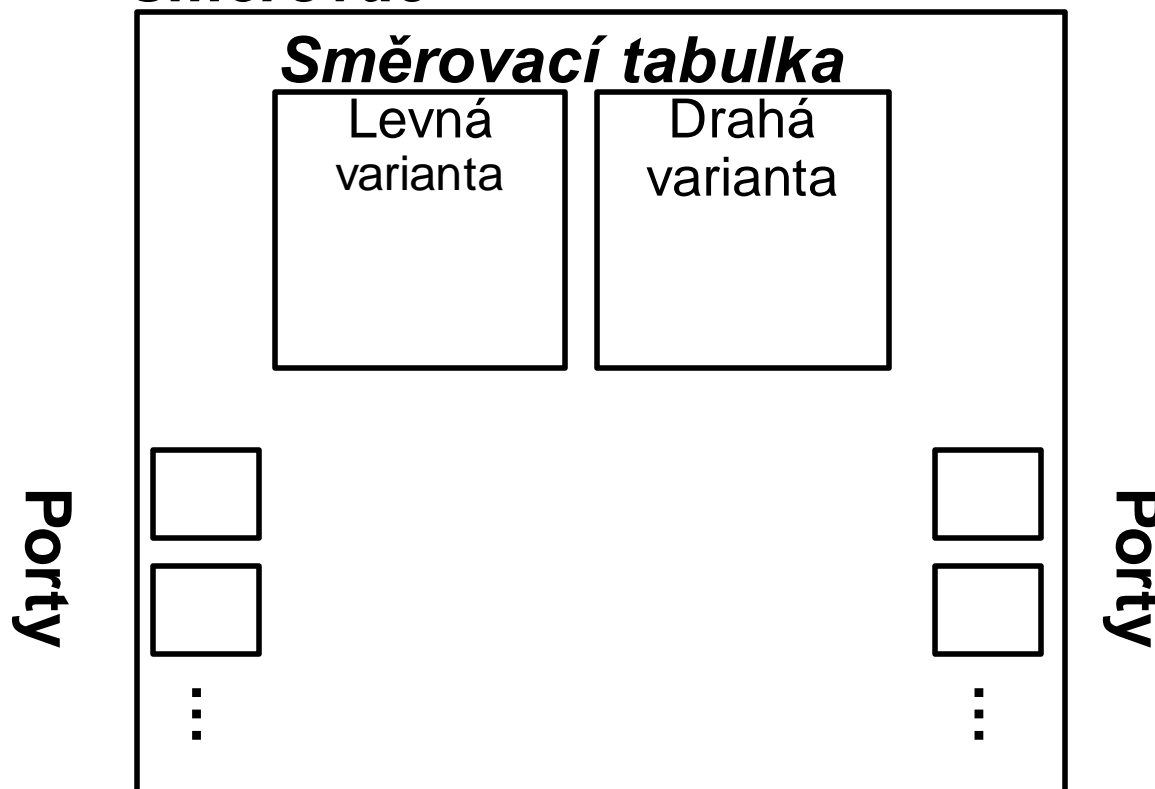
OON – Příklad – SW směrovač

- Příklad: SW **směrovač** se bude skládat ze:
 - **Směrovací tabulky**, budou se nabízen ve dvou variantách,
 - levné a pomalé a
 - dražší a rychlé (např. HW realizace)
 - **Portů:**
 - FastEthernet
 - GigabitEthernet
 - do budoucna se předpokládá, že budou přibývat nové
 - Jeho funkcí je poslouchat na portech příchozí pakety a odeslat je na patřičný odchozí port dle směrovací tabulky



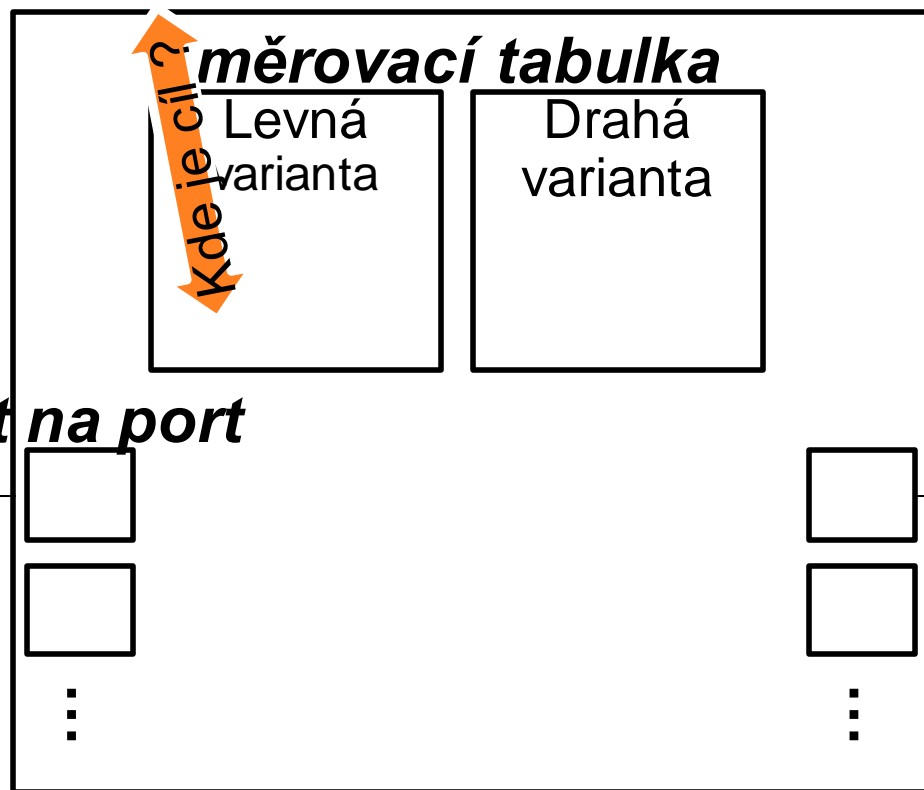
OON – Příklad – Popis možného fungování

Směrovač



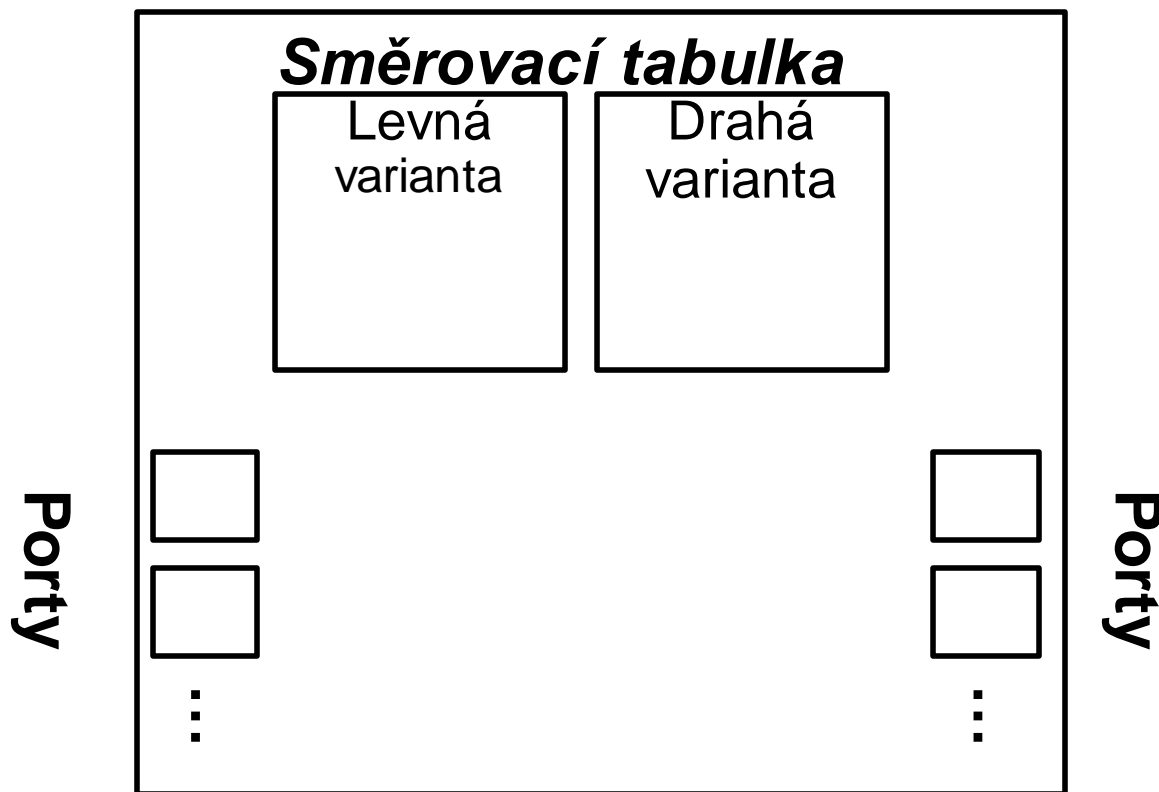
OON – Příklad – SW směrovač - Funkce

Toto jsem obdržel, udělej s tím něco
Směrovač



OON – Příklad – Popis možného fungování

Směrovač



- Co mne bude zajímat:
 - Jaké operace budou provádět jednotlivé části (Funkce)?
 - Mají některé části společné vlastnosti (Dědičnost)?
- Co s čím souvisí aneb, co je součástí čeho (Asociace)?
 - Směrovač
 - Směr. tabulka
 - Porty vstupní a současně výstupní

OON – Příklad – SW směrovač - Asociace

- Směrovač obsahuje
- 1x směrovací tabulku (levnou či drahou)
- 0.. n portů

Násobnost asociace

- Porty a směrovací tabulka spolu nemusí mít vazbu – komunikace bude přes směrovač – tj.
 - Port oznámí, že obdržel paket
 - Směrovač požádá směrovací tabulku o cílový výstupní port
 - Směrovač odešle z výstupního portu paket

OON – Příklad – SW směrovač - Funkce

- Port
 - Čti příchozí pakety
 - Informuj směrovač, že byl obdržén paket
- Tabulka
 - Vyhledej výstupní port
- Směrovač
 - Spust'
 - Vypni

OON – Příklad – SW směrovač - Dědičnost

- Jsou tam některé části, které mají obdobné chování?
- Směrovací tabulka
 - Levná
 - Drahá
- Porty
 - FastEthernet
 - GigabitEthernet

OON – Příklad – SW směrovač - Dědičnost

- (směrovací tabulka & porty)
 - Lišit se bude jejich chování (algoritmus SW či HW realizace)
 - Tak jak je budeme používat bude zcela identické
 - Asi je vhodné vytvořit pro části se společným chováním nějaké rozhraní

OON – Příklad – SW směrovač - Použití

- **Všimněte si:**

- Návrh se nezabývá samotnou implementací, jen vazbami mezi jednotlivými bloky
- Co když budeme potřebovat 2 takové směrovače? Musíme provést opět předchozí návrh ?

- **NEMUSÍME**

- předešlé slouží jako nějaký popis (dokumentace)
- Z této „dokumentace“ lze potom vytvářet konkrétní objekty

OON – Příklad – SW směrovač

- Pokud jste schopni určit:
 - Jaké entity zde vystupují,
 - Co s čím má vztah (Asociace),
 - Jaká je násobnost asociací (1, konst. ,0..N anebo 1..N)
 - Jaké chování je společné (dědičnost)
- Jste schopni provést OON
- Pro dobrý OON návrh je třeba mít představu o **náročnosti následné implementace**, tzv. **návrhových vzorech** a **abstraktních datových typech**.



Problém

- Tento popis by měl splňovat tyto požadavky:
 - Jednoznačnost
 - Jednoduchost
 - Srozumitelnost
 - Snadná čitelnost pro člověka
 - Standard (a známý po celém světě)
- Odpověď:
Unified Modeling Language (UML)



Návrh: Proč standard?



Brno



Praha



Jak zajistit, aby si týmy rozuměly a mohly spolu komunikovat efektivněji a přehledněji než jen pomocí zdrojového kódu?

Proces návrhu systému

Fáze	Akce	Výstup
Inicializace	Nová obchodní příležitost	Obchodní dokumenty
Požadavky	Rozhovory se zúčastněnými stranami, zkoumání prostředí systému	Strukturalizov. Dokumentace
Analýza a specifikace	Analýza technických aspektů systému, sestavení konceptu systému	Logický systémový model
Návrh	Definice architektury, komponent, datových typů, algoritmů	Implementační model
Implementace	Program, překlad, jednotkové-testování, integrace, dokumentace	Testovatelný systém
Testování & Integrace	Integrace všech komponent, ověřování, validace, instalace, poradenství	Výsledky testů, Funkční systém
Údržba	Opravy chyb, úpravy, přizpůsobení	Verze systému

Proces návrhu systému

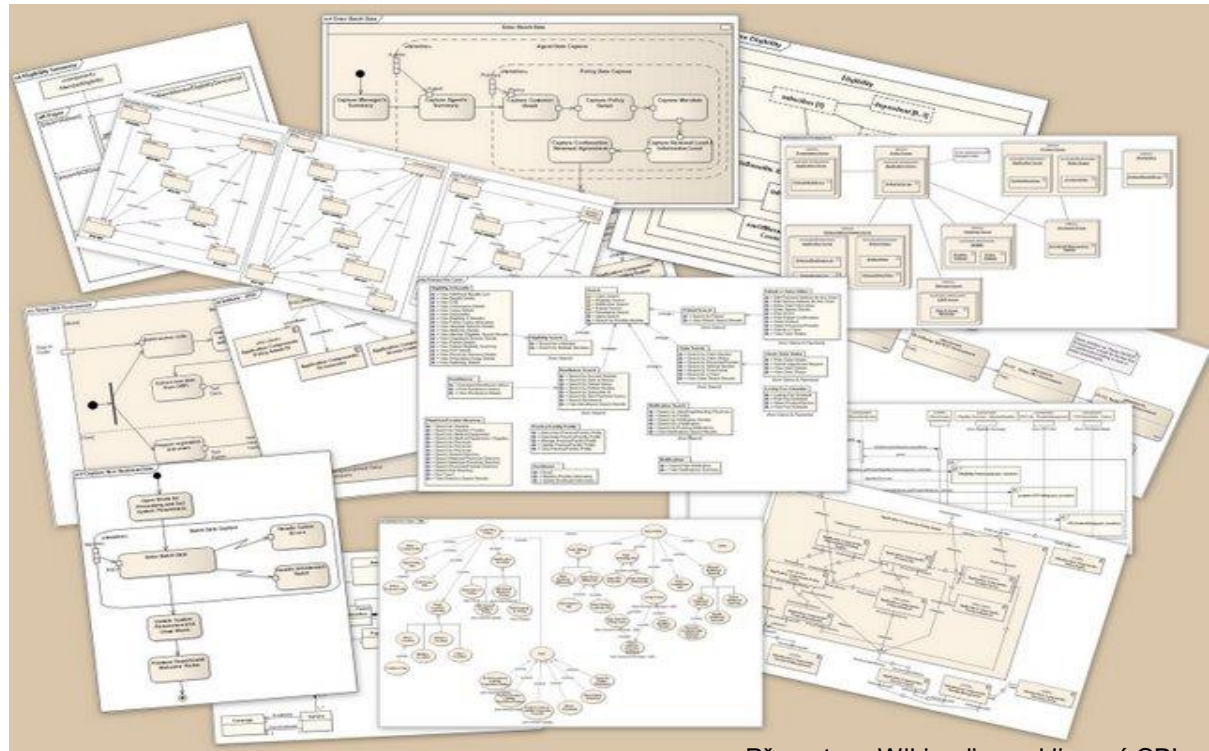
Fáze	Akce	Výstup
Inicializace	Nová obchodní příležitost	Obchodní dokumenty
Požadavky	Rozhovory se zúčastněnými stranami, zkoumání prostředí systému	Strukturalizov. Dokumentace
Analýza a specifikace	Analýza technických aspektů systému, sestava konceptu systému	UML případů užití model
Návrh	Definice architektury, komponent, datových typů, algoritmů	Implementační model
Implementace	Program, překlad, jednotkové-testování, integrace, dokumentace	UML Diagram tříd system
Testování & Integrace	Integrace všech komponent, ověřování, validace, instalace, poradenství	Výsledky testů, Funkční systém
Údržba	Opravy chyb, úpravy, přizpůsobení	Verze systému

Obvyklý postup návrhu

- Jak jednoznačně předat informaci od zákazníka vývojovému týmu?
- Jak komunikovat uvnitř vývojového týmu
- Jak komunikovat mezi vývojovými týmy (např. Praha, Brno, USA)

Jazyk UML

- Grafický jazyk pro popis programových systémů:
 - vizualizace,
 - specifikace,
 - návrh ,
 - a dokumentace.

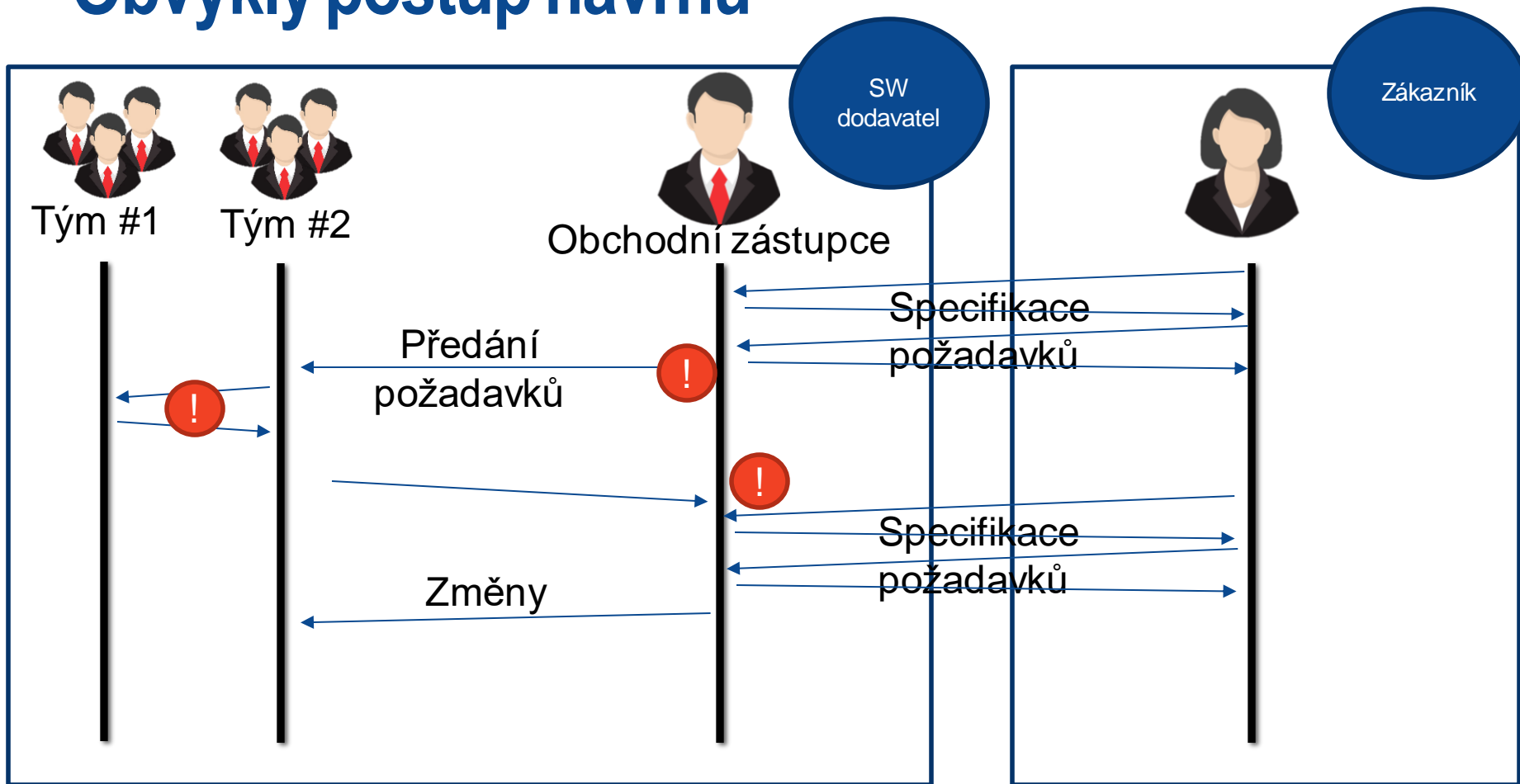


Převzato z Wikipedia pod licencí CPL

Jazyk UML

- Hlavní motivací je jednoduchost
- Nejčastěji používané diagramy:
 - **Strukturální** (popisují strukturu programu)
 - **Diagram tříd**
 - Diagram komponent **x**
 - Diagram nasazení **x**
 - **Diagram případů užití**
 - **Behaviorální** (popisují chování po spuštění programu)
 - Diagram aktivit **x**
 - Diagram sekvencí **x**
 - Diagram stavů **x**

Obvyklý postup návrhu



Proces návrhu software

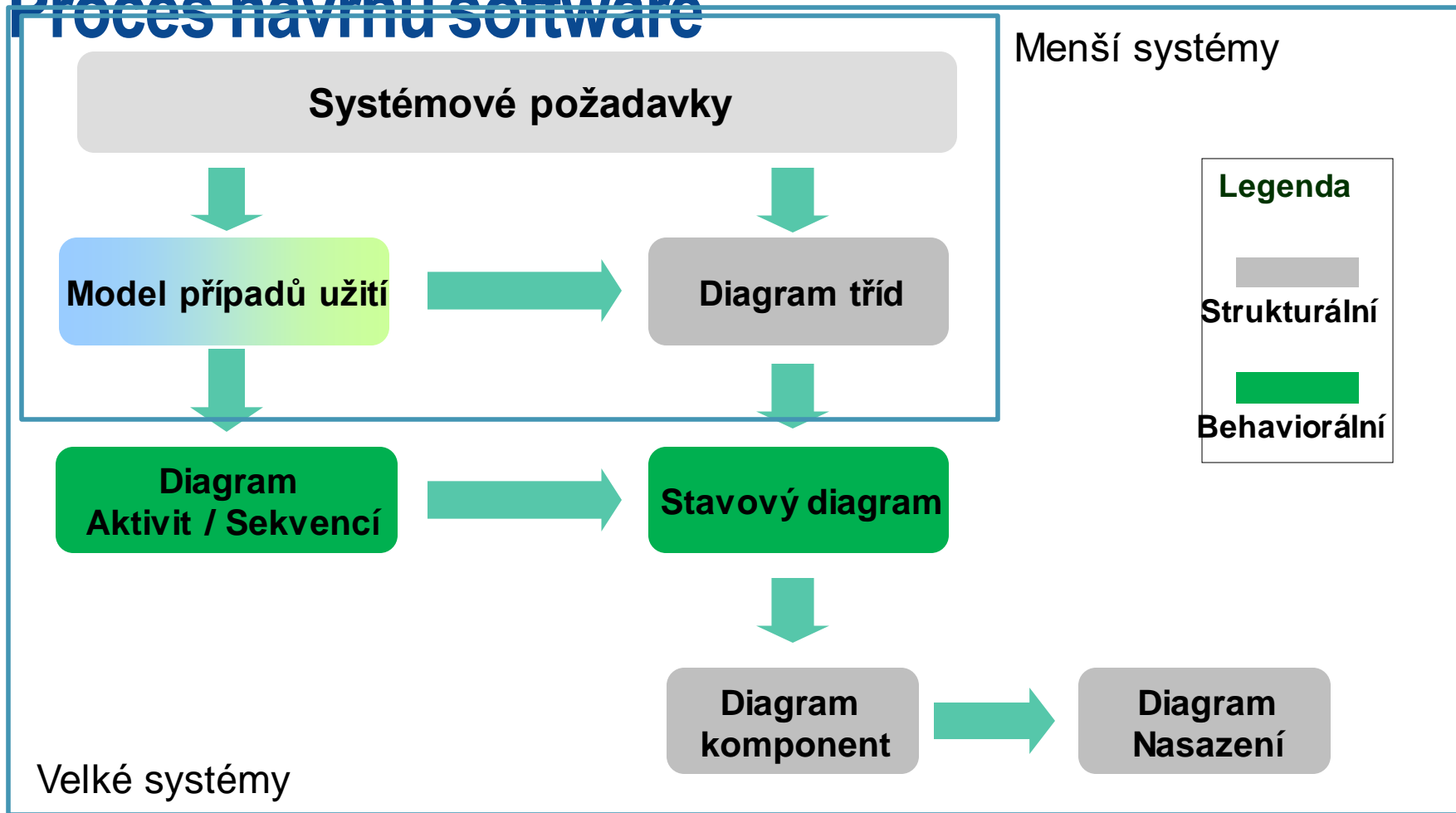


Diagram případů užití

Diagram případů užití: Elementy notace (1/2)

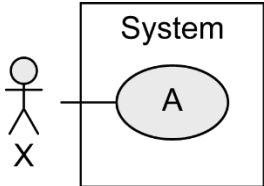


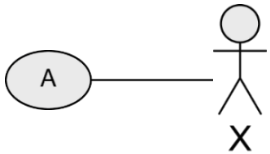
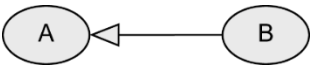
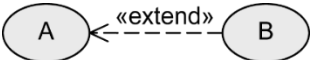
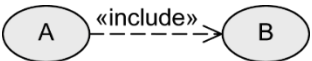
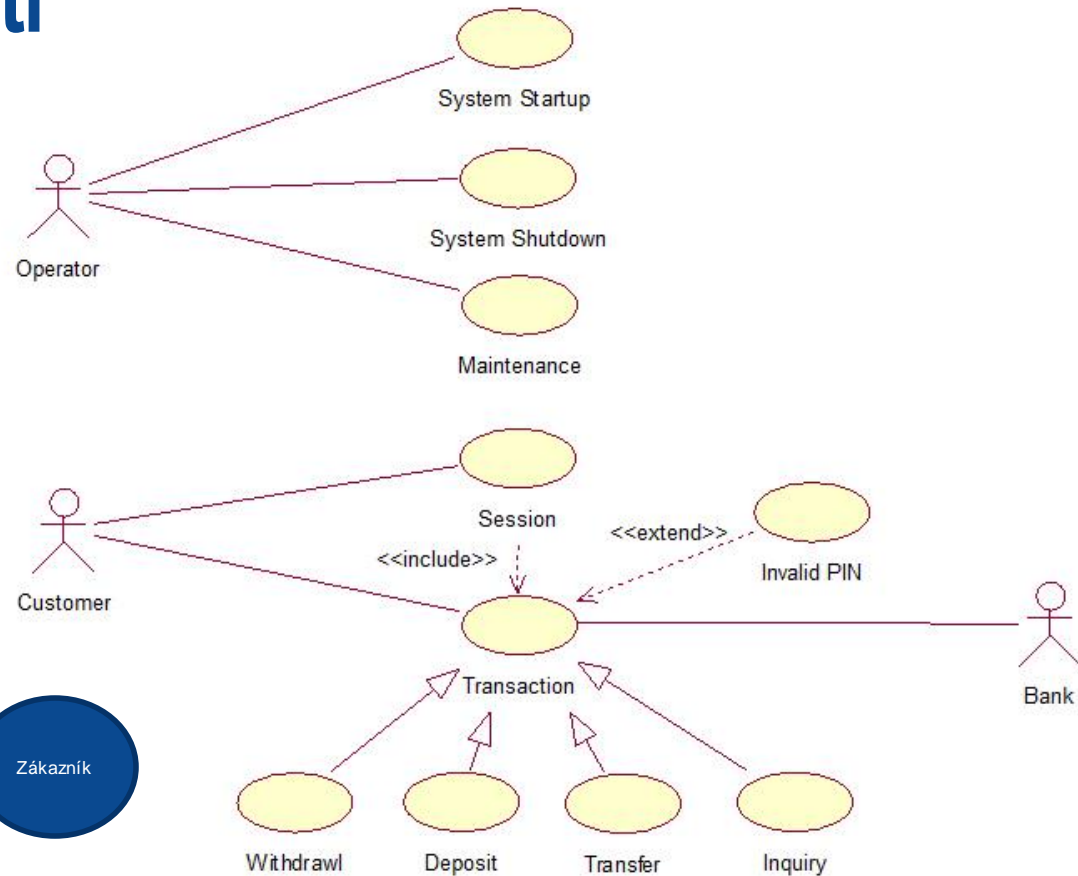
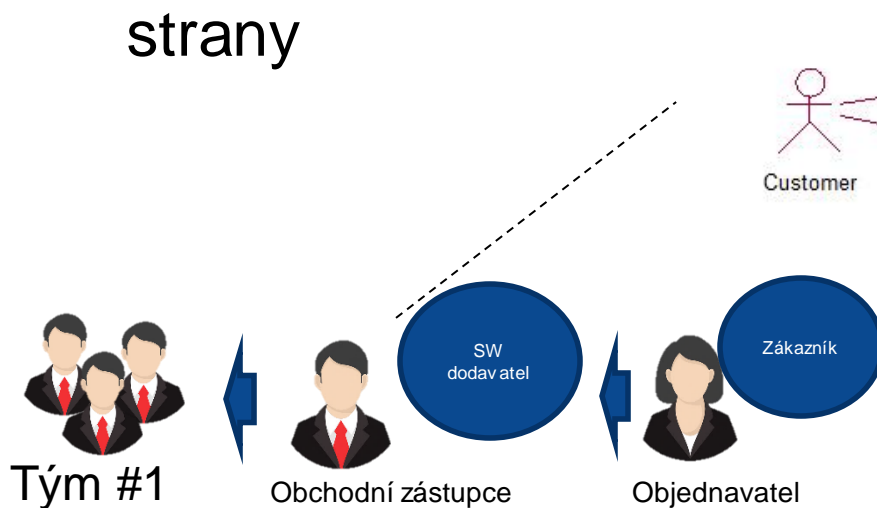
Název	Notace	Popis
System		Hranice mezi systémy a uživateli systémů
Případ užití		Jednotka funkčnosti systému
Herec (Actor)		Role uživatele systému

Diagram případů užití: Elementy notace (2/2)

Název	Notace	Popis
Asociace		Vztah mezi případy použití a herci
Generalizace		Dědičný vztah mezi herci nebo případy použití
Vztah rozšiřuje		B rozšiřuje A: volitelně B zahrnuje A
Vztah zahrnuje		A zahrnuje B (povinně)

Příklad případu užití

- Jak by mohla vypadat specifikace požadavků ze strany



Příklad: Specifikace návrhu

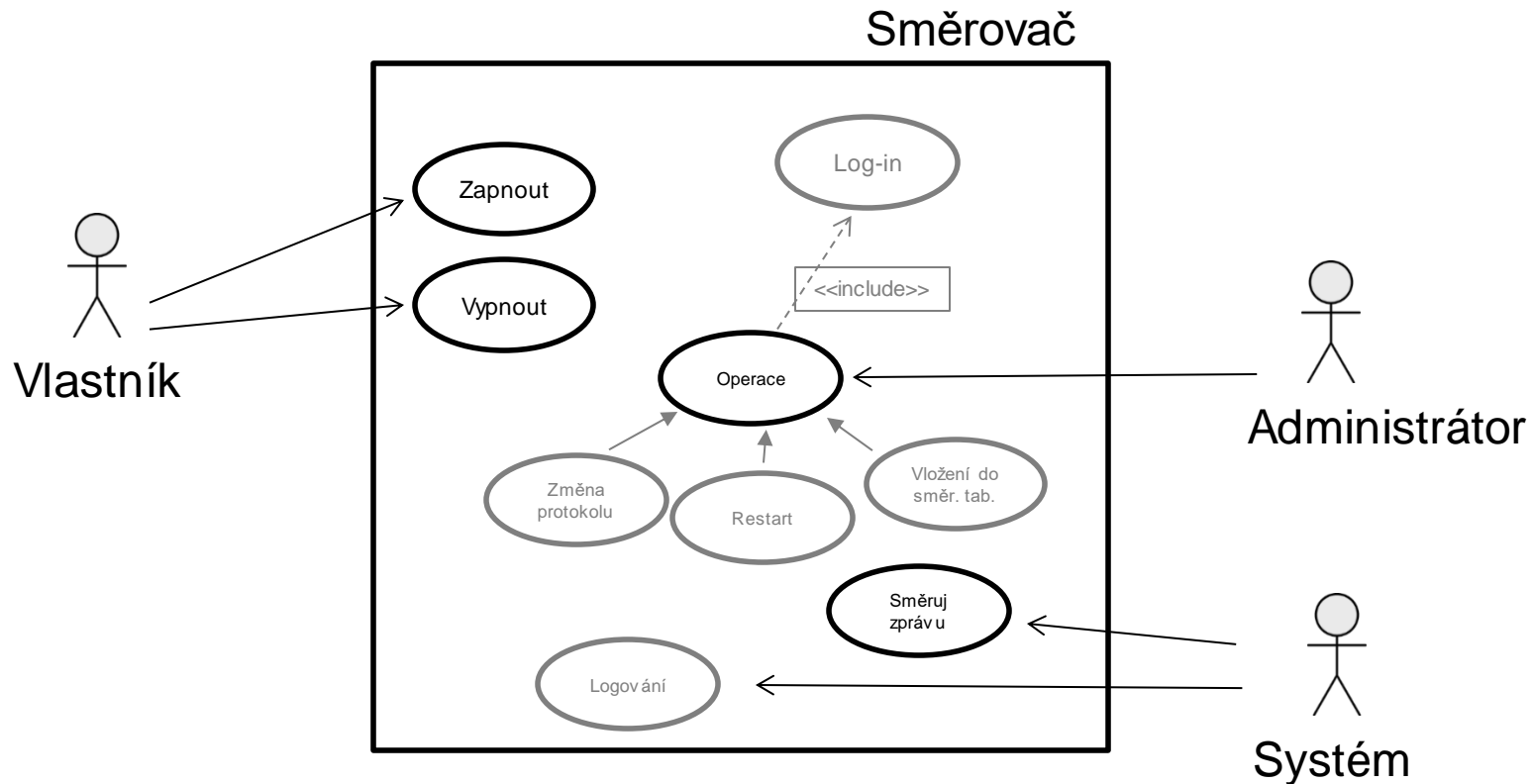
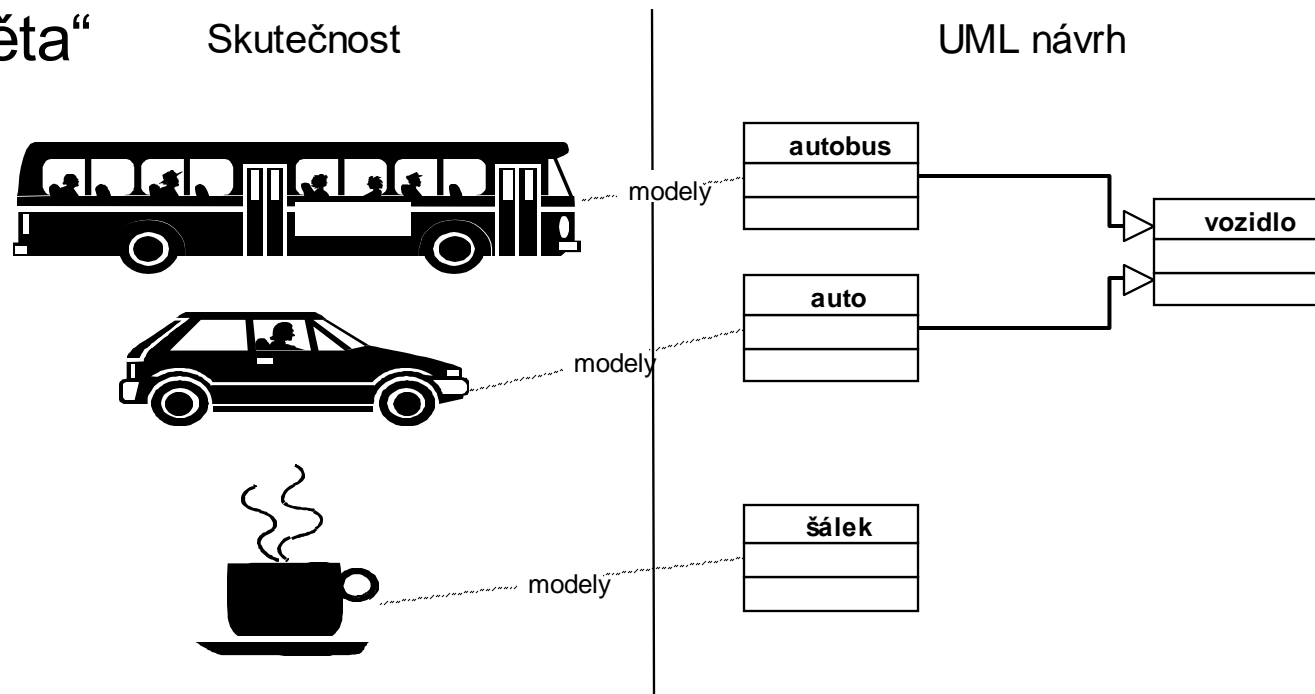


Diagram tříd

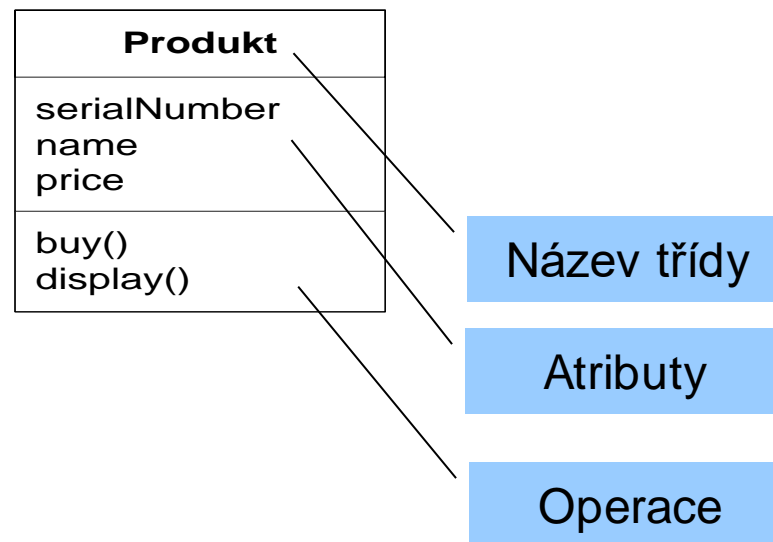
Objektově orientovaný návrh

- Objekty jsou abstrakcí skutečných objektů z „reálného světa“



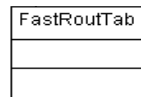
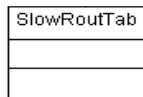
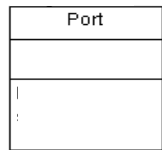
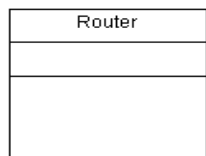
Třídy

- Třída je jakási předloha (popis, dokumentace) pro skutečné objekty (instance), které jsou v paměti
- Z každé třídy lze vytvářet libovolný počet objektů
- Zpravidla v angličtině
- Raději bez diakritiky



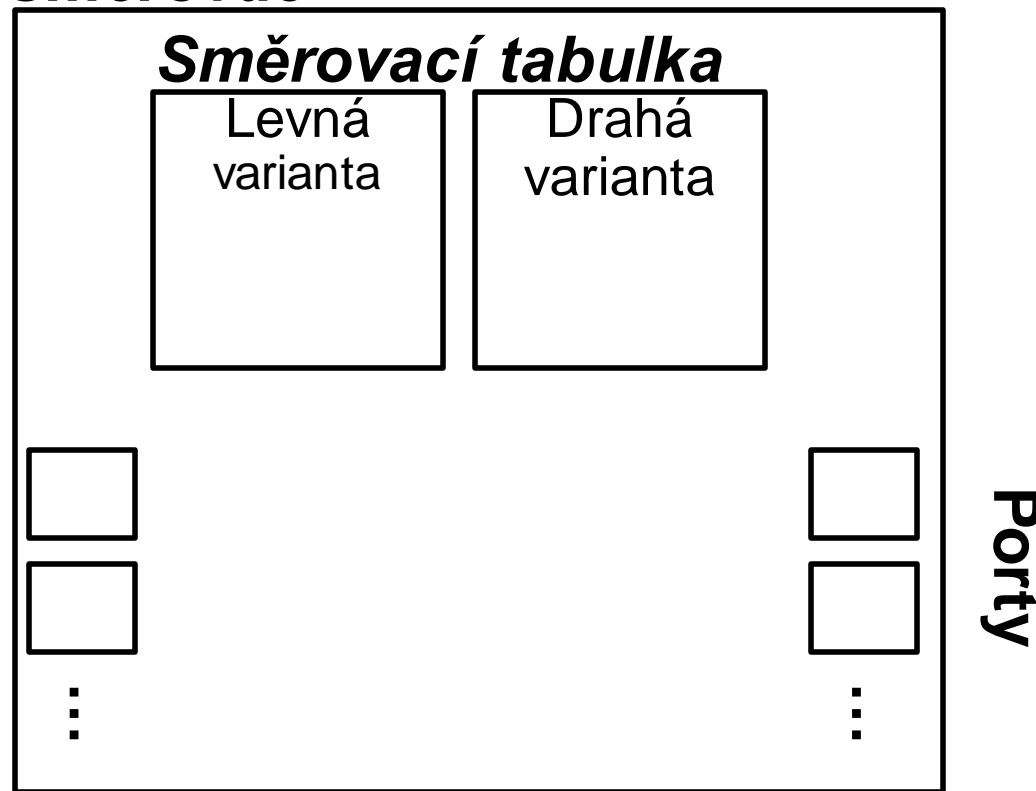
Příklad: Směrovač

- Navrhněte třídy pro příklad směrovače:
- Jaké entity lze nalézt?



Porty

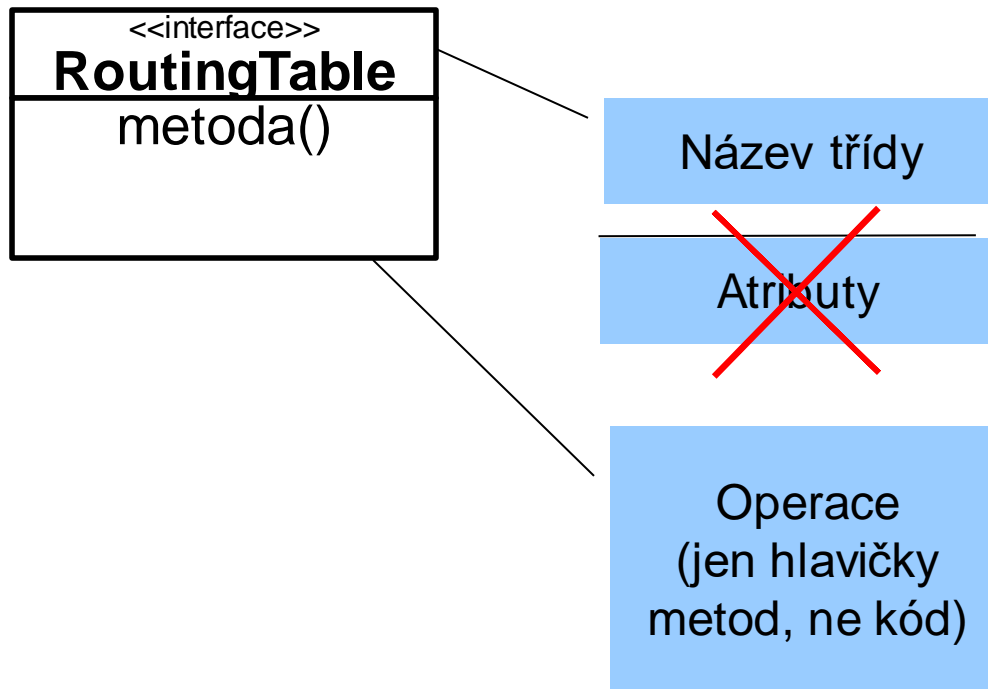
Směrovač



Rozhraní

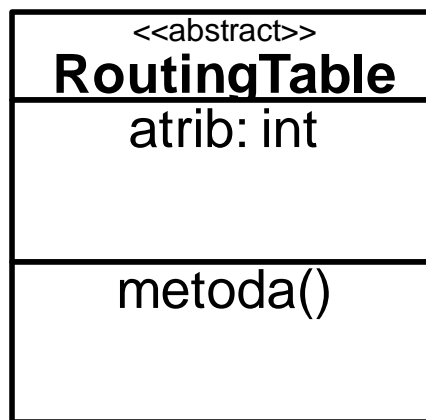
- Třída je jakási předloha (popis, dokumentace) pro skutečné objekty (instance), které jsou v paměti

- Zpravidla v angličtině
- Bez diakritiky



Abstraktní třídy

- Kompromis mezi rozhraním a třídou
- Nelze vytvořit objekt (ale lze z ní dědit u jiných tříd)
- Mohou existovat tzv. abstraktní metody – jako u rozhraní – pouze hlavičky metod, bez samotného kódu
- Jinak vše jako třída



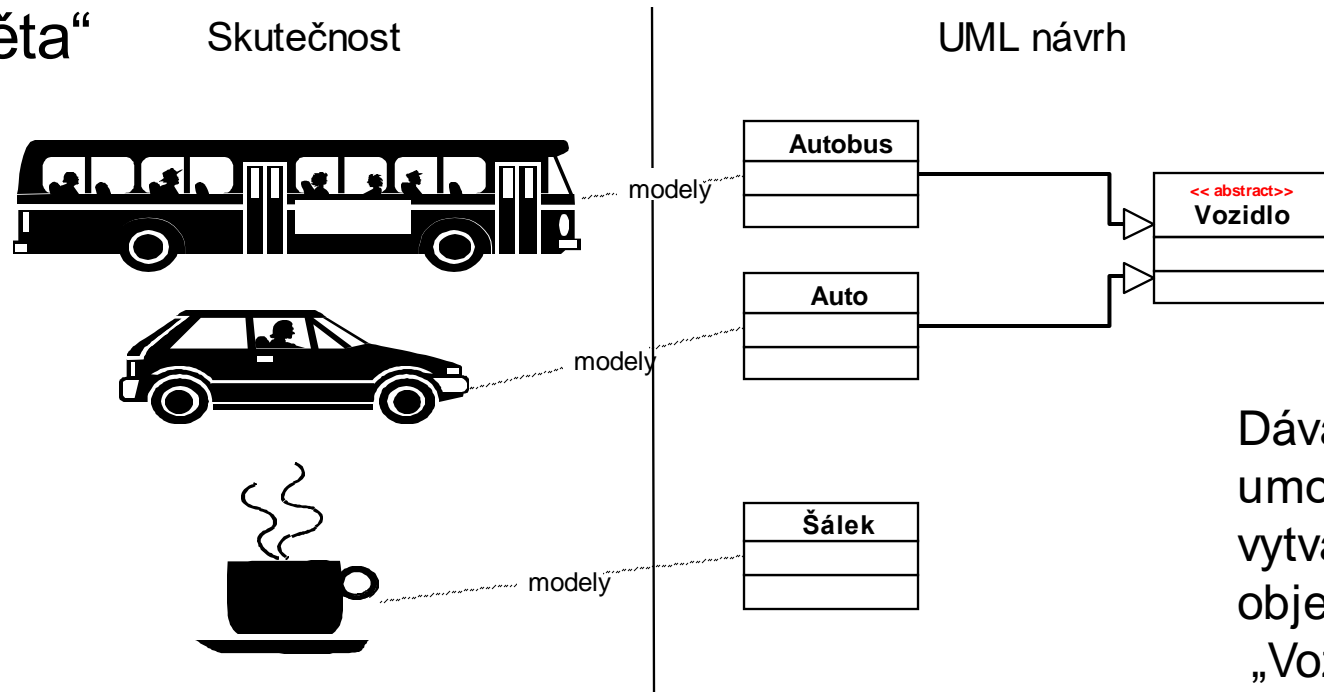
Název třídy

Atributy

Operace
(jen hlavičky
metod, ne kód)

Objektově orientovaný návrh

- Objekty jsou abstrakcí skutečných objektů z „reálného světa“



Dává smysl
umožnit
vytvářet
objekty
„Vozidlo“ ??

Atributy

**[viditelnost] název [[násobnost]] [: typ] [=výchozí hodnota]
[{příznaky}]**

- Často se zkracuje pouze na viditelnost a název
 - viditelnost: přístupová práva k atributům
 - Typ: datový typ atributu (int, float, String, Osoba, Autobus)
- (ostatní zanedbáme)

Operace

**[viditelnost] název [(výčet parametrů)] [: návratový typ]
[{příznaky}]**

- Často se zkracuje pouze na **viditelnost a název**
 - viditelnost: přístupová práva k atributům
 - Návratový typ: datový typ, který funkce vrátí (int, float, String, Osoba, Autobus)
- (ostatní parametry zanedbáme)

Viditelnost

- public (+) – mohou přistupovat vnitřní i externí objekty
- private (-) – pouze vnitřní metody
- protected (#) – jen vnitřní či speciální objekty
- *POZN.: package-private – je speciálním případem v JAVA*

Product
- serialNumber - name # price
+ buy() + display() - swap(x:int,y: int)

Vždy je snaha omezit viditelnost jak jen to lze (menší šance na chyby, které se špatně hledají)

- Omezení viditelnosti umožní zapouzdření a abstrakci (zjednodušení)

Viditelnost

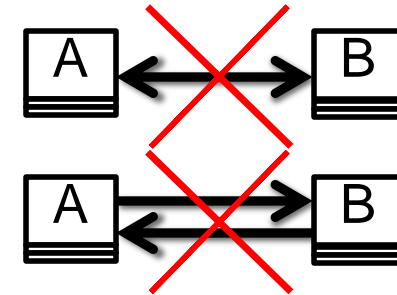
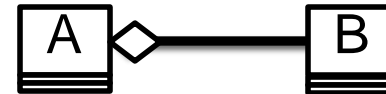
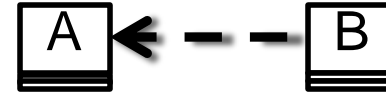
- Prostředek pro abstrakci (zapouzdření)
- Vše by mělo být maximálně schováno
- Oproti procedurálnímu přístupu je snazší porozumět, jelikož zapouzdřují (schovávají detaily)
- Omezuje se viditelnost
 - metod
 - atributů

Vztahy mezi třídami - Asociace



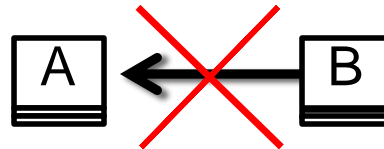
Vztahy mezi třídami - Relace

- Závislost: dynamický vztah (nejslabší)
 - Nějak závisí na
- Asociace pevný vztah
 - Je možné dostat se z A do B (ve směru šipky)
 - Pokud není orientovaná, jakoby $A \rightarrow B$ i $B \rightarrow A$
- Agregace:
 - Udrží odkaz na několik objektů
- Kompozice (nejsilnější)
 - je výhradní vlastník všech objektů



Vztahy mezi třídami - Dědičnost

- Dědičnost
 - B přebírá vlastnosti () od A (proti směru)
 - A zobecňuje B (generalizace)



Není dědičnost a
zaměnění s dědičností je
velkou chybou

Asociace



- „Objekty ve směru šipky jsou schopny nalézt odkaz na následující objekty“
- Relace přetrvává po celý čas aplikace

Násobnost



1 – pokud neuvedeno jinak

3 – přesně 3 objekty

* (nebo n , nebo $0..n$ nebo $1..*$) – libovolný počet

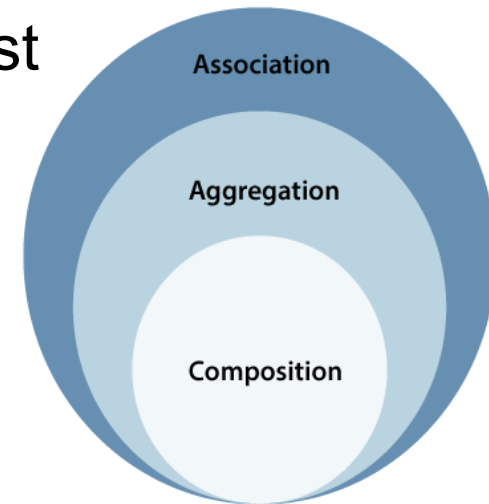
$1..*$ (či $1..n$) 1 až nekonečno

$3..9$ – 3 až 9 (ale nepoužívá se často)

Agregace	Kompozice
<p>Dokumenty mohou být součástí více kategorií</p> 	<p>Tabulka okna může být jen v jediném okně (je nesmysl, aby byla současně ve dvou oknech)</p> 
<p>Části mohou existovat nezávisle na sobě násobnost je vždy 0..* a proto se násobnost zpravidla neuvádí</p>	<p>Části existují jako součást celku. Když je okno zrušeno, tabulky okna jsou také zrušeny násobnost je vždy 0..* a proto se násobnost zpravidla neuvádí</p>
<p>Celek není zodpovědný za podčásti (speciální případ asociace)</p>	<p>Celek je zodpovědný a zodpovídá za vytváření/mazání objektů (speciální případ agregace)</p>

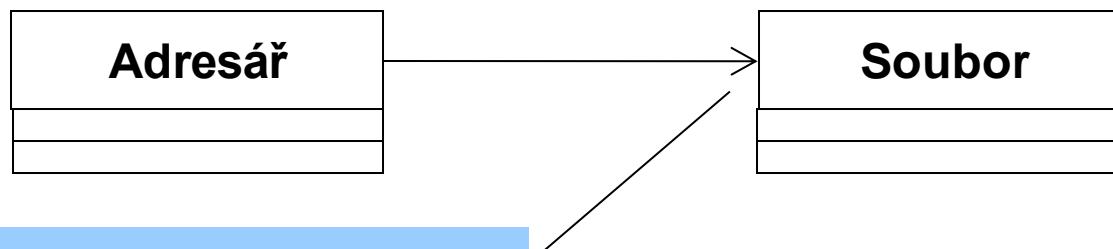
Kompozice vs. Agregace

- Nejste-li si jisti, zdali použít Kompozici, Agregaci či obecnou Asociaci, zvolte asociaci – nejobecnější a tím pádem nelze nic zkazit
- Agregace zdůrazňuje vztah celek - součást
- Kompozice se používá méně často



Orientace asociace

- Jeli asociace orientovaná, jsem schopen zjistit referenci na objekt jen ve směru orientace
- Není-li orientovaná, jedná se o oboustranně orientovanou

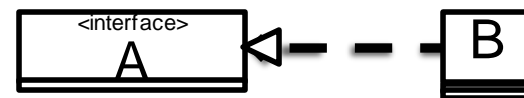


Šipkou od adresáře k souboru říkám, že pro daný adresář, jsem schopen zjistit soubor. Naopak, pro ze souboru nejsme schopni zjistit, v jakém je adresáři.

Relace - Dědičnost



Dědičnost



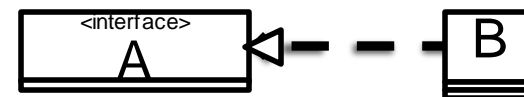
Implementuje

- - Častá situace – některé části kódu jsou pro dvě třídy společné – řešení:
 - Duplicita kódu (špatná varianta)
 - Dědičnost
 - B „přeber vlastnosti“ z A (všechny atributy a metody)
 - Je hlavní předností Objektově Orientovaného Návrhu
 - Umožňuje snadnou rozšiřitelnost do budoucna
 - Nemá nic společné s asociací

Relace - Dědičnost



Dědičnost

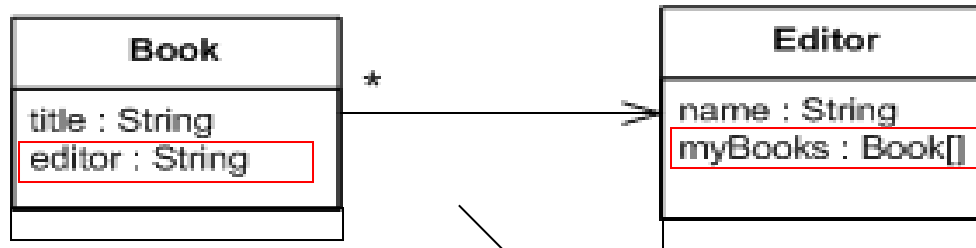


Implementuje

- Dědičnost: A zobecňuje (**generalizuje**) B
- Implementuje: B implementuje A
 - Všem „hlavičkám metod“ dodá společné rozhraní (chování)
- Z B je možné použít všechny (+,#) funkce či atributy třídy A.
- Vícetypovost – na objekty z třídy B lze odkazovat proměnnou typu A i B.

Atributy

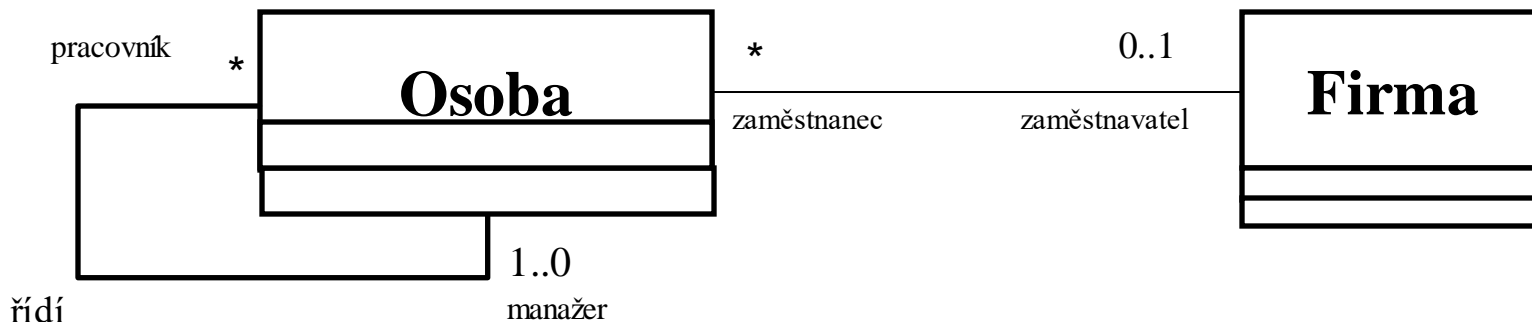
- Relace je vyznačena atributy, nikoli asociacemi relací.
- Významově to samé, měla by být upřednostněna varianta s relací – čitelnější pro člověka.



Jaký je problém?

Názvy rolí

- Z důvodu srozumitelnosti je možné doplnit názvy rolí, pod kterými vystupují
- Poskytuje lepší srozumitelnost vazeb
- Mělo by být použito vždy u sebe-asociujících relací

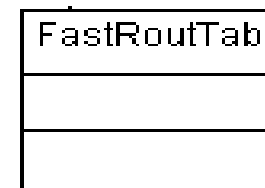
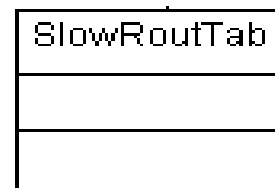
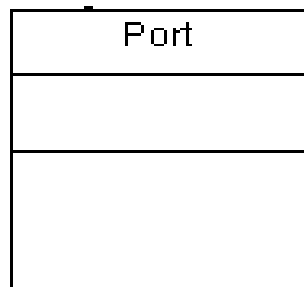
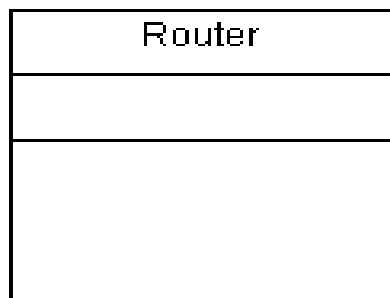


Příklad – směrovač (pokračování)

- Dodavatel bude uvádět na trh 2 typy směrovačů: levnější (SW realizace) a dražší (např. HW realizace)
- Je zřejmé, že nějaké části kódu budou stejné
- Navrhněte tak, aby jste zamezili duplicitám v kódu.

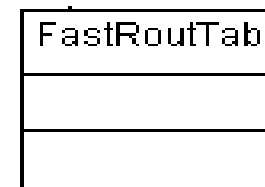
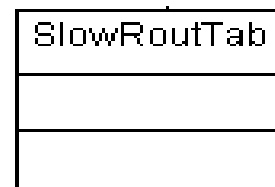
Příklad: Směrovač pokračování

- Pohled z úhlu jazyka UML – rozdělení na funkční bloky



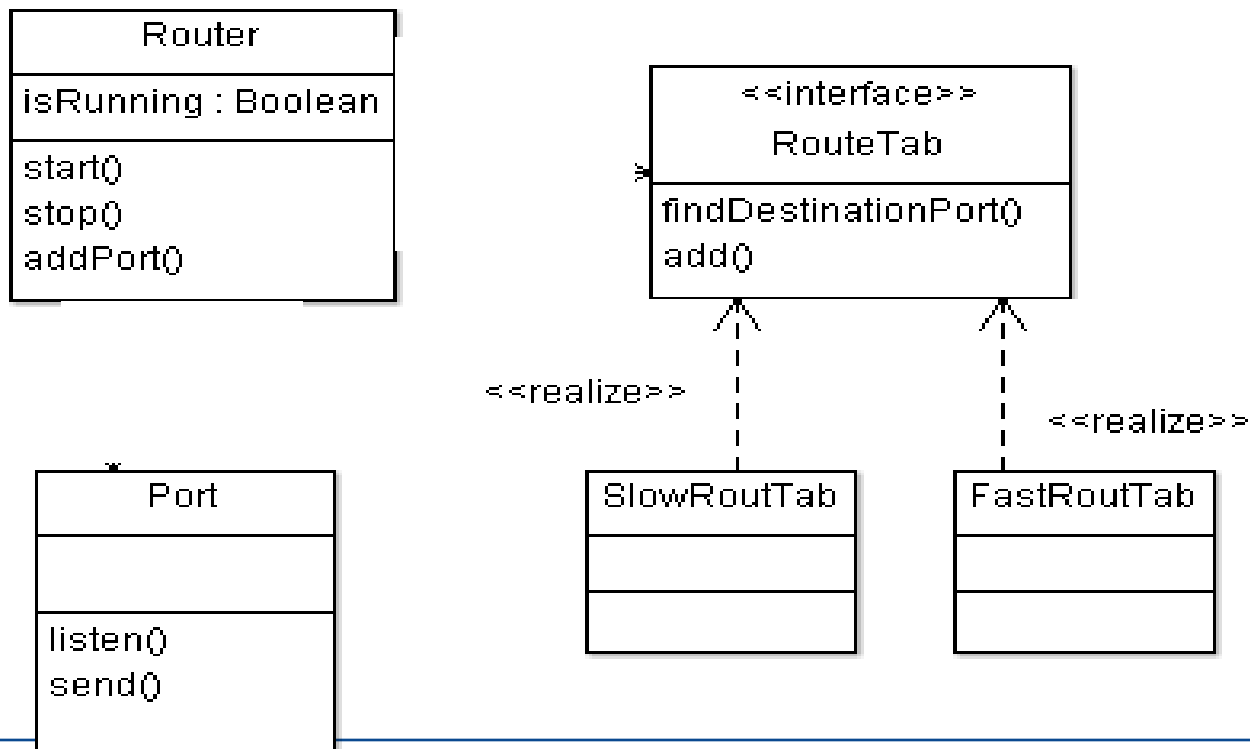
Příklad: Směrovač pokračování

- Pohled z úhlu jazyka UML – rozdělení na funkční bloky



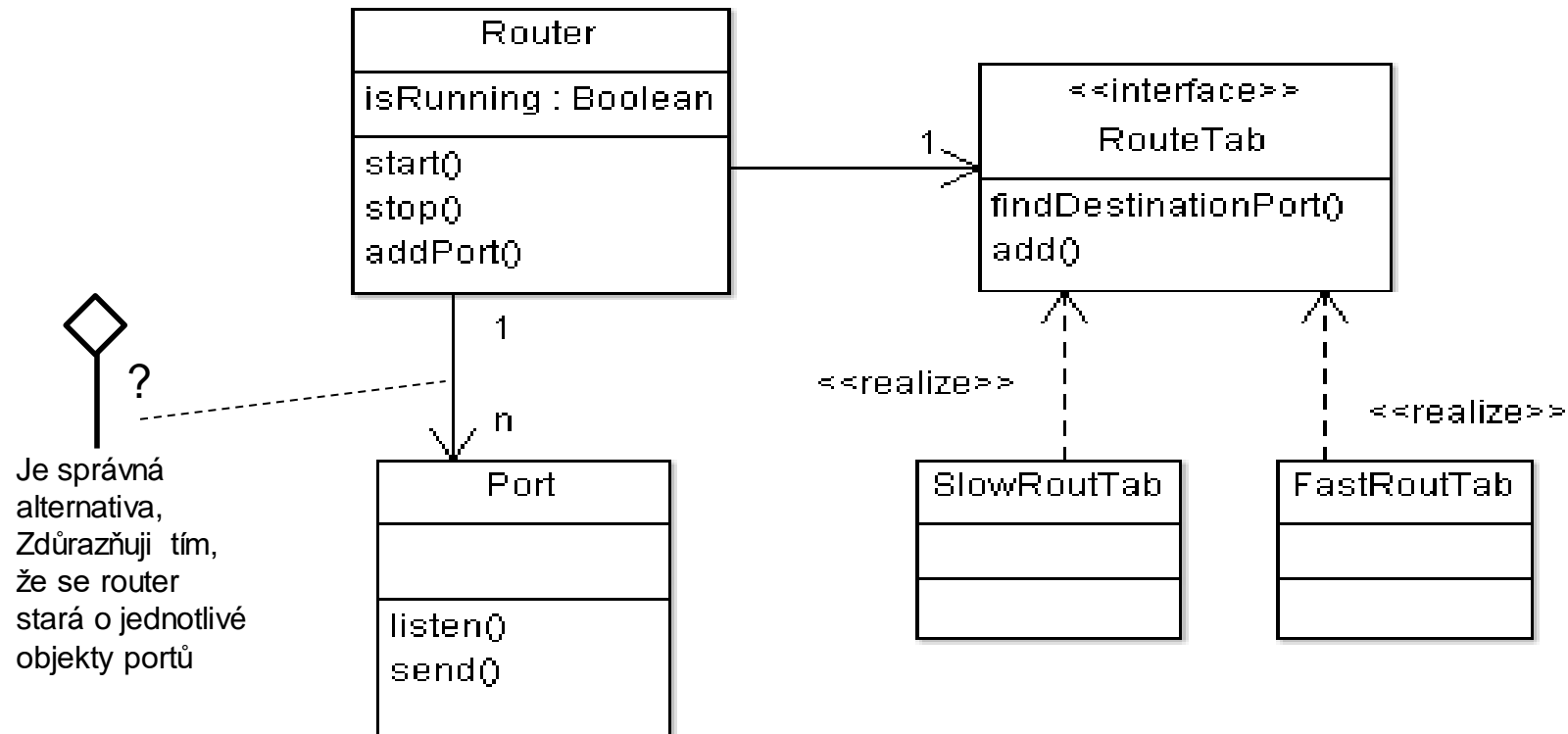
Příklad: Směrovač pokračování

- Najdu společné bloky a zavedu zobecňující třídy => odstraním případné duplicity kódu=>méně kódu



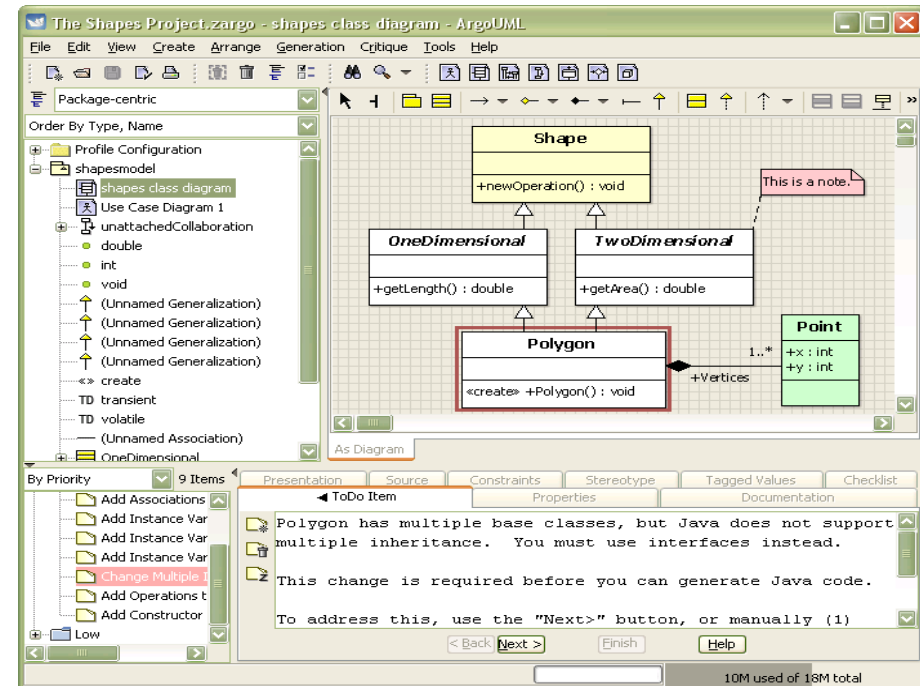
Příklad: Směrovač pokračování

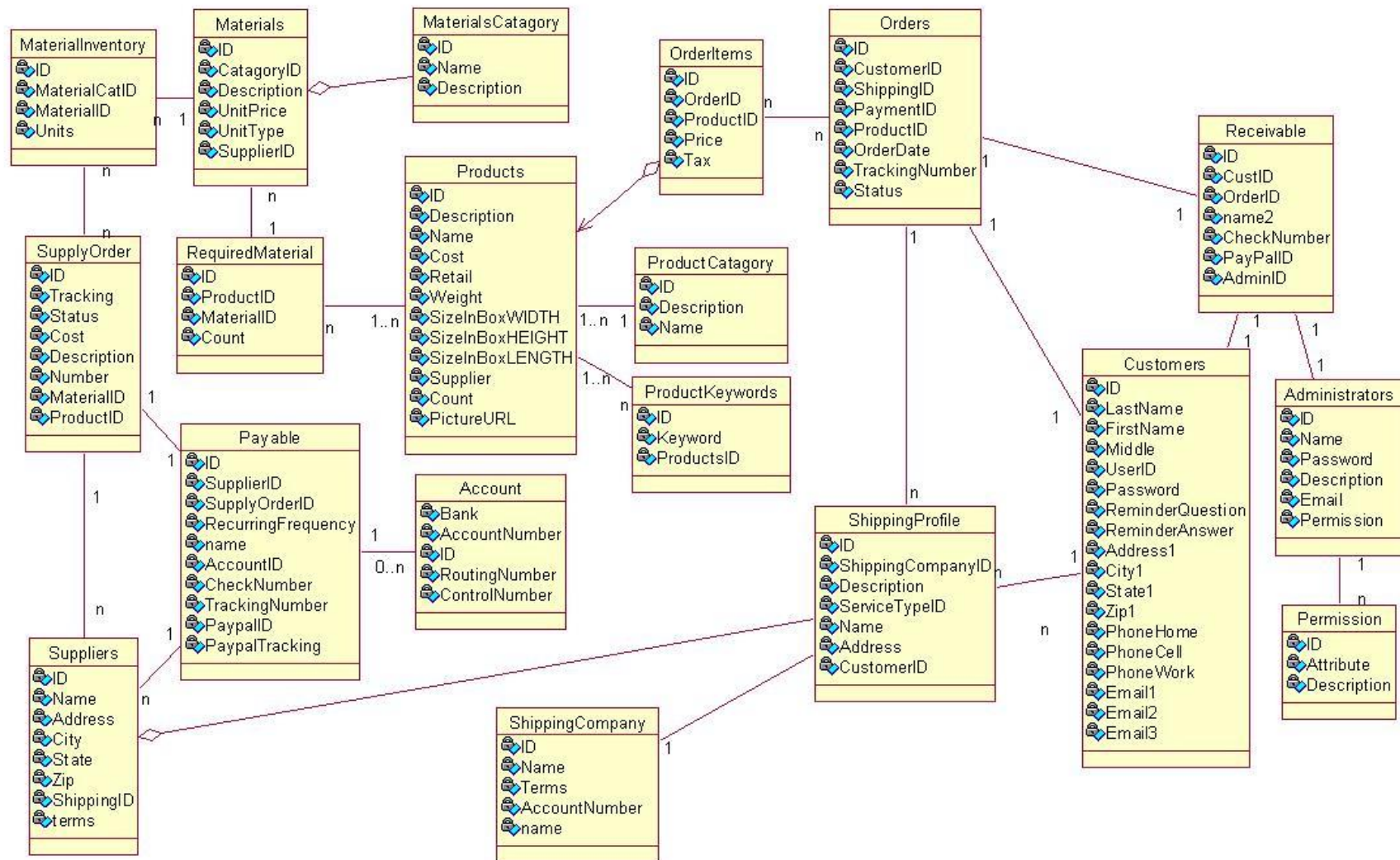
- Dodám asociace



Software pro UML modelování

- RationalRose
- **ArgoUML** (<http://argouml.tigris.org/>)
- Dia
- MS Visio
- Visual Paradigm
- Doporučuji pro procvičení





Jak přepsat UML do programovacího jazyka

- Máme slovní popis, UML popis a je možné předešlé zapsat i OO jazykem (JAVA, C++)
- Jsou tyto zápisy významově ekvivalentní?
 - ANO
- Proč tedy 3 různé zápisy
 - 1. – nejvhodnější pro člověka – přesně popisuje konkrétní problém, je zapotřebí detailně rozumět dané oblasti.
 - 2. – univerzální popis – netřeba detailně rozumět dané oblasti, budou tomu rozumět i další lidé po celém světě.
 - 3. – pro člověka hůře čitelný, nezbytný pro specifikaci všech detailů stroji.

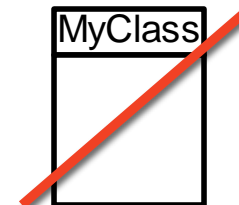


Jednoduchá Třída

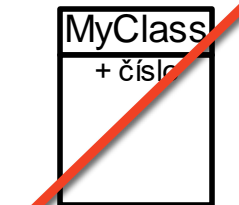


```
1: class MyClass
2: {
3: }
```

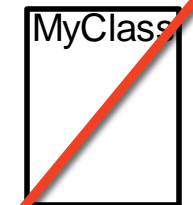
Poznámka:



Není třída,
ale interface



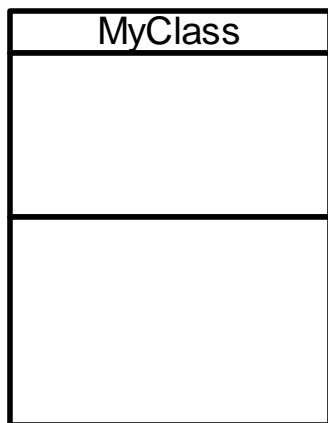
Interface, ten
ale smí mít
pouze
metody, ne
atributy



Je nesmysl,
není dle UML
nic

Jak vytvořit první spustitelnou aplikaci?

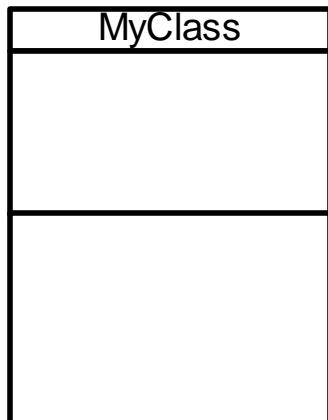
- Metoda **main** s OON nemá nic společné, **main** se zpravidla neznačí v UML diagramu tříd (diagram sekvencí)
- Dobrým zvykem dávat spouštěcí třídu zcela samostatně



```
1: class MyClass
2: {
3:     public static void main(String args[])
4:     {
5:         System.out.println("Hello World!");
6:     }
7: }
```

↑
Není strukturální, ale behaviorální
=> není v diagramu tříd uveden

Jak z třídy vytvořit první objekt?

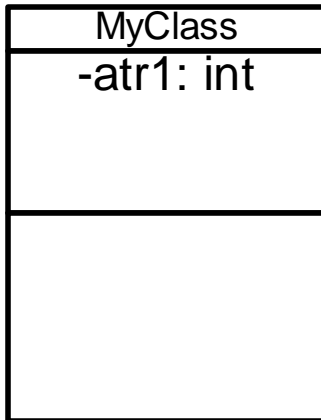


```
1: class MyClass
2: {
3:     public static void main(String args[])
4:     {
5:         MyClass mojeRef = new MyClass();
6:     }
7: }
```

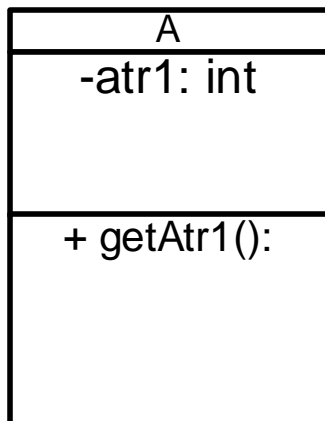
↑
Není strukturální, ale behaviorální část
=> není v diagramu tříd uveden

Jednoduchá třída + atribut

```
1: class MyClass  
2: {  
3:     private int atr1;  
4: }
```

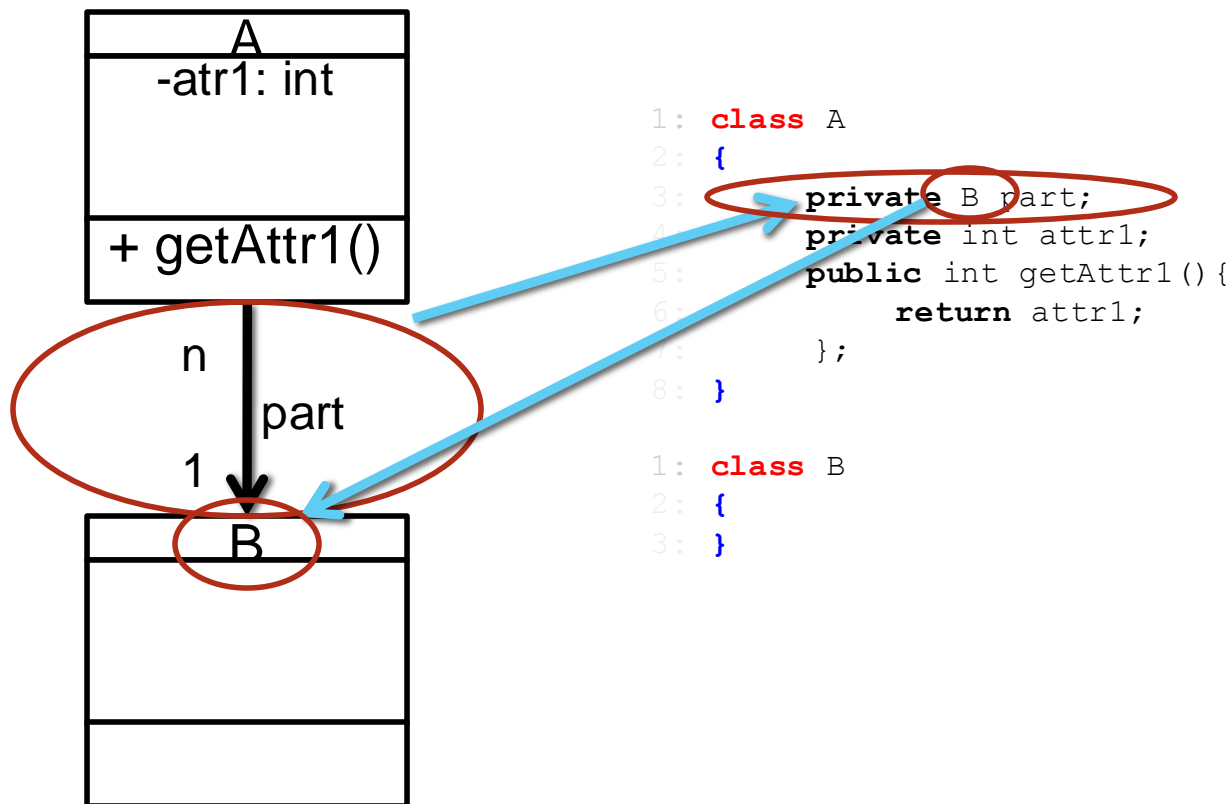


Jednoduchá třída + atribut a metoda

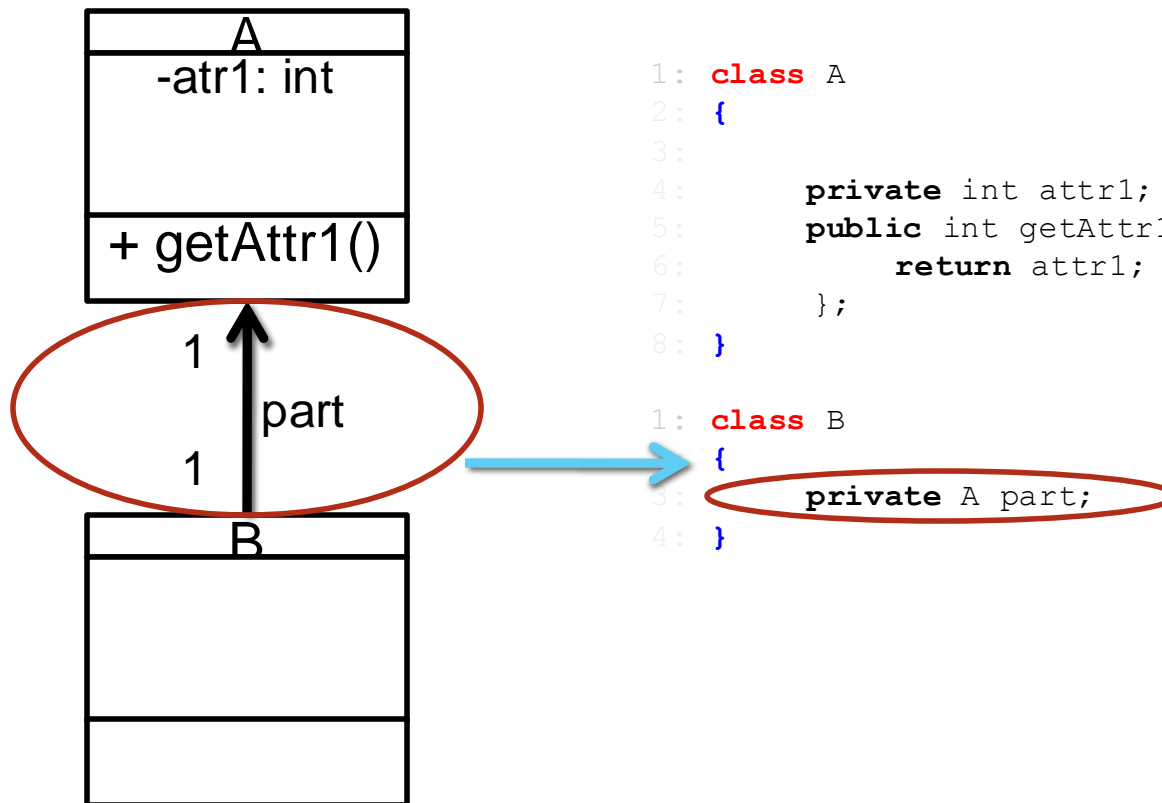


```
1: class A
2: {
3:     private int atr1;
4:     public int getAtr1() {;
5:         return atr1;
6:     }
7: }
```

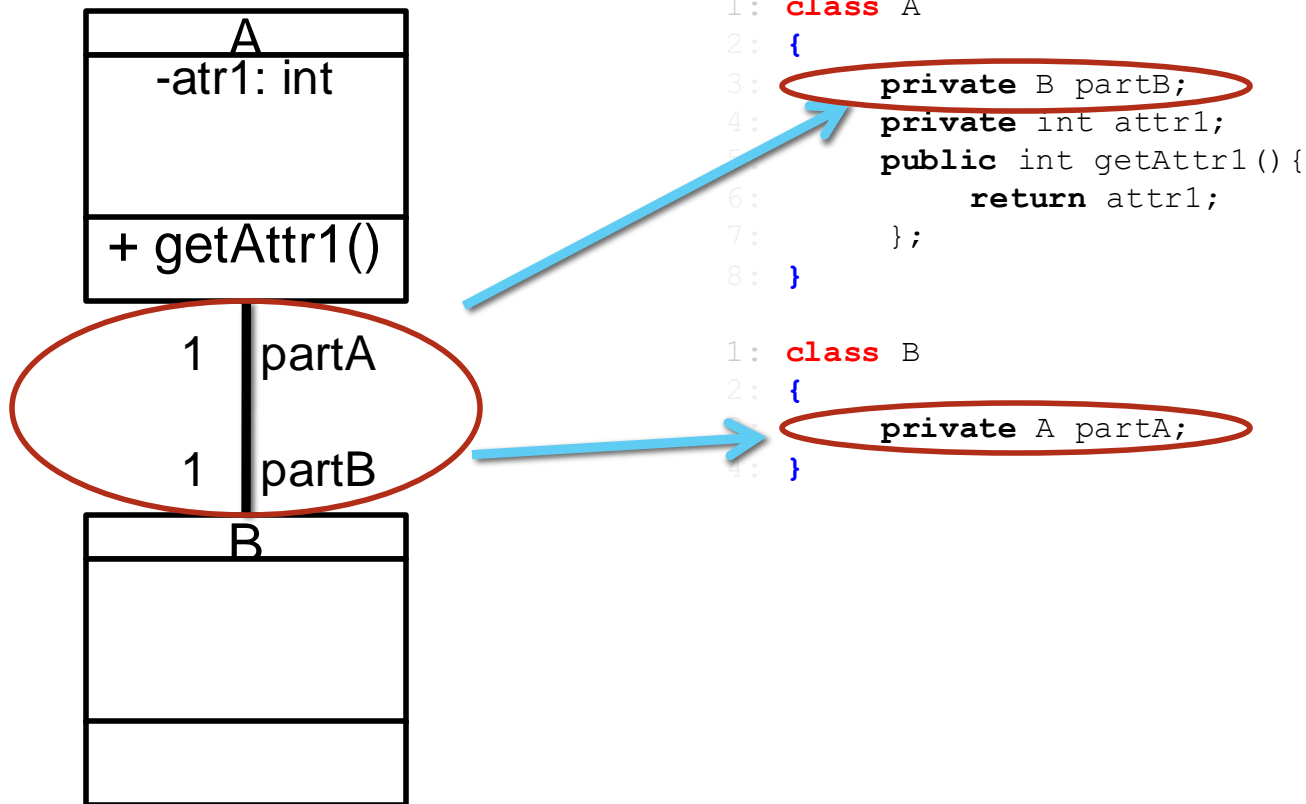
2 třídy + a orientovaná asociace



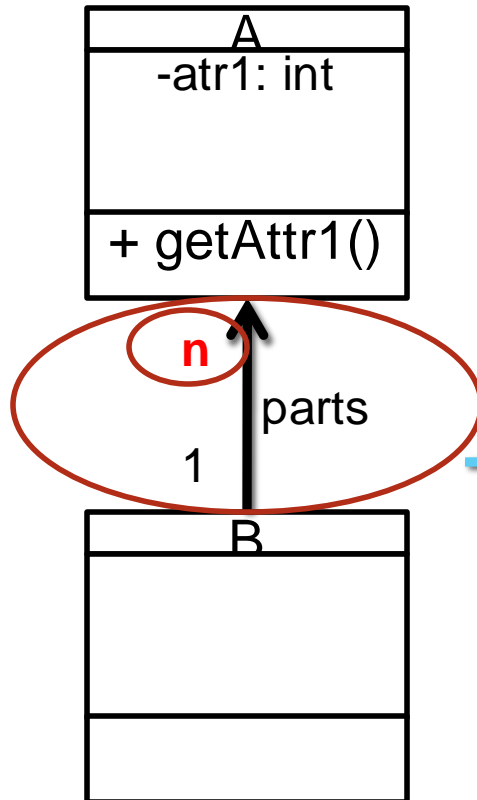
2 třídy + a orientovaná asociace



2 třídy + a neorientovaná asociace



2 třídy + a orientovaná n-násobná asociace



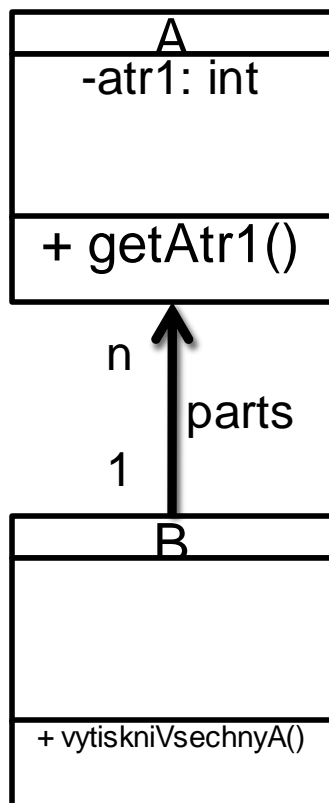
```
1: class A
2: {
3:
4:     private int attr1;
5:     public int getAttr1() {
6:         return attr1;
7:     };
8: }
```

```
1: class B
2: {
3:     private Vector<A> parts = new Vector<A>();
4: }
```

Vazba 1:N

- Jakým způsobem reprezentovat vazbu cokoli:N?
 - ArrayList
 - Vector
 - HashSet
 - HashMap
 - TreeSet
 - TreeMap
 - ...
- Ve skutečnosti toto rozhoduje o výkonnosti výrazně více nežli volba programovacího prostředí
- Jim se budeme věnovat několik následujících přednášek...

2 třídy + a orientovaná n-násobná asociace



Jak zajistit, vytisknutí všech hodnot attr1 třídy A, které jsou v třídě B?

Vypsání textu: System.out.println("AHOJ");

```

1: class A
2: {
3:
4:     private int atr1;
5:     public int getAtr1(){
6:         return atr1;
7:     };
8: }

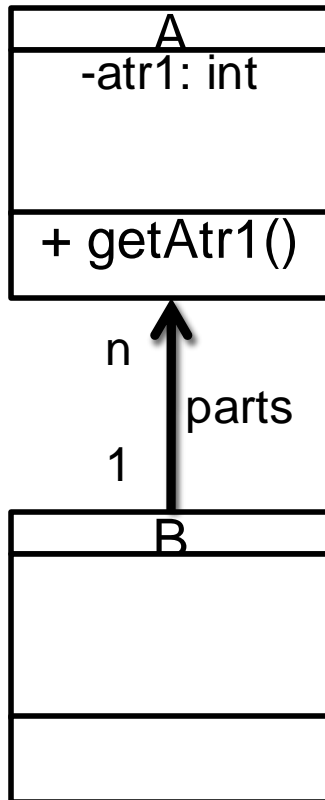
1: class B
2: {
3:     private Vector<A> parts = new Vector<A>();

4:     public void vytiskniVsechnyA() {
5:         for(int i = 0; i < parts.size(); i++) {
6:             System.out.println("Value"+
7:                 parts.get(i).getAtr1());
8:         }
9:     }
10: }
  
```

2 třídy + a orientovaná n-násobná asociace

Jak zajistit, vytisknutí všech hodnot attr1 třídy A, které jsou v třídě B?

Vypsání textu: `System.out.println("AHOJ");`



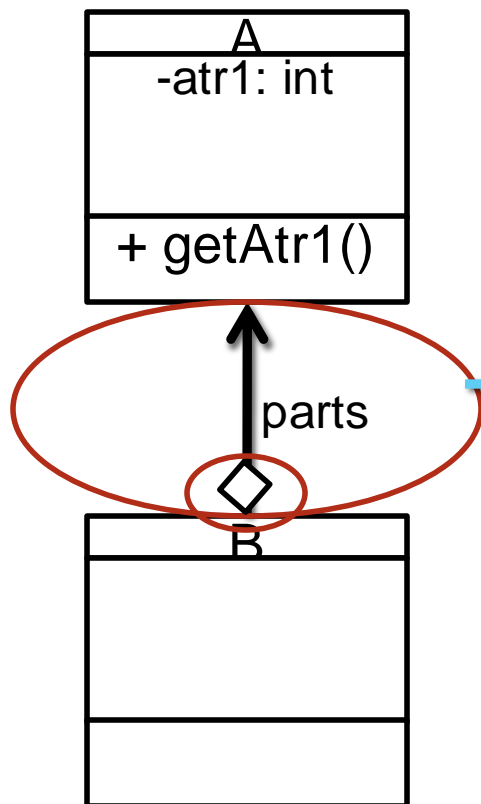
```

1: class A
2: {
3:
4:     private int atr1;
5:     public int getAtr1(){
6:         return atr1;
7:     };
8: }

1: class B
2: {
3:     private Vector<A> parts = new Vector<A>();

4:     public void vytiskniVsechnyA() {
5:         for(A tmp : parts) {
6:             System.out.println("Value"+
7:                 tmp.getAtr1());
8:         }
9:     }
10: }
  
```

2 třídy + a agregace



```
1: class A
2: {
3:
4:     private int atr1;
5:     public int getAtr1(){
6:         return attr1;
7:     };
8: }
```

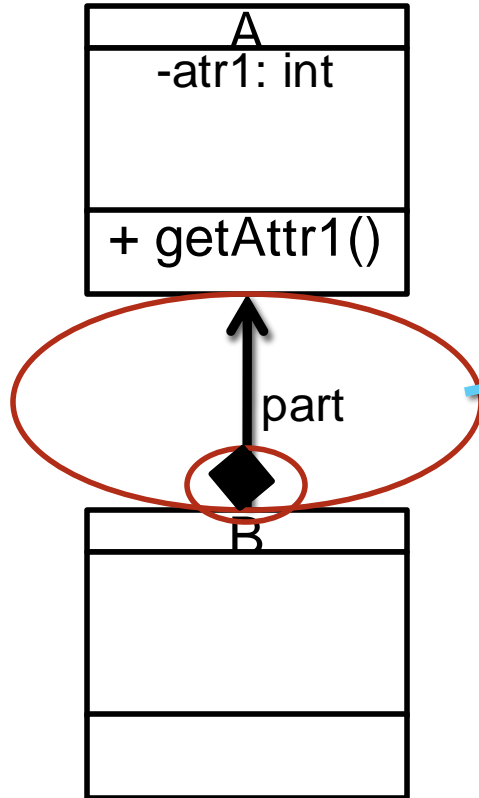
```
1: class B
2: {
3:     private Vector<A> parts = new Vector<A>();
4: }
```

Stejný kód. Navíc ale říkám, že se o objekty stará třída B.

U agregace se neuvádí násobnost (vždy je to 1:N)

Pokud si nejste jistí – použít asociaci

2 třídy + a kompozice



```

1: class B
2: {

```

```

1: class A
2: {
3:
4:     private int attr1;
5:     public int getAttr1() {
6:         return attr1;
7:     };
8: }

```

```

3:     private Vector<A> part = new Vector<A>();
4: }

```

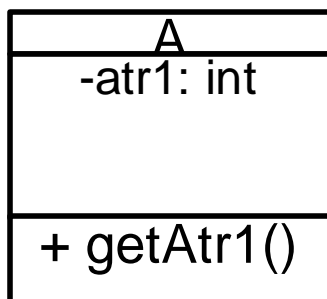
Stejný kód. Navíc ale říkám, že se o objekty stará výhradně třída B a nikdo jiný.

A je pevnou součástí B

Neuvádí se násobnost (vždy je to 1:N)

Pokud si nejste jistí – použít asociaci

2 třídy + a dědičnost



```
1: class A
2: {
3:
4:     private int attr1;
5:     public int getAttr1(){
6:         return attr1;
7:     };
8: }
```

```
1: class B extends A
2: {
3:
4: }
```

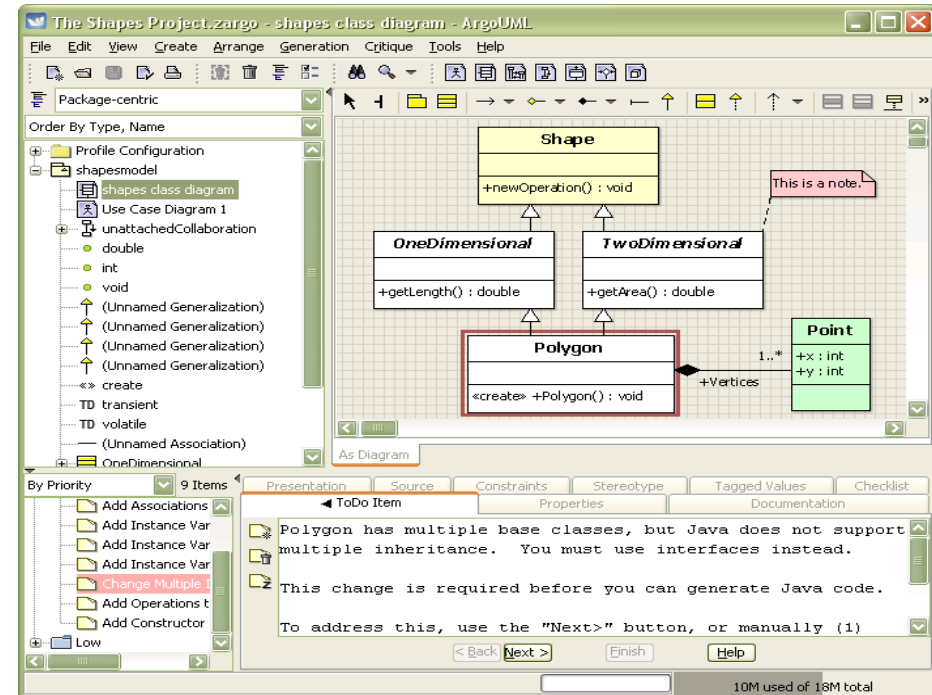
B přebírá vlastnosti z A, atribut `attr1` a metodu `getAttr1()`.

Násobnost je nesmysl v tomto případě

```
B mojeProm = new B();
mojeProm.getAttr1();
```


Tip

- ArgoUML umí jak kreslit, tak generovat kód a to nejen pro JAVA ale i do mnoha dalších jazyků.
- Zdarma (open-source)
- Je možné experimentovat



Provázanost kódu

- Dobré: hard disk, cdrom, floppy



- Špatné: spaghetti code

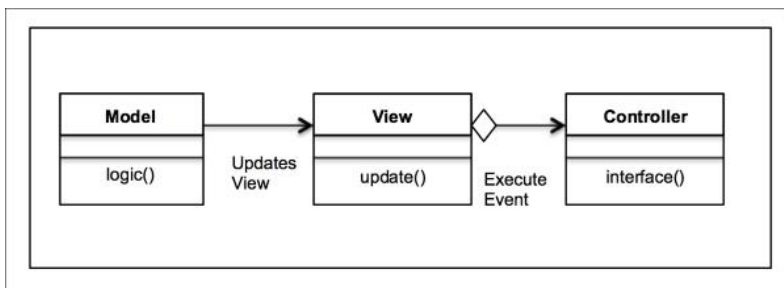


Návrhové vzory

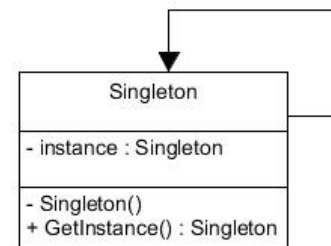
- Za řadu let se ustálily ověřené:
 - Zpřehledňují kód
 - Umožňují vývoj znovupoužitelných částí / komponent
 - Umožňují snadnou rozšiřitelnost a flexibilitu změn
- Měli byste vědět, že existují
- Příklad:
 - Jedináček
 - Model-View-Controller
- *Pozn.: Chcete lépe porozumět myšlence OON? – podívejte se na návrhový vzor Composite.*

Vybrané návrhové vzory

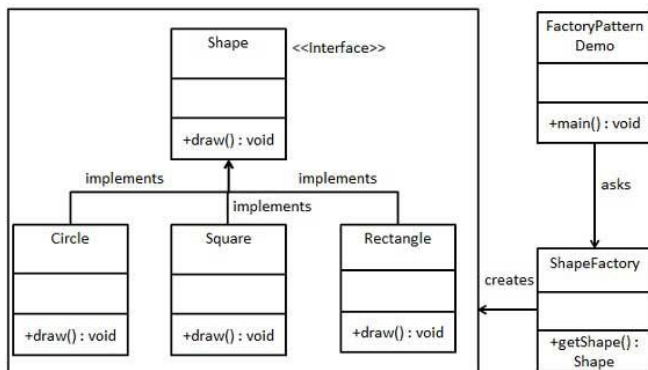
Model Pohled Kontrolér:



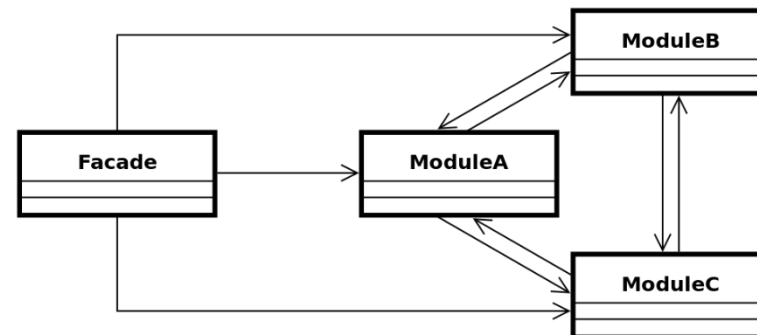
Jedináček:



Továrna:



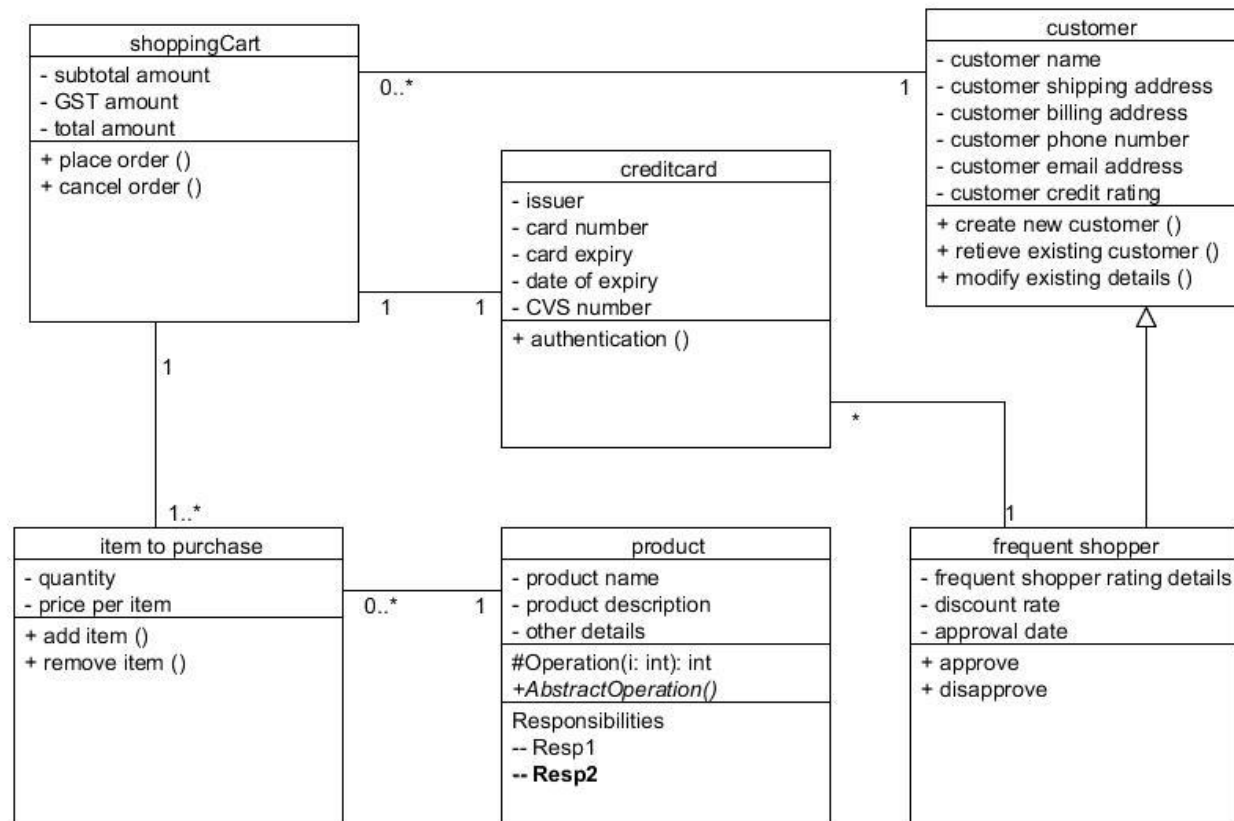
Fasáda:



Příklad 1: S pomocí UML diagramu tříd popište

- Vytvořme datový model pro e-shop. O každém produktu a zákazníkovi jsou udržovány základní informace
 - Produkt: název, cena
 - Zákazník: jméno, adresa
- Speciální případ zákazníka je „častý zákazník“, který navíc může udržovat i informaci o platebních kartách. V jejich případě je košík placen právě skrze kartu.
- E-shop uchovává informaci o zakoupeném zboží, kde je nutno uchovat cenu v době zakoupení zboží (nikoli aktuální cenu produktu, která se může měnit)

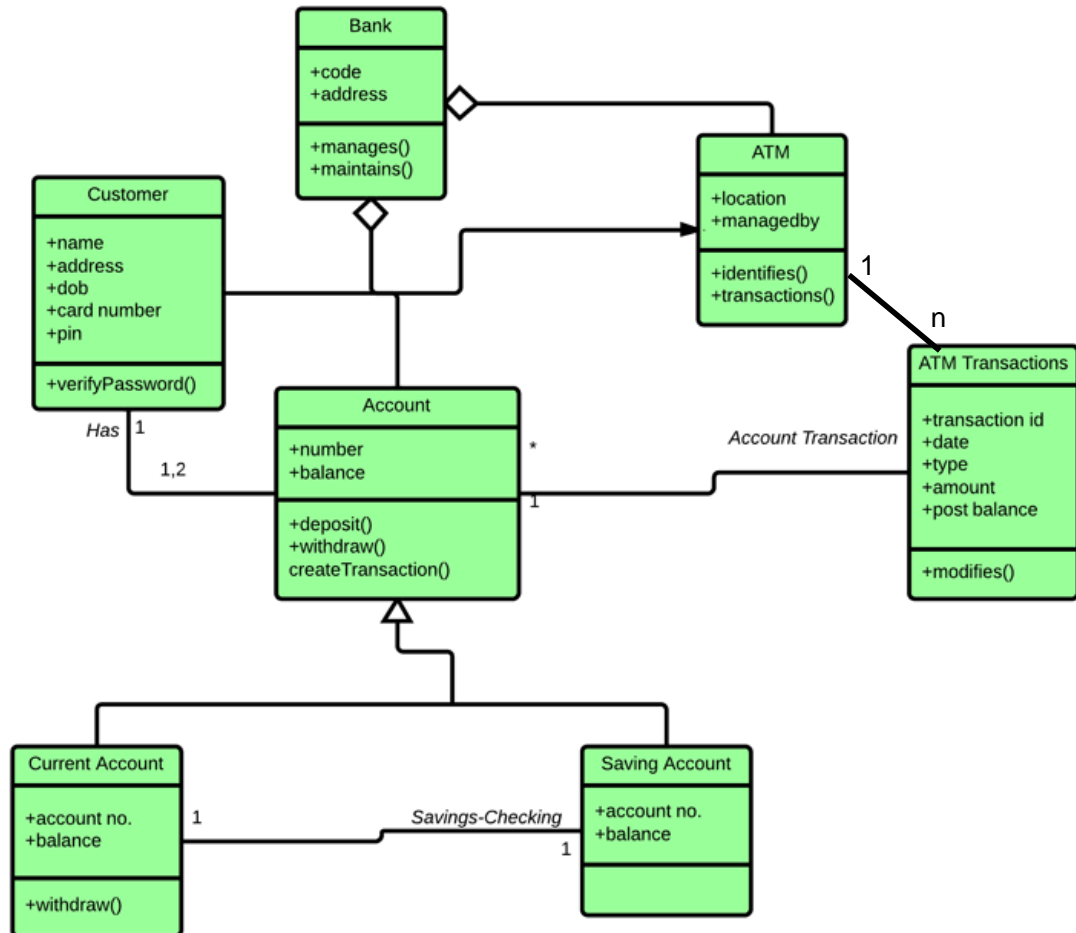
Příklad 1 – možné řešení



Příklad 2

- Navrhněte datový model pro bankovní systémy. Každá banka je identifikován číslem účtu a adresou a každý zákazník může mít účet u více bank. Účty rozlišujeme na standardní a spořicí. Každá banka má různé bankomaty ATM, kde chceme uchovávat informace o jejich poloze.

Příklad 2 - řešení



Shrnutí

- Objektově orientovaného návrh je bližší světu známého z reálného života, lze snáze modelovat složitější případy
- Mezinárodní standard používaný po celém světě
- Seznámit se jak koresponduje OON s programovým kódem
- Příklady

Pojmy

- Návrh software
- Diagram tříd
 - Třída
 - Objekt
 - UML + typy diagramů
 - Relace
 - Asociace (agregace, kompozice)
 - Generalizace
 - Návrhový vzor
 - Vektor

Děkuji za pozornost