

# Úvod do softwarového inženýrství

## IUS

### 2024/2025

#### 4. přednáška

Ing. Radek Kočí, Ph.D.  
Ing. Bohuslav Křena, Ph.D.

7. a 11. října 2024

# Téma přednášky

- Objektově orientované modelování
- Jazyk UML
  - Diagram tříd
  - Diagram objektů
- Principy objektového návrhu

# Objektově orientované modelování

## Objektově orientovaný přístup k modelování a vývoji systémů

- kolekce vzájemně komunikujících objektů
- soubor objektově orientovaných prostředků (objekty, třídy, UML, ...) a metodik (RUP, ...)
- vykazuje vyšší stabilitu navrhovaných prvků z pohledu měnících se požadavků
- *Objektový návrh nutně neimplikuje objektovou implementaci!*

## Objekt reprezentuje entitu řešeného problému

- má jasně vymezenou roli (*zodpovědnost*)
- zná sám sebe (*identita*)
- uchovává data (*stav*)
- má metody (*chování*)
- umí zpracovávat a posílat zprávy (*protokol*)

# OO modelování – Vlastnosti

## Abstrakce (*Abstraction*)

- vytvářený systém objektů je abstrakcí řešeného problému (zjednodušený pohled na systém bez ztráty jeho významu)
  - *analýza problému*  $\Rightarrow$  *klasifikace do abstraktních struktur*
    - rozpoznávání podobností v řešené problematice
    - entity *klient*  $\Rightarrow$  entitní množina *Klient*
    - objekty *klient*  $\Rightarrow$  třída *Klient*
- snaha o abstrakci abychom byli schopni dekompozice a jasného vymezení problému

## Třídy objektů

recept na vyvareni objektu  
a tím padem je takv seskupuie

- seskupení objektů do tříd podle podobnosti (typu)
- třída je
  - generická definice (šablona) pro množinu objektů stejného typu
  - množina objektů se stejným chováním a stejnou množinou atributů
- objekt (konkrétní jedinec) je instancí třídy

# OO modelování – Vlastnosti

## Zapouzdření (*Encapsulation*)

- seskupení souvisejících idejí (data, funkcionalita) do jedné jednotky
- *seskupení operací a atributů do jednoho typu objektu (třídy) – stav je dostupný či modifikovatelný pouze prostřednictvím rozhraní (sady operací)*
- *omezení externí viditelnosti informací nebo implementačních detailů*

## Strukturovaný přístup

funkce	data
<pre>int obsahObdelniku(int x, int y) {     return x * y; }</pre>	<pre>(20, 10) (15, 20)</pre>
<pre>int obsahTrojuhelniku(int x, int y) {     return (x * y) / 2; }</pre>	<pre>co data reprezentují?  if (trojuhelnik) ...</pre>

# OO modelování – Vlastnosti

## Zapouzdření (*Encapsulation*)

- seskupení souvisejících idejí (data, funkcionalita) do jedné jednotky
- *seskupení operací a atributů do jednoho typu objektu (třídy) – stav je dostupný či modifikovatelný pouze prostřednictvím rozhraní (sady operací)*
- *omezení externí viditelnosti informací nebo implementačních detailů*

shrňeme všechno do jedné skupiny zapoudíme a tím jim dáme jasné pravidla jak se skupině přistupovat a jsme tak schopni zajistit větší bezpečnost

## Strukturovaný přístup

funkce	data
<pre>int obsahObdelniku(struct o e) {     return e.x * e.y; }</pre>	<pre>struct o {     int x;     int y; }</pre>
<pre>int obsahTrojuhelniku(struct t e) {     return (e.x * e.y) / 2; }</pre>	

# OO modelování – Vlastnosti

## Zapouzdření (*Encapsulation*)

- seskupení souvisejících idejí (data, funkcionalita) do jedné jednotky
- *seskupení operací a atributů do jednoho typu objektu (třídy) – stav je dostupný či modifikovatelný pouze prostřednictvím rozhraní (sady operací)*
- *omezení externí viditelnosti informací nebo implementačních detailů*

## Objektový přístup

```
class Obdelnik {  
    int x;  
    int y;  
    int obsah() {  
        return x * y;  
    }  
}
```

`o.obsah()`

```
class Trojuhelnik {  
    int x;  
    ...  
    int obsah() {  
        ...  
    }  
}
```

`t.obsah()`

```
class Obdelnik {  
    Point p1;  
    Point p2;  
    int obsah() {  
        ...  
    }  
}
```

`o.obsah()`

# OO modelování – Vlastnosti

```
class Obdelnik {  
    int x;  
    int y;  
    int obsah() {  
        return x * y;  
    }  
}
```

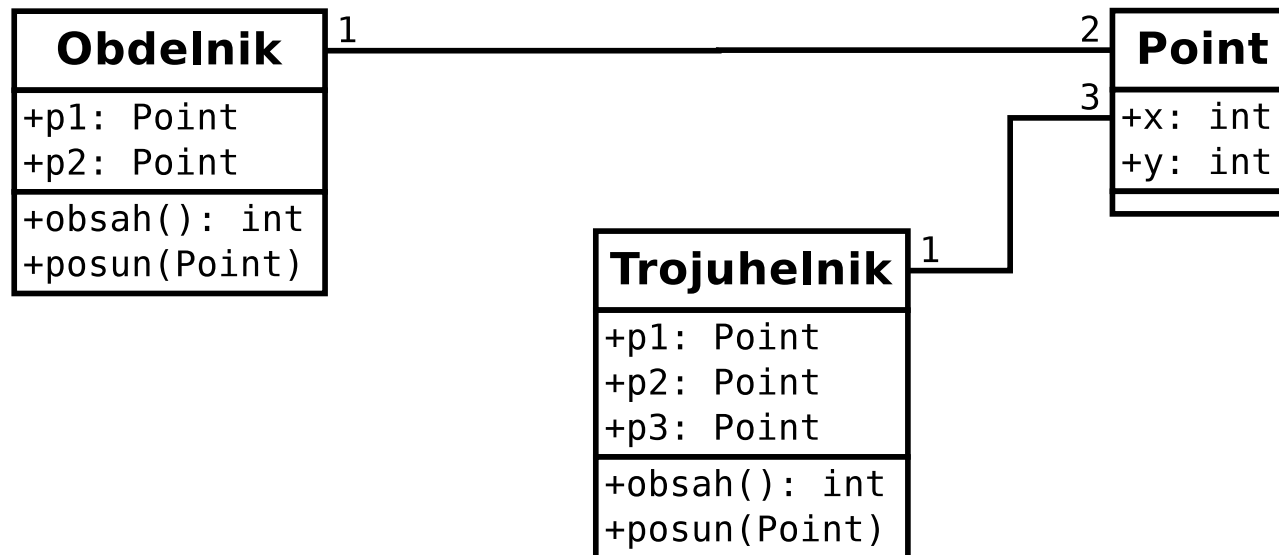
o.obsah()

```
class Trojuhelnik {  
    int x;  
    ...  
    int obsah() {  
        ...  
    }  
}
```

t.obsah()

```
class Obdelnik {  
    Point p1;  
    Point p2;  
    int obsah() {  
        ...  
    }  
}
```

o.obsah()



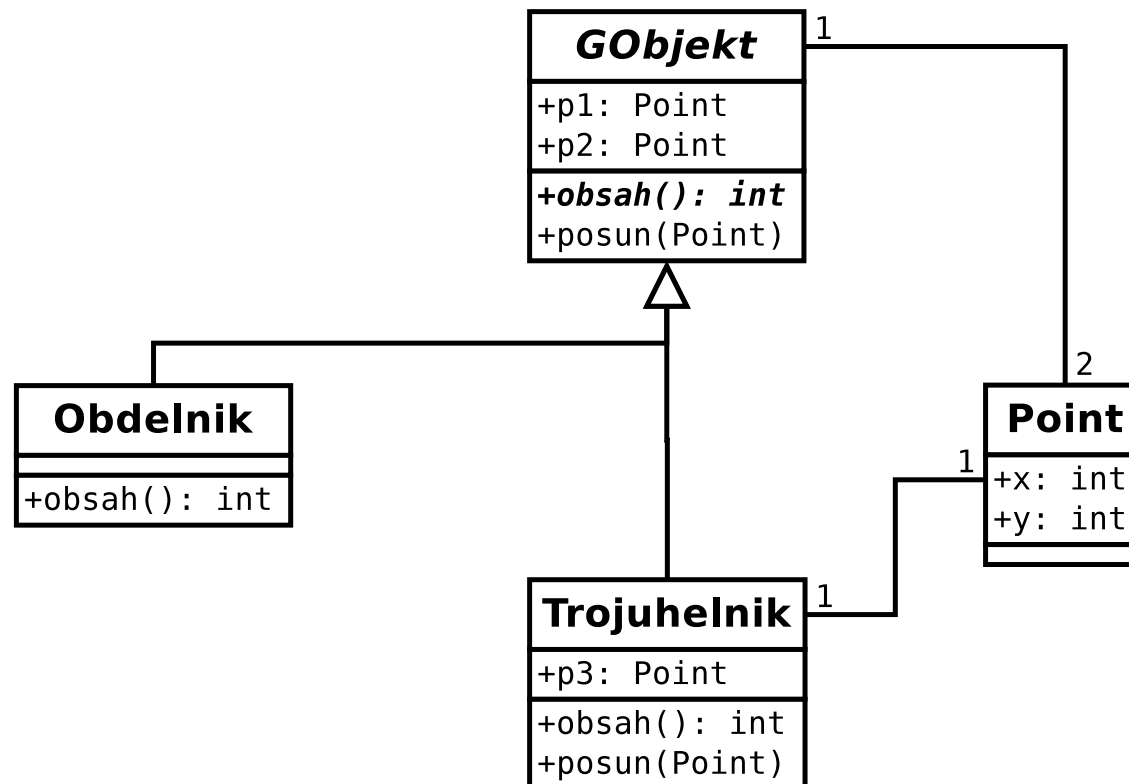


# OO modelování – Vlastnosti

## Dědičnost (*Inheritance*)

- definuje a vytváří třídy (objekty) na základě již existujících tříd (objektů)
  - možnost sdílení chování bez nutnosti reimplementace
  - možnost rozšíření chování
- *mezi třídami (objekty) vzniká hierarchický vztah podle dědičnosti (strom)*

Mensi opakovani kodu tim vetsi cistota kodu take jsem schopni upravovat nektre vlastnosti pouze na jednom miste

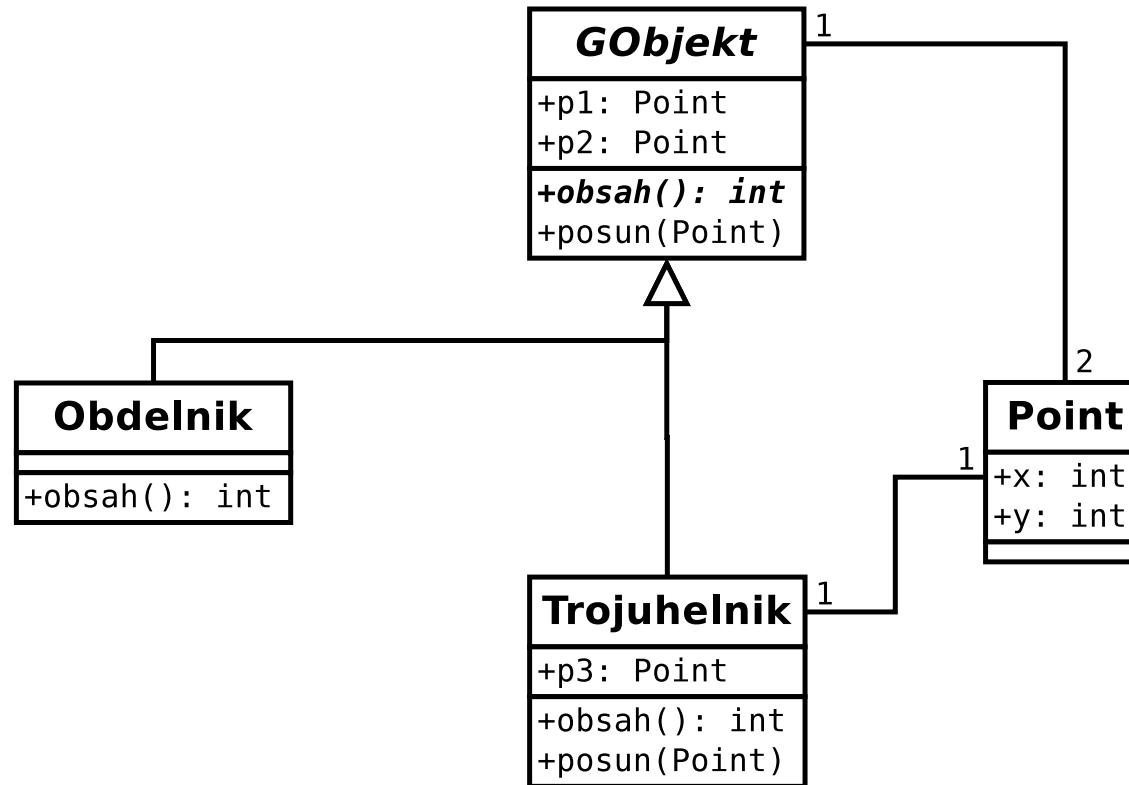


# OO modelování – Vlastnosti

## Polymorfismus (*Polymorphism*)

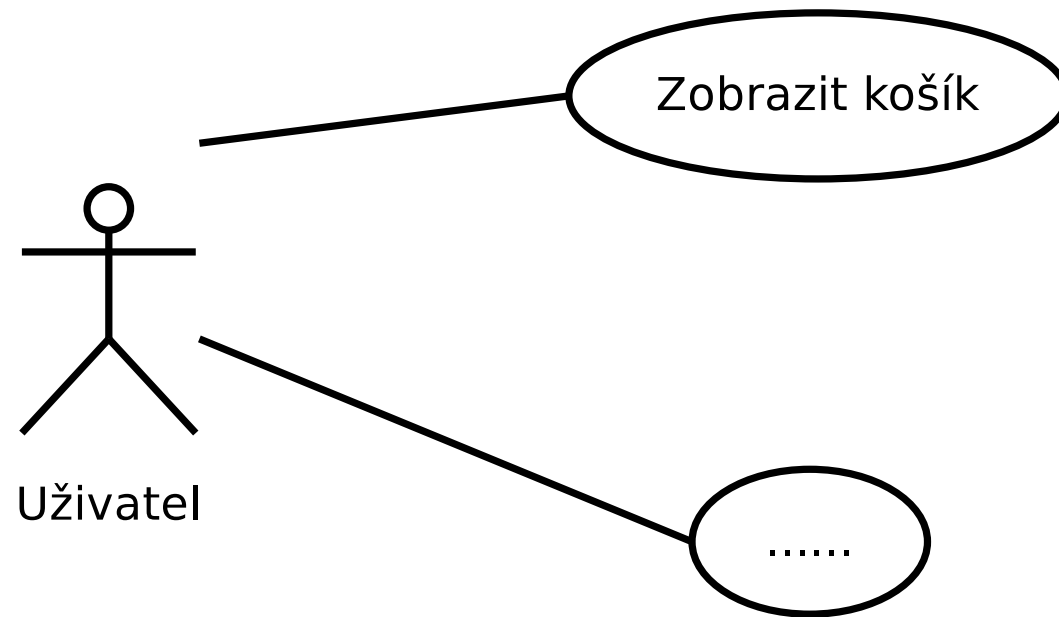
- znalost třídy (objektu), jak provést určitou operaci, která může být obecně společná pro více tříd (objektů)
- stejná operace s jedním názvem může mít více implementací

trida i presto ze je jineho typu ma schopnost provest operaci  
ktera je vseobecna pro vice tird



# Objektově orientované modelování – Příklad

## Diagram případů užití



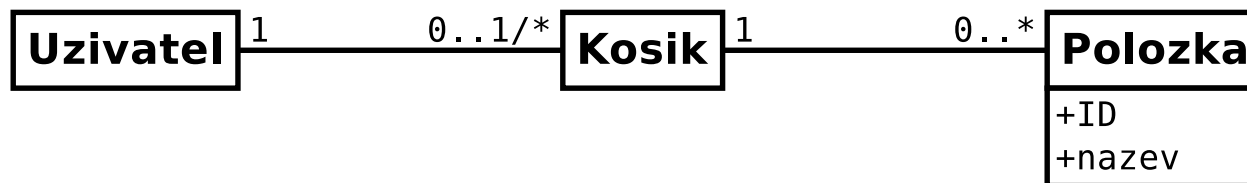
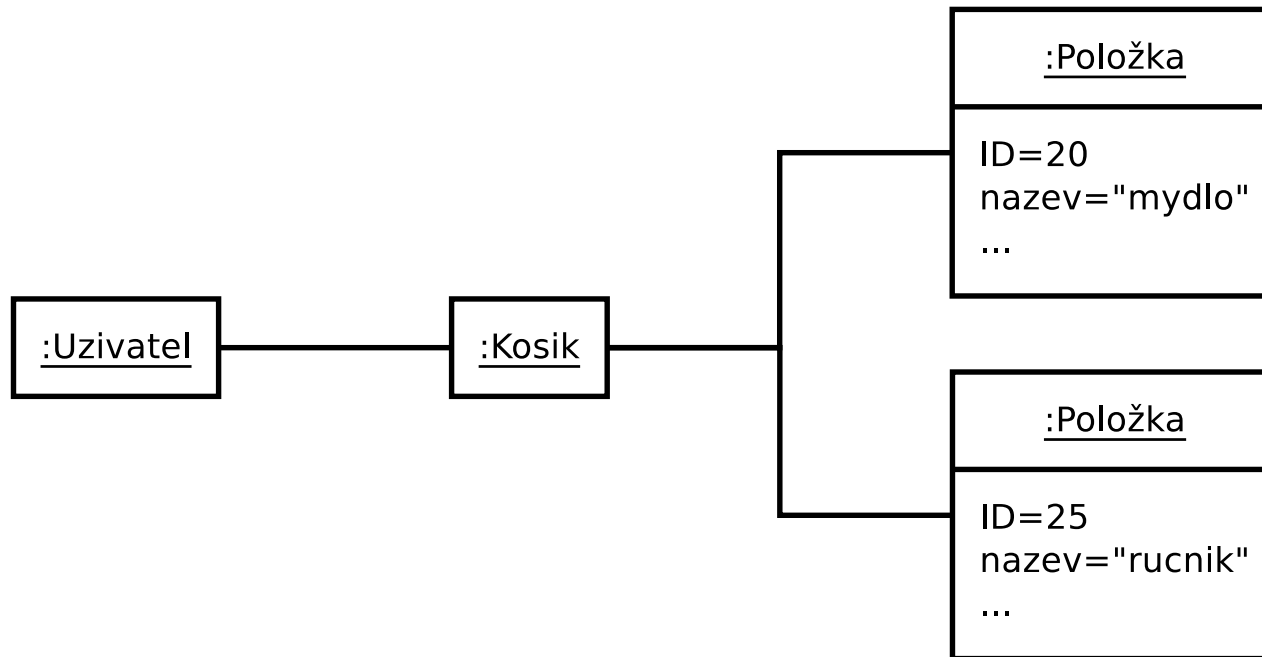
# Objektově orientované modelování – Příklad

## Specifikace případu užití

Případ užití: Zobrazit košík
<b>ID: UC11</b>
<b>Účastníci:</b> Zákazník
<b>Vstupní podmínky:</b> 1. Zákazník je přihlášen do systému.
<b>Tok událostí:</b> 1. Případ užití začíná volbou „zobrazit obsah košíku“. 2. <b>KDYŽ</b> je košík prázdný: 2.1 Systém oznámí Zákazníkovi, že košík neobsahuje žádné položky. 2.2 Případ užití končí. 3. Systém zobrazí seznam všech položek v nákupním košíku zákazníka včetně ID, názvu, množství a ceny každé položky.
<b>Následné podmínky:</b>  ...

# Objektově orientované modelování – Příklad

## Diagram objektů / tříd

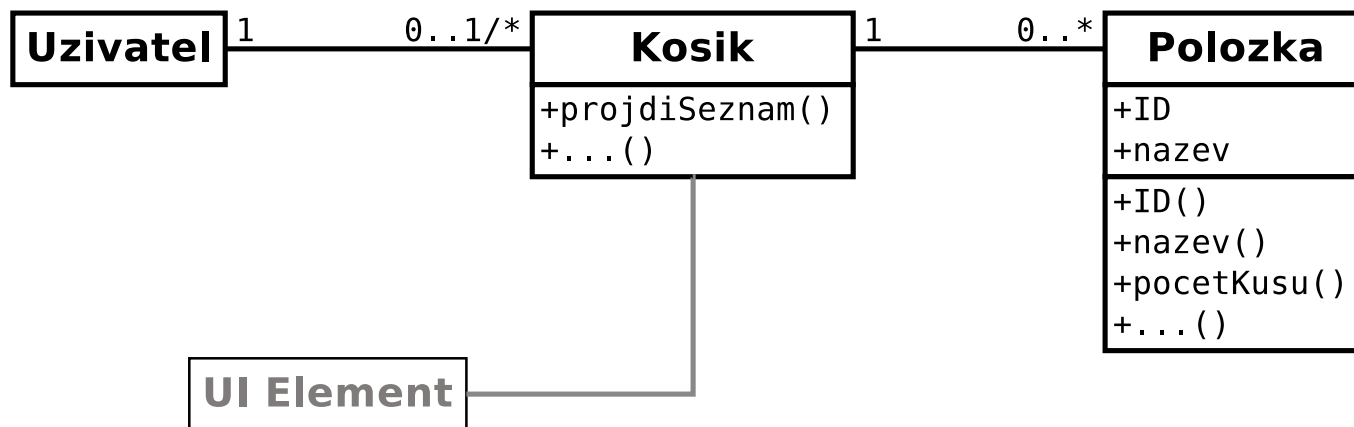


# Objektově orientované modelování – Příklad

snazíme se v systému vypíchnout věci které jsou pro systém a pro nás důležité  
a tím že jsou pro nás důležité tak i vztahy mezi nimi jsou pro nás důležité

## Doménový model

- zachycuje konceptuální elementy (koncepty/prvky/objekty) doménového systému
- cíl = najít a pojmenovat význačné prvky systému a vztahy mezi nimi
- doménový model  $\neq$  datový model



# Objektově orientované modelování v UML

## Jazyk UML

- Unified Modelling Language
- inspirován existujícími analytickými jazyky a modely  
výběr nejlepších myšlenek
- základní modelovací jazyk metodiky RUP – Rational Unified Process  
(první návrhy vytvořeny společně)

## Vývoj jazyka UML

- 1994: Booch a Rumbaugh; Rational Software Corp.  
(metodiky Booch a OMT – Object-Modeling Technique)
- 1995: Jacobson; Rational Software Corp.  
(metodika OOSE – Object-Oriented Software Engineering)
- 1997: UML 1.1, akceptován jako průmyslový standard  
(OMG – Object Management Group)
- 2005: UML 2.0
- 2017: UML 2.5.1 (<https://www.omg.org/spec/UML/>)

# Objektově orientované modelování v UML

Standard OMG UML 2.x obsahuje

- **popis diagramů a jejich použití**
- metamodel (MOF – Meta-Object Facility) specifikuje (modeluje) elementy diagramů UML
- **jazyk pro specifikaci omezení a podmínek OCL – Object Constraint Language**
- popis struktur pro výměnu modelů mezi nástroji



# Stavební bloky jazyka UML

## Předměty (*Things*)

- samostatné prvky modelu
- např. *třída, případ užití, stav, poznámky (anotace)*

## Vztahy (*Relationships*)

- určují vzájemnou souvislost předmětů
- např. *závislost, asociace, agregace, kompozice, zobecnění, realizace*

## Diagramy (*Diagrams*)

- pohledy na modely UML; kolekce předmětů a vztahů
- *analytické diagramy* – co bude systém dělat
- *návrhové diagramy* – jak to bude systém dělat
- např. *use case diagram, diagram tříd* umožňuje nám lépe zobrazit co bude systém dělat

# Vybrané společné mechanismy jazyka UML

## Ornamenty (*Adornments*)

- každý prvek modelu má svůj symbol (např. třída), který může být obohacen různými *ornamenty* (např. atributy, operace)
- obvykle není potřeba vždy zobrazovat všechny podrobnosti, některé *ornamenty* mohou být skryty (různé pohledy na systém)

obohacuje prvek systému o ornamenty, které dávají prvku něco navíc a specifického pro ten daný prvek

I

# Vybrané společné mechanismy jazyka UML

## Mechanismy rozšiřitelnosti

- omezení (*constraints*)
  - definují pravidla, která musí být vyhodnocena jako pravdivá
  - textový řetězec uzavřený do složených závorek {}
  - jazyk OCL (*Object Constraint Language*)
- stereotypy (*stereotypes*)
  - definuje nový prvek, který je založen na existujícím prvku
  - název stereotypu se většinou uzavírá do dvojitých závorek << *nazev* >>
  - musí se definovat sémantika nového prvku – podpora CASE nástroje, textová dokumentace, metamodel UML, ...

# Diagramy jazyka UML 2.5

## Diagramy struktury

- Diagram tříd (*Class Diagram*)
- Diagram objektů (*Object Diagram*)
- Diagram seskupení (balíčků) (*Package Diagram*)
- Diagram vnitřní struktury (*Composite Structure Diagram*)
- Diagram komponent (*Component Diagram*)
- Diagram rozmístění zdrojů (nasazení) (*Deployment Diagram*)
- *Profile Diagram* – popisuje rozšiřující mechanismy
- *další diagramy, které nejsou součástí oficiálního standardu*

# Diagramy jazyka UML 2.5

## Diagramy chování

- Diagram případů užití (*Use Case Diagram*)
- Diagram aktivit (*Activity Diagram*)
- Stavový diagram (*State Machine Diagram*)
- *další diagramy, které nejsou součástí oficiálního standardu*

## Diagramy interakce

- Sekvenční diagram (*Sequence Diagram*)
- Diagram komunikace (*Communication Diagram*)
  - původní diagram spolupráce (*Collaboration Diagram*) z UML 1.x
- Diagram přehledu interakcí (*Interaction Overview Diagram*)
- Diagram časování (*Timing Diagram*)

# Diagram tříd

## Diagram tříd

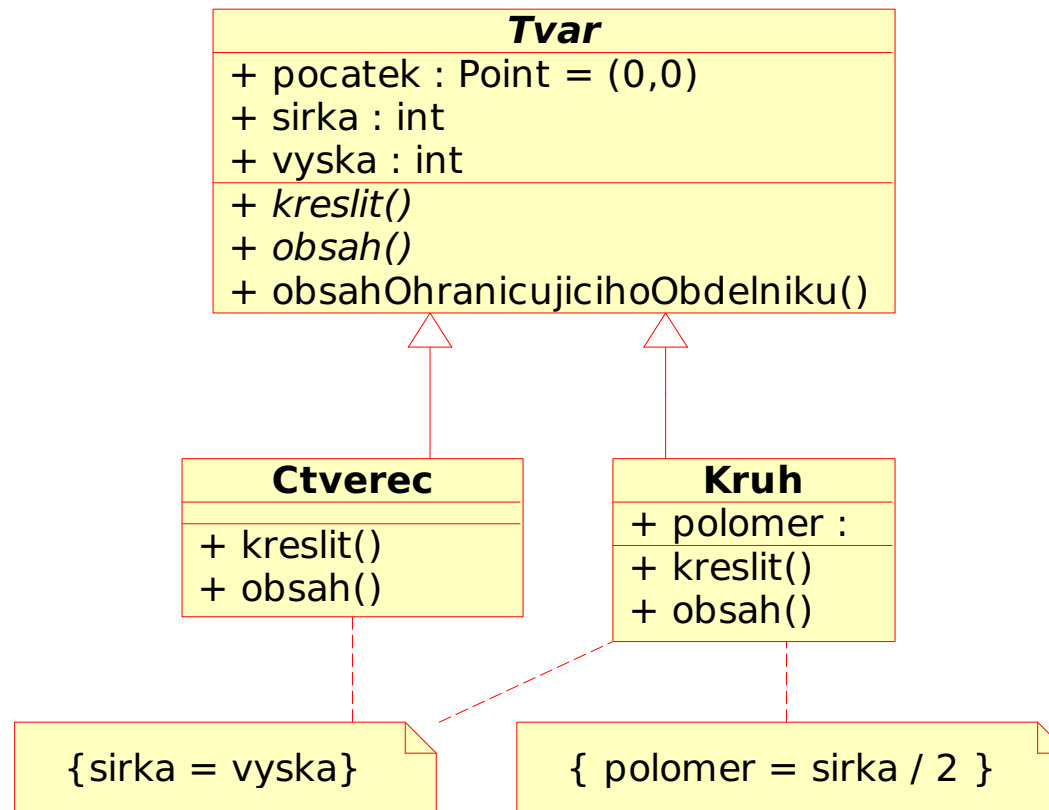
- zobrazuje třídy a statické vztahy mezi nimi

## Vztahy mezi třídami

- zobecnění (generalization)
- asociace (association)
- závislost (dependency)
- realizace (realization)

# Dědičnost tříd

- vztah generalizace/specializace mezi třídami
- odvozená třída sdílí atributy, chování, vztahy a omezení obecnější třídy
- odvozená třída může přidávat a modifikovat atributy a chování



# Dědičnost tříd

## Operace vs. metoda

- operace reprezentuje abstraktní pohled na chování objektu
- metoda implementuje operaci
- *signatura operace* = název, typ návratové hodnoty, typy všech stejně seřazených argumentů

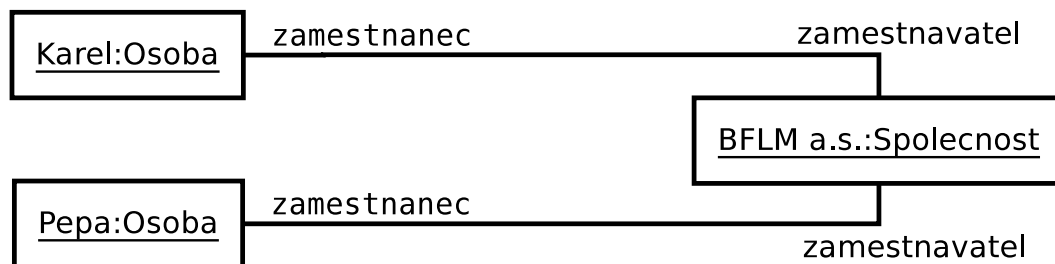
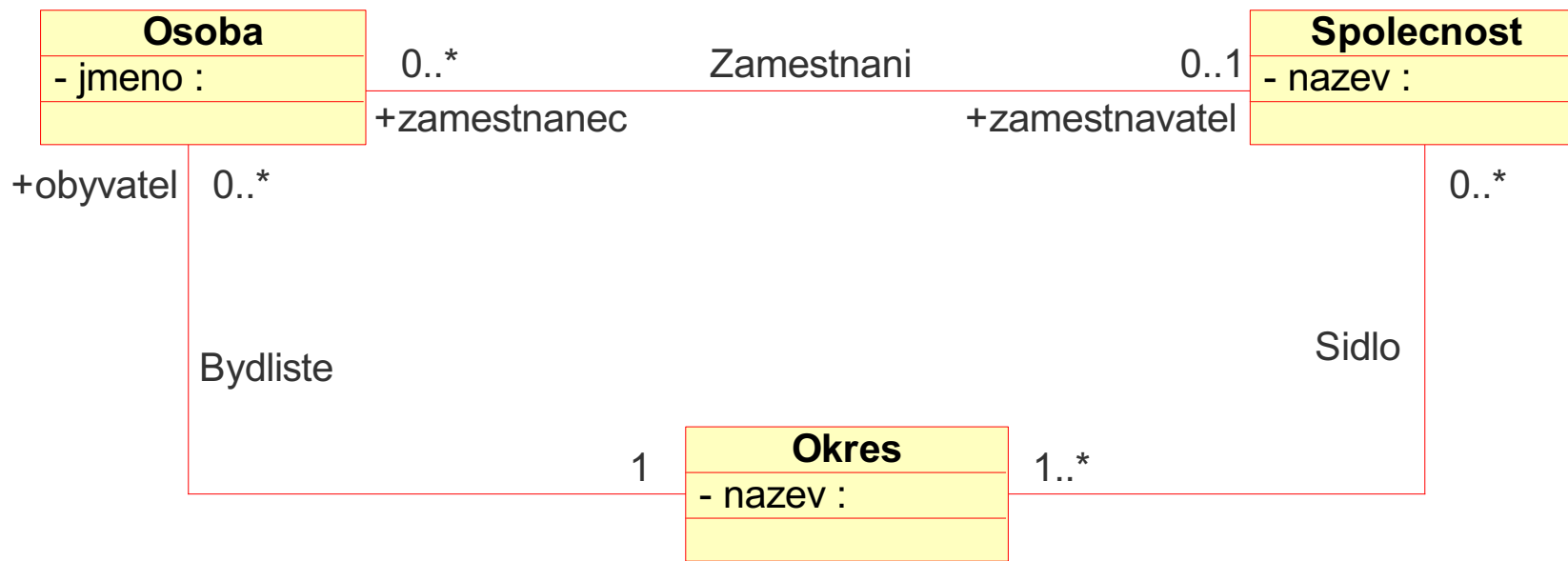
## Abstraktní operace a třídy

- odložení implementace operace na potomky
- abstraktní třída deklaruje všechny operace, ale některé ponechává bez implementace
- např. třída *Tvar* a operace *kreslit* a *obsah*



# Asociace

Asociace slouží k zachycení vztahů mezi třídami (jejich instancemi).

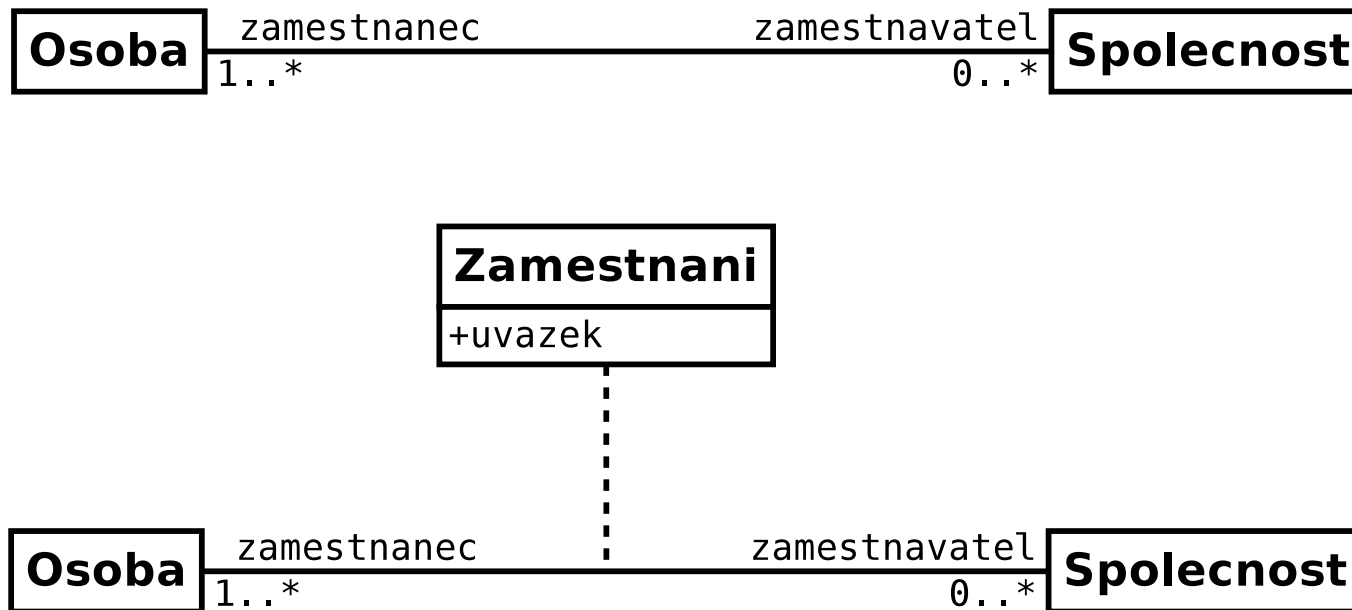


# Vlastnosti asociace

- objekt má ve vztahu svou roli
- asociace má své násobnosti (mohutnosti)
  - násobnost je odrazem cíle modelu  
bez této znalosti nelze určit špatnou/dobrou násobnost
- asociace má svůj název
  - název může být sloveso nebo podstatné jméno
  - Zaměstnání;  $\Rightarrow$  je zaměstnán v ;  $\Leftarrow$  zaměstnává
  - *v případě slovesa se často označuje směr vazby*
- vyjadřuje proměnlivý vztah mezi objekty (instancemi tříd)
  - každé spojení váže instanci jedné třídy s instancí druhé třídy
  - počet spojení se v čase může měnit
  - v OO návrhu lze asociaci povýšit na třídu (asociační třída)

# Asociační třída

- přiřazení atributů asociaci
- asociace *Zamestnani*, atribut *uvazek*



# Asociace vyššího stupně

- binární asociace (vztah dvou tříd, resp. jejich instancí)
- N-ární asociace (vztah více tříd, resp. jejich instancí)
  - jsou méně časté,
  - většinou se dají převést na binární asociace,
  - pokud ne, bývá nutné povýšit asociaci na třídu.

# Asociace celek/část – Agregace

## Agregace (Seskupení)

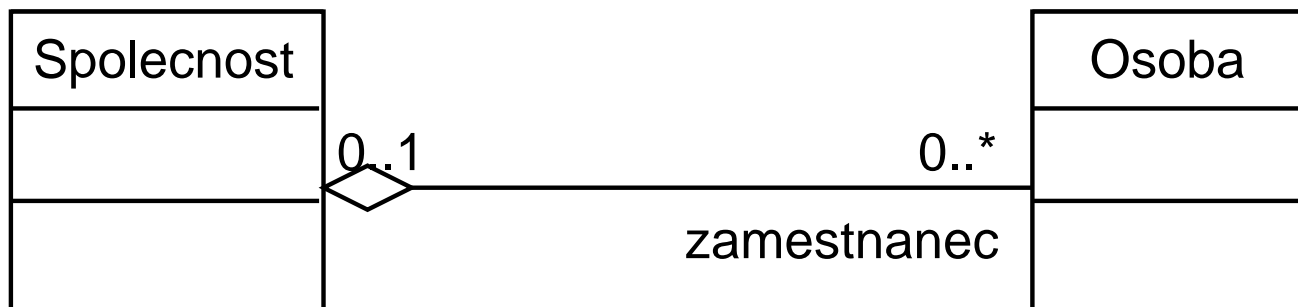
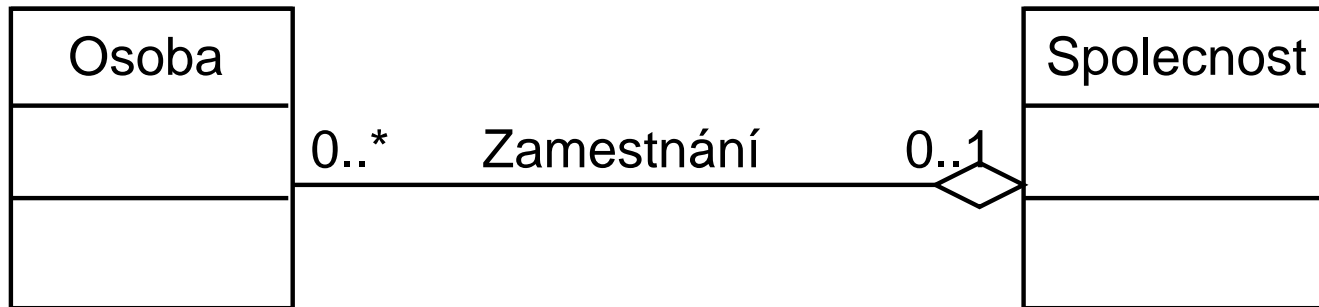
- celek je seskupen z více částí
- celek = agregační (seskupený) objekt
- část celku = konstituční (tvořící) objekt

objekt může být součástí více celku a může také existovat samostatně např. je Celek auto a objekt je kolo a kolo může existovat bez auta

## Vlastnosti agregace

- seskupený objekt může existovat bez svých konstitučních objektů
- konstituent (konstituční objekt) může být součástí více seskupení
- implicitní násobnost se nedá předpokládat
- asociace agregace nemá název (vyjadřuje vztah *má*)
- agregace bývají homeometrické (tj. konstituenti patří do téže třídy)

# Asociace celek/část – Agregace



# Asociace celek/část – Kompozice

## Kompozice (Složení)

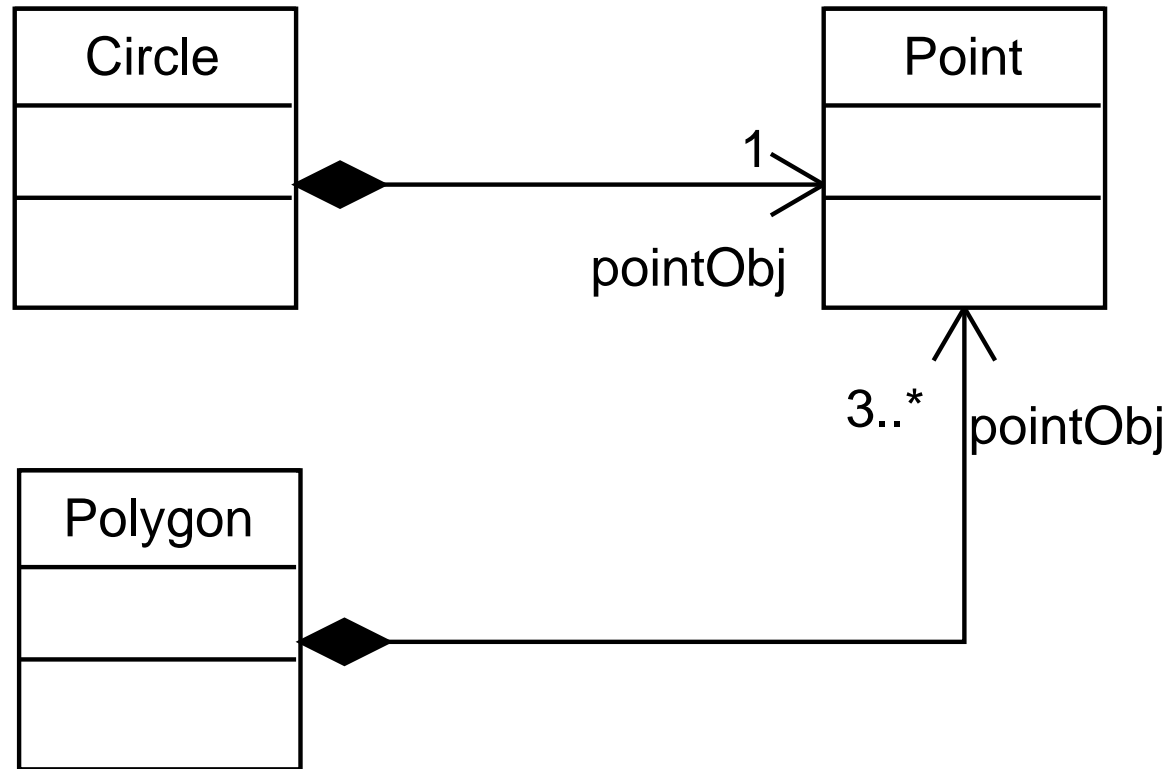
mnohem pristejsi cast celku neboli objekt nemuze existovat mimo celek

- celek je složen z více částí
- celek = kompozitní (složený) objekt
- část celku = komponentní (složkový) objekt

## Vlastnosti kompozice

- **složený objekt (většinou) neexistuje bez svých komponent**
- **komponenta (komponentní objekt) může být součástí nejvýše jedné kompozice**
- implicitní násobnost každé složky je 1
- asociace kompozice nemá název
- kompozice bývají heterometrické (tj. komponenty patří do různých tříd)
- při rušení celku se ruší jeho složky (příp. jsou z kompozice vyňaty)

# Asociace celek/část – Kompozice





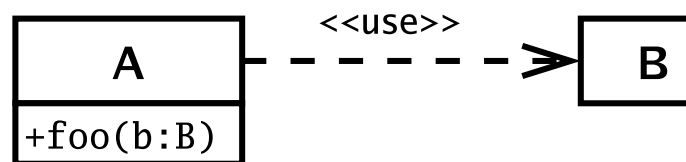
# Diagram tříd – Závislost

## Závislost

- vyjadřuje jiné různé vztahy mezi objekty či třídami
- typ závislosti se označuje pomocí *stereotypů*

Nejběžnější typ stereotypu – *používání* <<use>>

- $A$  (klient)  $\rightarrow$   $B$  (dodavatel)
  - metoda třídy  $A$  potřebuje argument třídy  $B$
  - metoda třídy  $A$  vrací hodnotu třídy  $B$
  - metoda třídy  $A$  používá objekt třídy  $B$ , ale *ne jako atribut*



- závislost bez uvedeného stereotypu se považuje za používání

# Diagram tříd – Typy závislostí (stereotypy)

- <<instantiate>> / <<create>>
  - klient vytváří instance dodavatele
- <<trace>>
  - klient realizuje dodavatele  
vazba mezi elementem v různých modelech
- <<refine>>
  - klientská třída poskytuje detailnější informace než dodavatel
- <<send>>
  - operace klienta zasílá signál příjemci
- <<call>>
  - klientská třída volá operaci dodavatele
- ...

# Diagram tříd – Realizace

## Rozhraní

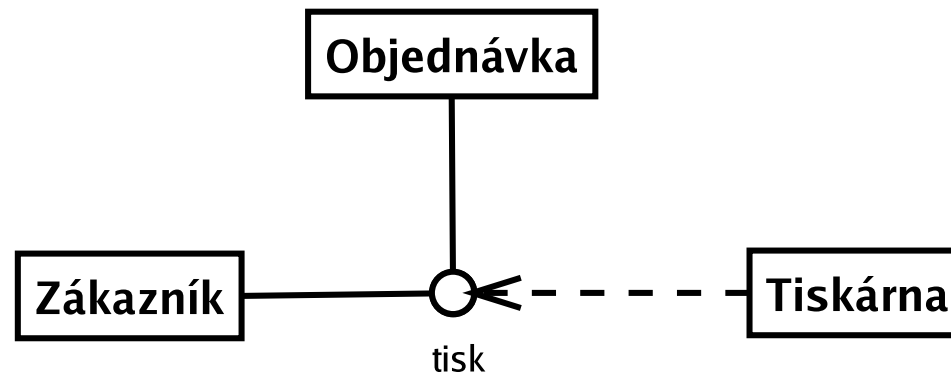
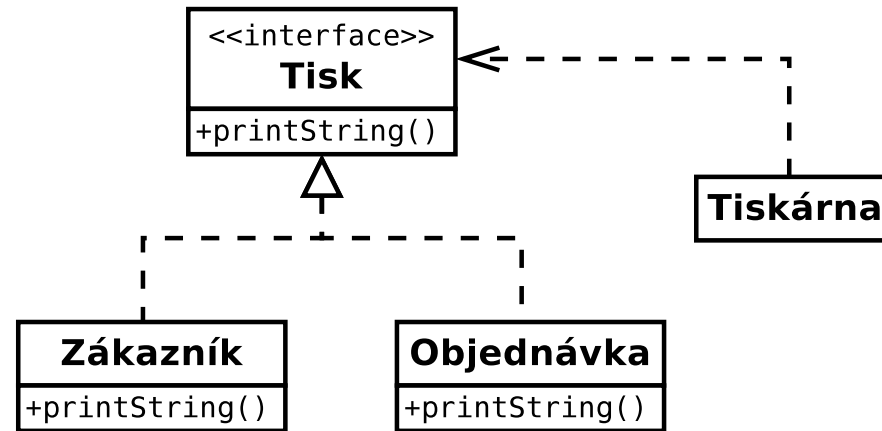
- množina operací, které určují chování objektu
  - pouze definuje, co objekt umí (nabízí), nedefinuje *jak*
  - způsob provedení operací závisí na jejich implementaci třídami
- lze modelovat jako speciální prvek
- zjednodušení návrhu, omezuje počet vazeb

rozhraní nabízí objektu vybrat operaci a nechává na samotném objektu jeho implementaci

## Realizace

- vztah mezi třídou a rozhraním
- třída implementuje všechny operace (metody) z daného rozhraní
- třída používající rozhraní pak umí používat i jeho implementační třídy

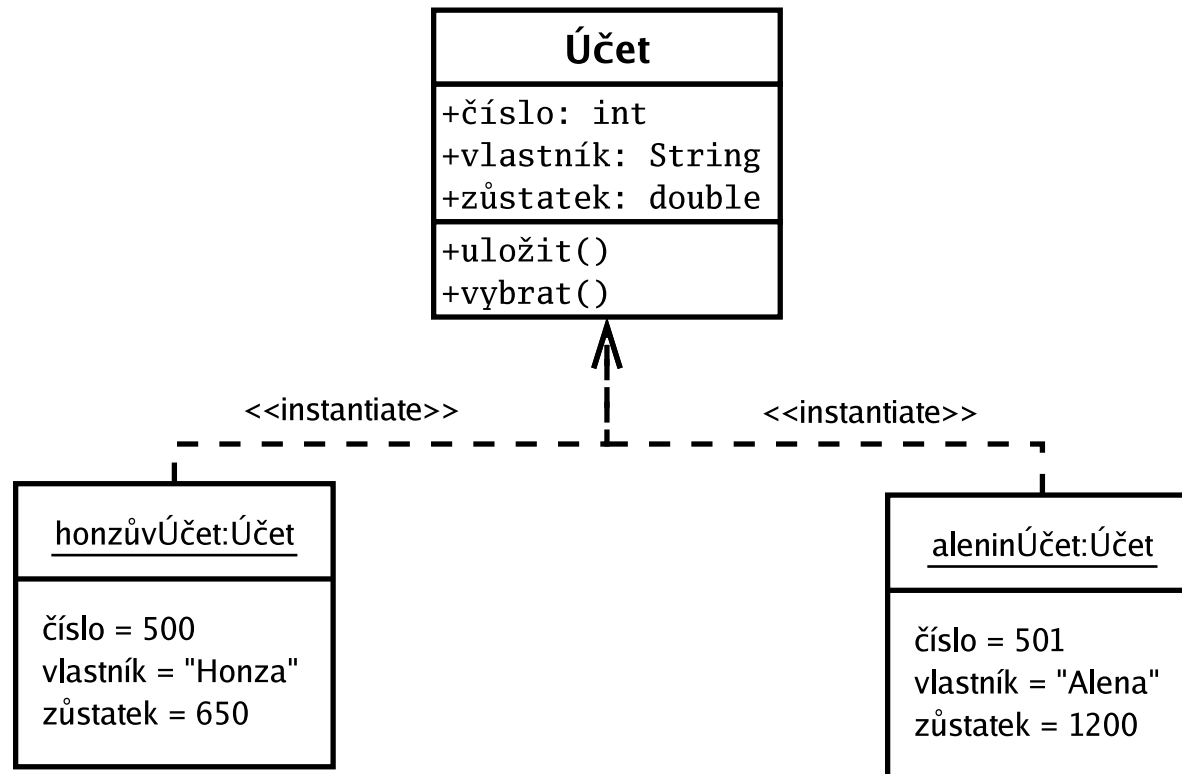
# Diagram tříd – Realizace



# Diagram objektů

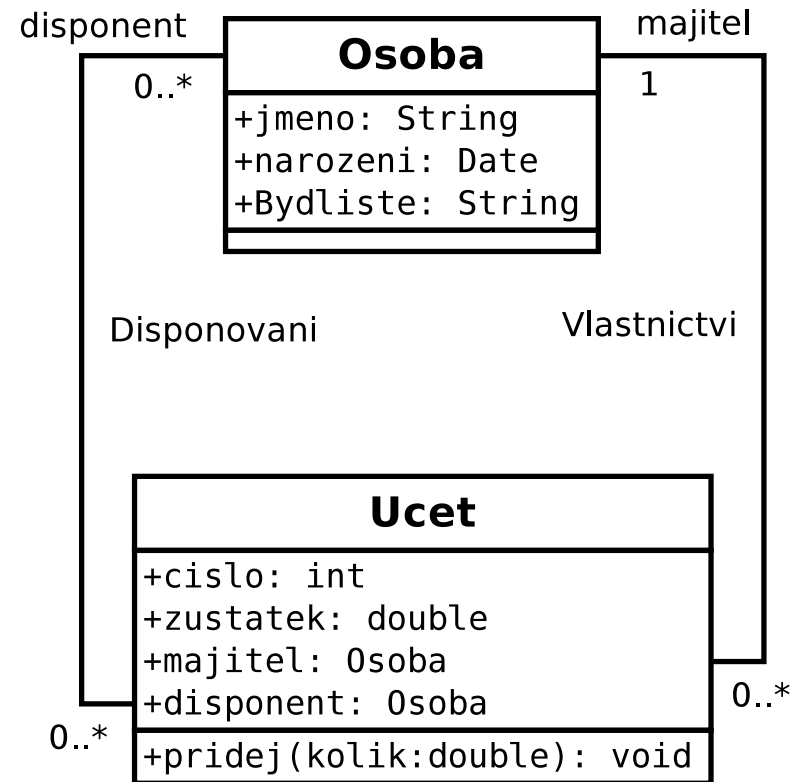
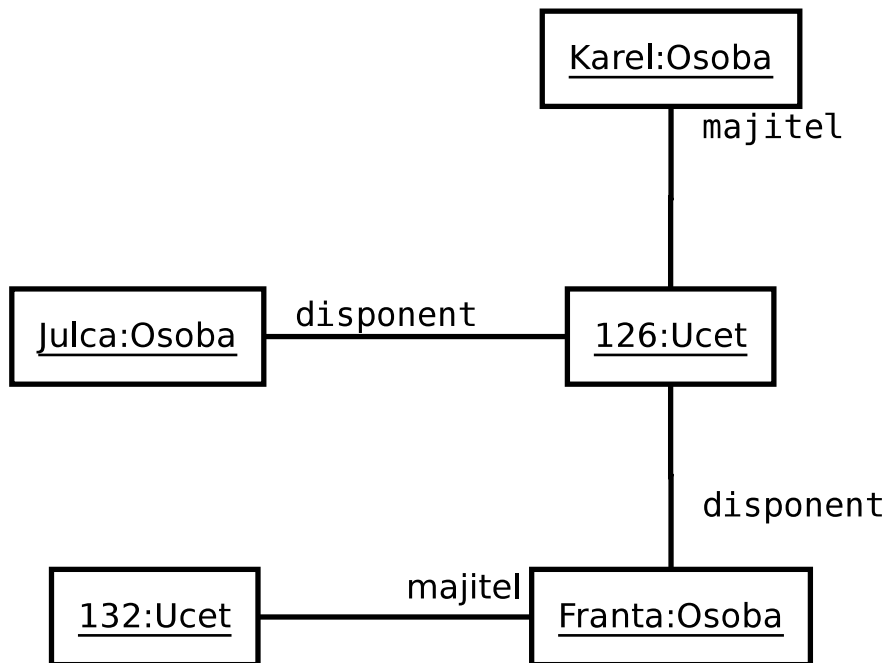
## Diagram objektů (Object Diagram)

- je úzce svázan s diagramem tříd
- znázorňuje objekty a jejich relace v určitém čase
- relace jsou dynamické (nemusí trvat po celou dobu existence objektů)



# Diagram objektů

- mezi objekty existuje *spojení* (*spojení* = instance vztahu *asociace*)



# Diagram seskupení

## Diagram seskupení (balíčků) (Package Diagram)

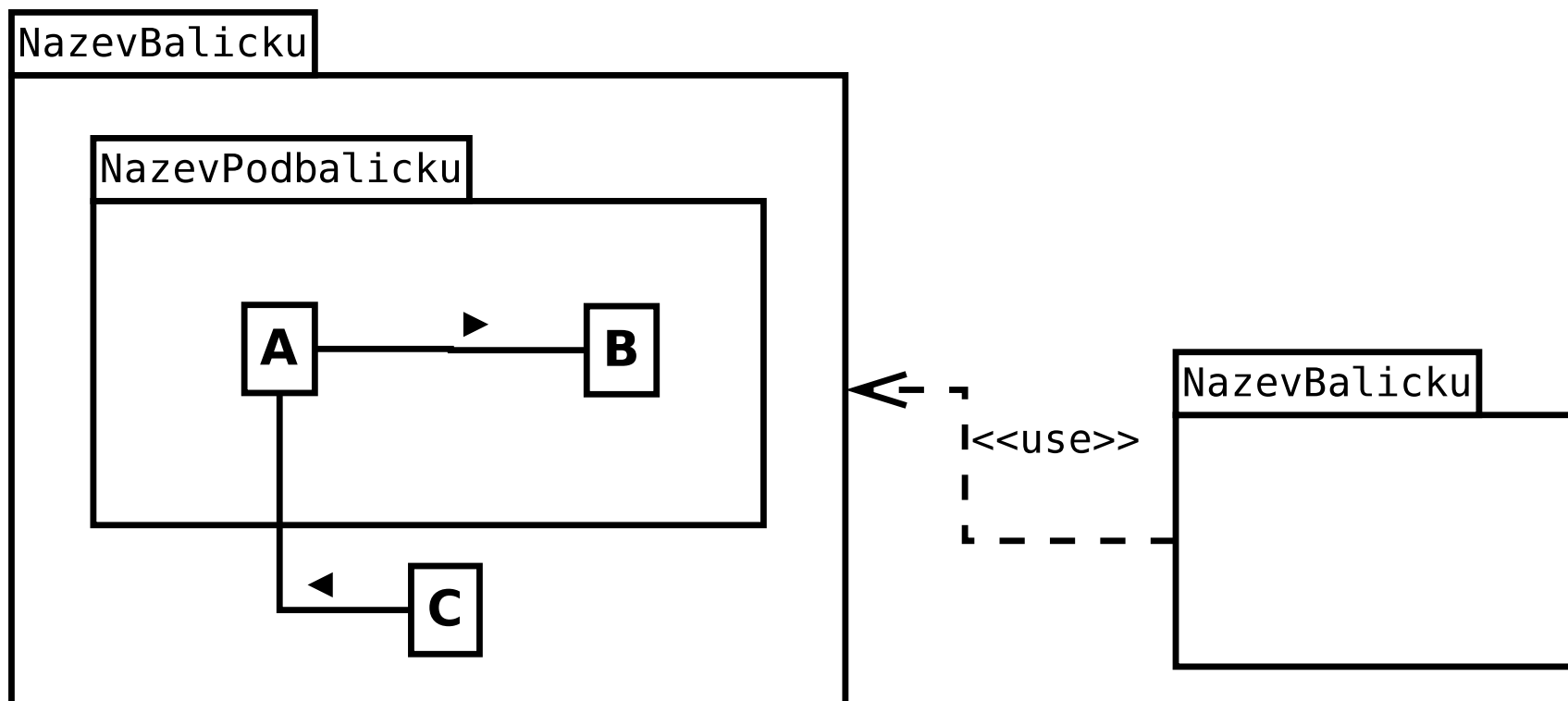
Pohled zhora na systém který je rozdělen na jednotlivé části

- seskupení sémanticky souvisejících elementů
- definuje sémantické hranice modelu
- umožňují souběžnou práci v etapě návrhu
- poskytují zapouzdření prostoru jmen

Balíčky mohou obsahovat

- případy užití
- analytické třídy
- realizace případů užití
- další balíčky

# Diagram seskupení





# Principy objektově orientovaného návrhu

## Problémy spojené se špatným návrhem

- změna v softwaru je náročná a vyžaduje úpravy na mnoha místech
- změna způsobí problémy v jiných, mnohdy nesouvisejících částech softwaru
- vyčlenit část softwaru pro znovupoužitelnost je náročnější než tuto část vytvořit znovu

Vytvarito solidní jungli

## Principy objektově orientovaného návrhu architektury

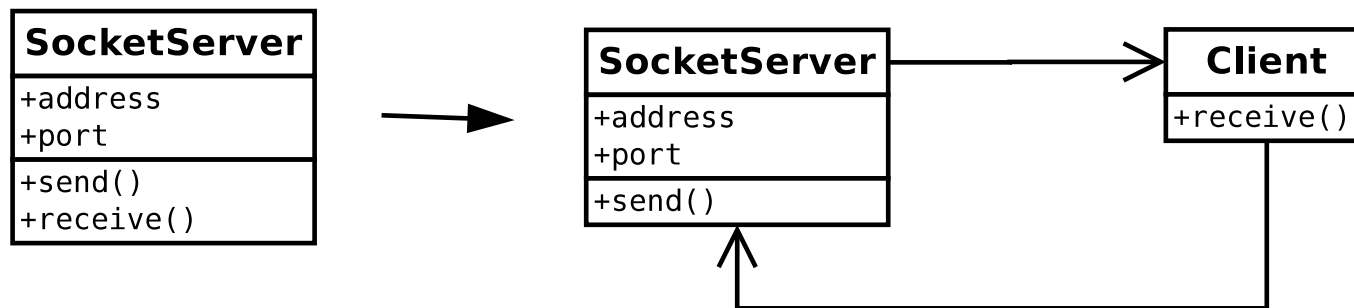
- předkládají vhodné postupy pro návrh architektury
- vhodné z pohledu údržby a rozšiřitelnosti architektury systému

# Principy objektově orientovaného návrhu

## Single Responsibility Principle (SRP)

- *třídy by měly mít jedinou zodpovědnost; jediný důvod ke změně*
- **Zodpovědnost (responsibility)**
  - závazek nebo povinnost prvku něco *dělat* nebo něco *vědět*
  - akce/znalost může prvek dělat/mít přímo nebo využívat jiné prvky (koordinace činností, agregace dat, ...)
  - zodpovědnost  $\neq$  metoda; metody jsou implementovány, aby byla splněna zodpovědnost

kazda trida ma jeden smysl a jeden ucel

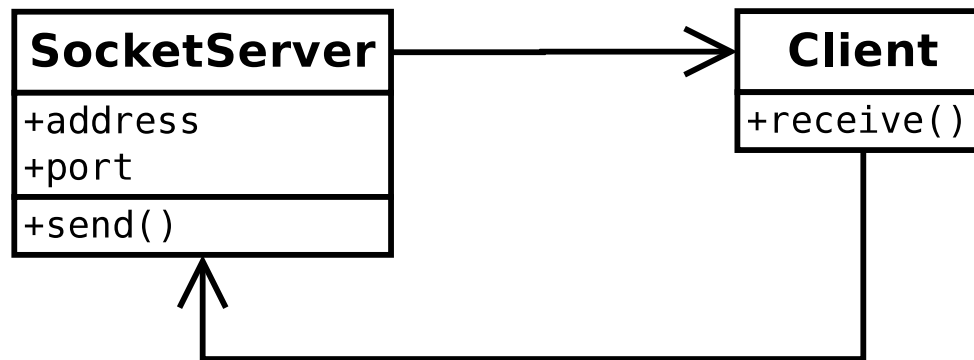


# Principy objektově orientovaného návrhu

## Open Closed Principle (OCP)

- *třída by měla být otevřená pro rozšíření ale uzavřená pro modifikace*
- *třída by měla být rozšiřitelná bez nutnosti modifikace kódu*

ideál toho že třída pokud chceme něco přidat je možné to udělat bez změny už existujícího kódu



## Problém

- chceme *SocketServer* použít i pro jiné klienty
- modifikace třídy *SocketServer* ⇒ **OCP !**

# Principy objektově orientovaného návrhu

## Dependency Inversion Principle (DIP)

- *závislost na abstraktním ne na konkrétním*
- závislosti by měly směřovat jedním směrem, a to od konkrétního k abstraktnímu
- závislosti by měly směřovat ke společným rozhraním a abstraktním třídám

Moduly zavisi na abstrakci místo na implementaci

## Důsledky

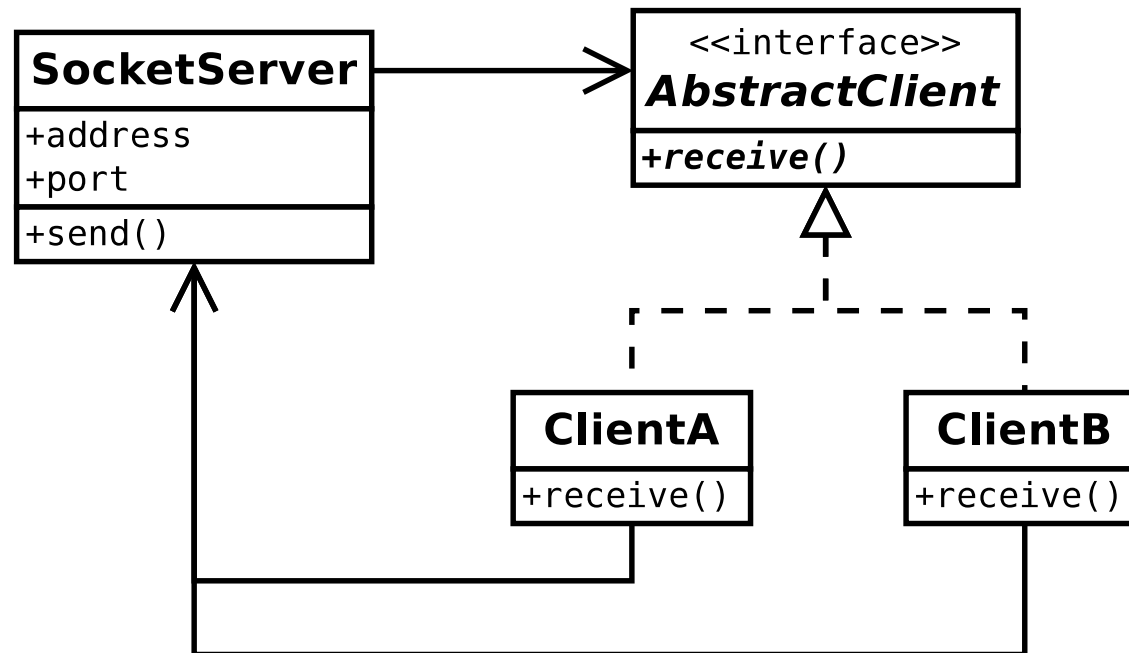
snaha o menší závislost modulu na sobě

- redukce závislosti v kódu
- abstraktní rozhraní se mění mnohem méně než konkrétní implementace  $\Rightarrow$  závislý kód se nemusí měnit tak často
- snadná možnost nahradit jednu implementaci za jinou

# Principy objektově orientovaného návrhu

## Problém

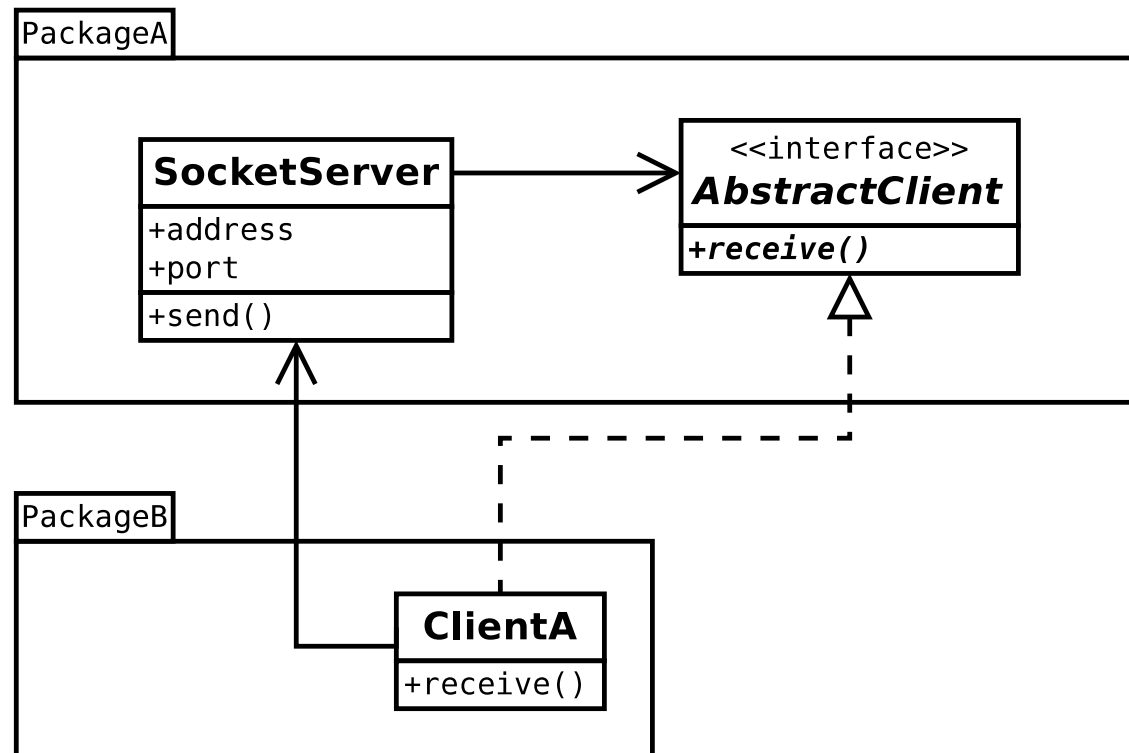
- chceme *SocketServer* použít i pro jiné klienty
- modifikace třídy *SocketServer*  $\Rightarrow$  **OCP !**
- aplikujeme *DIP*



# Principy objektově orientovaného návrhu

## Balíčky

- třídy a rozhraní řešící komunikaci dáme do jednoho balíčku
- třídy obsluhující klienta patří do jiného balíčku



# Principy objektově orientovaného návrhu

## Principy návrhu balíčků (komponent)

- Release Reuse Equivalency Principle (REP)
  - *granularita znovupoužitelnosti je shodná s granularitou uvolnění nové verze*  
to ze nové verze jednotlivých komponent mají být kompatibilní se stávajícím systémem  
zachována zpětná kompatibilita
- Common Closure Principle (CCP)
  - *třídy, které se mění společně, patří k sobě*
- Common Reuse Principle (CRP)
  - *třídy, které nejsou znovupoužívány společně, by neměly patřit k sobě*

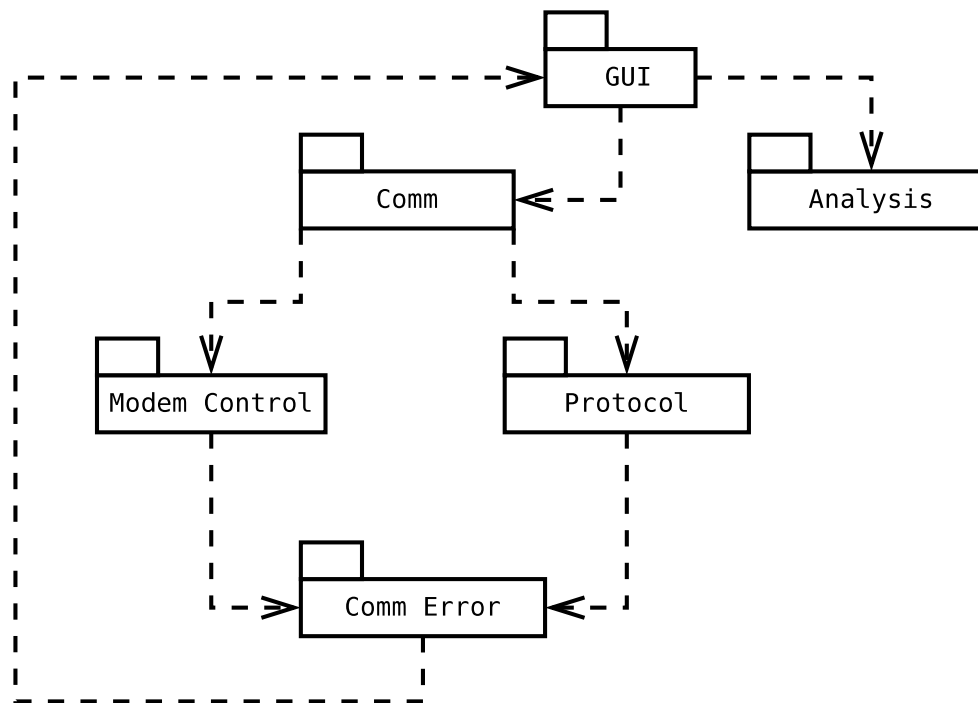
Nelze dodržet všechny principy

- REP a CRP usnadňují vývoj s využitím znovupoužitelnosti
- CCP usnadňuje práci při údržbě

# Principy objektově orientovaného návrhu

## Acyclic Dependencies Principle (ADP)

- **závislosti mezi balíky nesmí tvořit cykly**
- minimální počet závislostí mezi balíky  $\Rightarrow$  jednodušší údržba a uvolňování nových verzí (menší počet závislých balíků pro testování)
- závislosti s cykly  $\Rightarrow$  velký počet závislých balíků

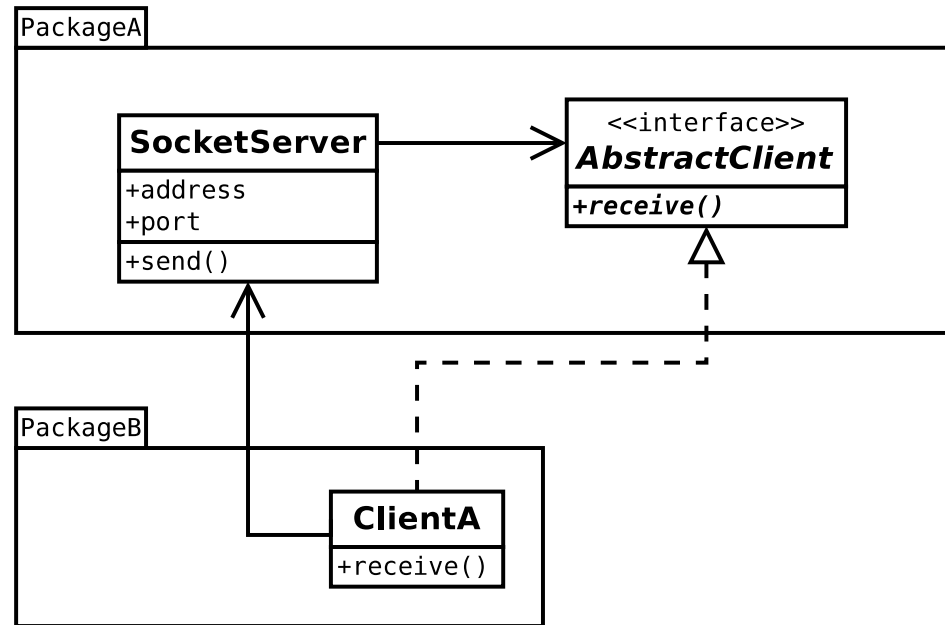
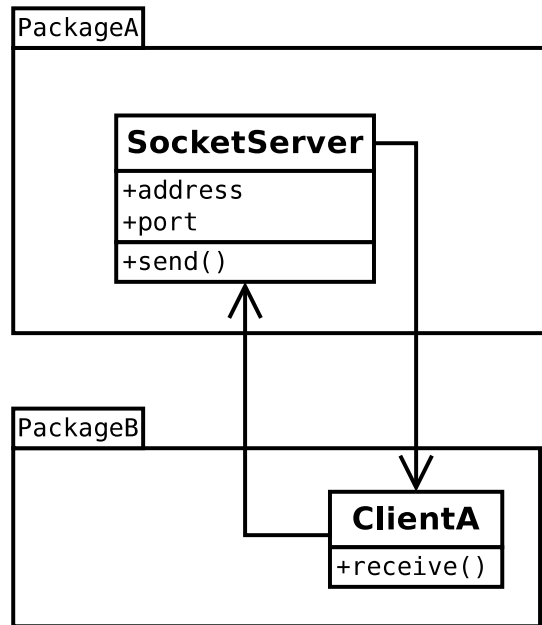




# Principy objektově orientovaného návrhu

## Balíčky

- aplikací DIP jsme odstranili cyklickou závislost (ADP)



# Principy objektově orientovaného návrhu

## Rozhraní versus implementace

- *signatura*
  - deklaruje formální podobu operace (název, typy, ...)
  - lze zkontrolovat překladačem
- *kontrakt*
  - deklaruje sémantiku operace a její podmínky (preconditions, ...)
  - nelze zkontrolovat překladačem
- *implementace*
  - realizuje operace definované signaturami a kontrakty
  - implementace by se měla skrývat

# Principy objektově orientovaného návrhu

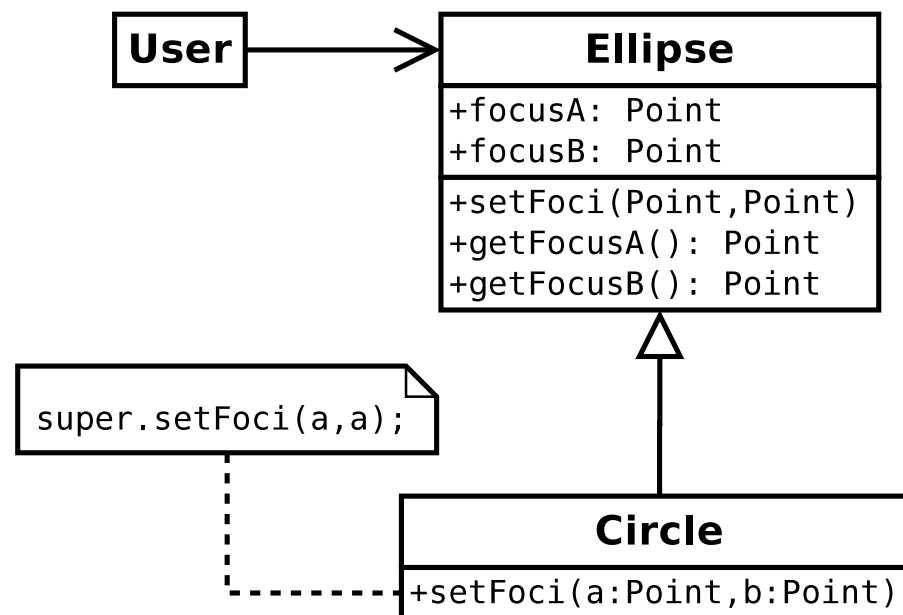
## Liskov Substitution Principle (LSP)

- *odvozené třídy by měly být zaměnitelné za базové třídy*
- uživatelé базové třídy by měli být schopni pokračovat bez chybného chování i při nahrazení odvozenou třídou

### Příklad kružnice / elipsa

- kružnice je elipsa se stejnými ohnisky

vice mene to znamena ze potrida musi dodrzovat pravidla nadtridy a zaroven ji rozsirovat?



# Principy objektově orientovaného návrhu

LSP – Příklad kružnice / elipsa

- kružnice je konzistentní sama se sebou
- elipsa je konzistentní sama se sebou
- tyto objekty jsou však používány různými třídami
- *předpoklad: klienti se snaží vše obejít či zničit*

```
Ellipse e = new Circle();
```

```
e.setFoci(a, b);
```

```
assert e.getFocusA() == a;
```

```
assert e.getFocusB() == b;
```

# Principy objektově orientovaného návrhu

## LSP – Příklad kružnice / elipsa

- kružnice není zastupitelná za báзовou třídu
- kružnice porušuje *kontrakt* elipsy

## Kontrakt

- kontraktem se rozumí podmínky definované rozhraním
- rozhraní definuje precondition a postcondition
- odvozené třídy musí kontrakt dodržet

## Řešení

- nemodifikovatelný objekt  
bez operace *setFoci(Point, Point)*, pouze konstruktory
- kružnice je elipsa se stejnými ohnisky  
⇒ není nutné vytvářet třídu pro kružnici

# Principy objektově orientovaného návrhu

## Do not Repeat Yourself (DRY)

- *neopakujte stejný kód na různých místech*
- problémy s modifikací a udržitelností
- opakující se kód  $\Rightarrow$  samostatná metoda
- soubor opakujících se metod  $\Rightarrow$  vytvoření obecnější třídy

# Studijní koutek – Studium v zahraničí

*Získání nových poznatků, kontaktů, poznání odlišného prostředí,  
sblížení s cizím jazykem, ...*

## Program Erasmus+

- Studijní pobyt je součástí studia na české univerzitě.
- Výsledky studijního pobytu se uznávají – kreditový systém ECTS.
- Student v zahraničí má stejné podmínky jako místní student.
- Student získává grant ve formě stipendia jako příspěvek na náklady spojené s cestou a pobytem.
- Délka pobytu 90-360 dnů (v rámci jednoho ak. roku).
- Suma pobytů během jednoho stupně studia je nejvýše 360 dnů.
- FIT má nasmlouváno cca 200 míst na 76 univerzitách v 25 zemích: Francie, Německo, Španělsko, Řecko, Finsko, Velká Británie, ...

## Další informace a programy

<https://www.vut.cz/studenti/staze>

<https://www.fit.vut.cz/study/study-abroad/>