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)

- **Vlastnosti podstatné pro OO (důležité na státnici)**
  - **Abstrakce (Abstraction)**
    - vytvářený systém objektů je abstrakcí řešeného problému (zjednodušený pohled na systém bez ztráty jeho významu), vybíráme jen vlastnosti, které nás zajímají
    - analýza problému ⇒ klasifikace do abstraktních struktur
      - rozpoznávání podobností v řešené problematice
      - entity klient ⇒ entitní množina Klient
      - objekty klient ⇒ třída Klient
  - **Třídy objektů**
    - 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
  - Oproti relačnímu systému se tady řeší spíš běžící systém, oproti vypnutému
  - **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 detail˚

Strukturovaný přístup

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

Strukturovaný přístup

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

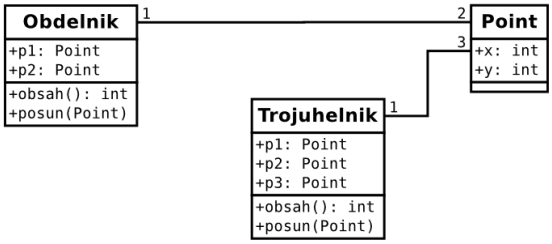
Objektový přístup

`class Obdelnik { int x; int y; int obsah() { return x * y; } }`	`class Trojuhelnik { int x; ... int obsah() { ... } }`	`class Obdelnik { Point p1; Point p2; int obsah() { ... } }`
`o.obsah()`	`t.obsah()`	`o.obsah()`

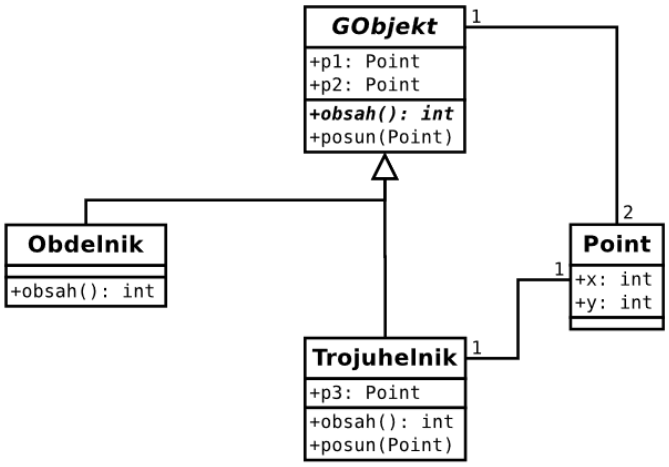
```
class Obdelnik {
    int x;
    int y;
    int obsah() {
        return x * y;
    }
}

o.obsah()
```

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

t.obsah()
```

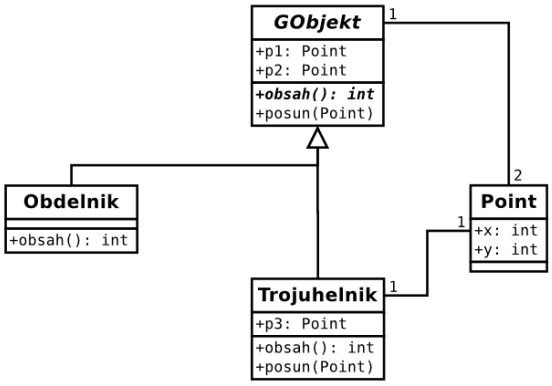
```
class Obdelnik {
    Point p1;
    Point p2;
    int obsah() {
        ...
    }
}

o.obsah()
```

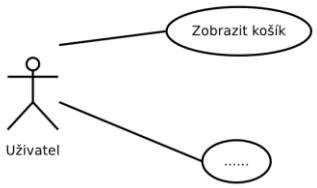


- **Dědičnost (Inheritance)**
  - definuje a vytváří třídy (objekty) na základě již existujících tříd (objektů)
    - možnost sdílení chování bez nutnosti reimplementace
    - možnost rozšíření chování
  - mezi třídami (objekty) vzniká hierarchický vztah podle dědičnosti (strom)



- **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 implementac´

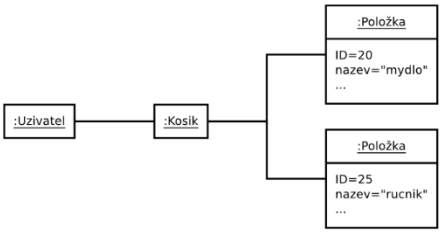


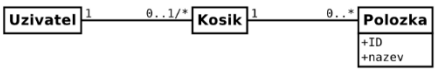
- **Příklad**
  - 

Specifikace případu užití

Případ užití: Zobrazit košík
ID: UC11
Účastníci: Zákazník
Vstupní podmínky: 1. Zákazník je přihlášen do systému.
Tok událostí: 1. Případ užití začíná volbou „zobrazit obsah košíku“. 2. KDYŽ 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.
Následné podmínky:
. . .

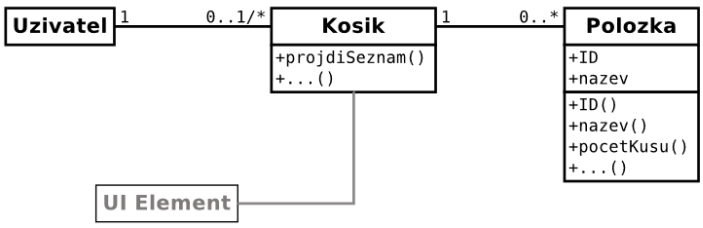
Diagram objektů / tříd





  - **Dvojtečka – instance třídy za dvojtečkou (název by byl před dvojtečkou)**
  - **+ znamená že jsou veřejné informace**
  - **– privátní**

  - **Doménový model**
    - zachycuje konceptuální elementy (koncepty/prvky/objekty) doménového systému
    - cíl = najít a pojmenovat význačné prvky systému a vztahy mezi nimi
    - doménový model ≠ 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/)

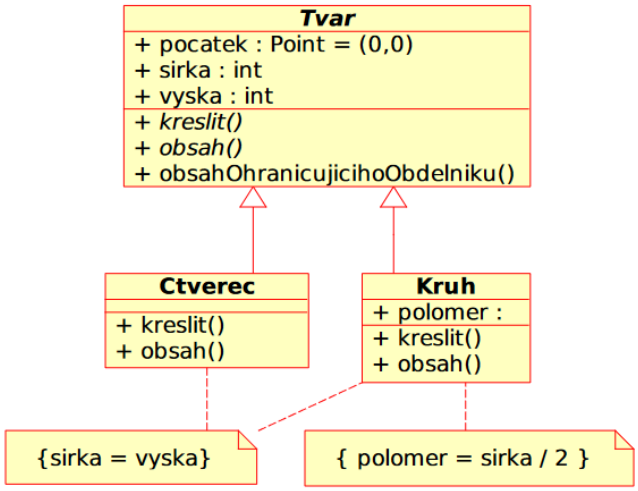
- Standard OMG UML 2.x obsahuje
  - **popis diagramů a jejich použití**
  - metamodel (MOF – Meta-Object Facility) specifikuje (modeluje) elementy diagramů UML
  - **jazyk pro specifikaci omezení a podmínek OCL – Object Constraint Language**
  - popis struktur pro výměnu modelů mezi nástroji

- 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
  - 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)
- 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 uzavíra do dvojitých závorek << nazev >>
    - musí se definovat sémantika nového prvku – podpora CASE nástroje, textová dokumentace, metamodel UML, . . .

- **Diagramy struktury**
  - **Diagram tříd (Class Diagram)**
  - **Diagram objektů (Object Diagram)**

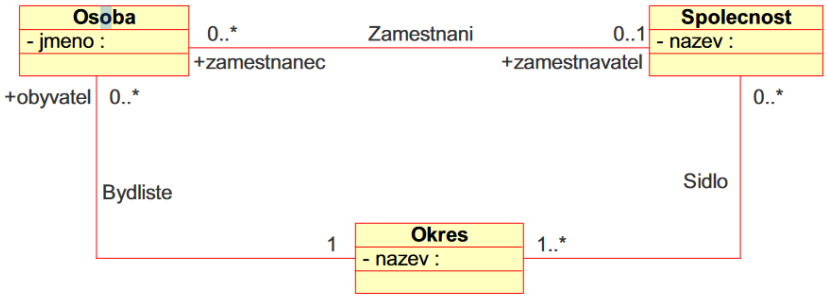
- Diagram seskupení (balíčků) (Package Diagram)
- Diagram vnitřní struktury (Composite Structure Diagram)
- Diagram komponent (Component Diagram)
- Diagram rozmístění zdrojů (nasazení) (Deployment Diagram)
- Profile Diagram – popisuje rozšiřující mechanismy
- další diagramy, které nejsou součástí oficiálního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

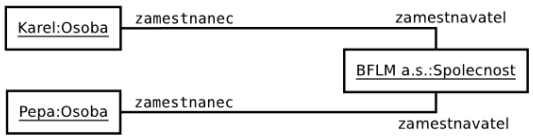
- **Diagram tříd**
  - zobrazuje třídy a statické vztahy mezi nimi
  - Vztahy mezi třídami
    - zobecnění (generalization)
    - asociace (association)
    - závislost (dependency)
    - realizace (realization)
  - **Dědičnost tříd**
    - vztah generalizace/specializace mezi třídami
    - odvozená třída sdílí atributy, chování, vztahy a omezení obecnější třídy
    - odvozená třída může přidávat a modifikovat atributy a chování

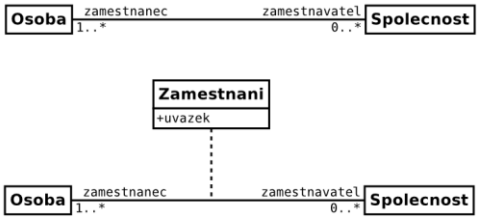


    -
    - **Operace vs. metoda**
      - operace reprezentuje abstraktní pohled na chování objektu
      - metoda implementuje operaci
      - signatura operace = název, typ návratové hodnoty, typy všech stejně seřazených argumentů
    - **Abstraktní operace a třídy**
      - odložení implementace operace na potomky
      - abstraktní třída deklaruje všechny operace, ale některé ponechává bez implementace
      - např. třída Tvar a operace kreslit a obsah

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

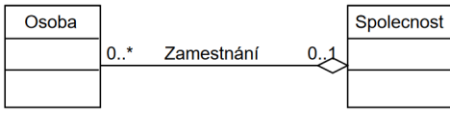


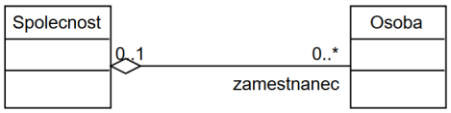


  -
  - **Vlastnosti asociace**
    - objekt má ve vztahu svou roli
    - asociace má své násobnosti (mohutnosti)
      - násobnost je odrazem cíle modelu
      - bez této znalosti nelze určit špatnou/dobrou násobnost
    - asociace má svůj název
      - název může být sloveso nebo podstatné jméno
      - Zaměstnání; ⇒ je zaměstnán v ; ⇐ 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
    - 

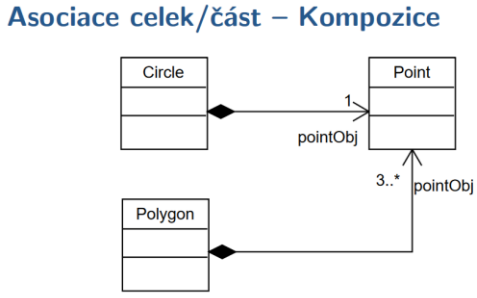
  - Asociace vyššího stupně
    - binární asociace (vztah dvou tříd, resp. jejich instancí)
    - N-ární asociace (vztah více tříd, resp. jejich instancí)
      - jsou méně časté,
      - většinou se dají převést na binární asociace,
      - pokud ne, bývá nutné povýšit asociaci na třídu
  - **Asociace celek/část – Agregace**
    - celek je seskupen z více částí

- celek = agregační (seskupený) objekt
- část celku = konstituční (tvořící) objekt
- 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 nemívá název (vyjadřuje vztah má)
  - agregace bývají homeometrické (tj. konstituenti patří do téže třídy)
- Zaznačíme ji takto:

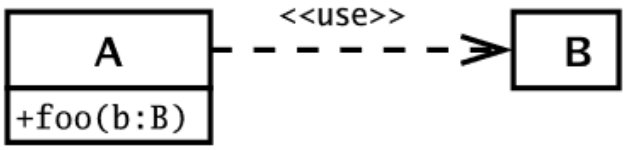




- **Asociace celek/část – Kompozice**
  - Kompozice (Složení)
    - celek je složen z více částí
    - celek = kompozitní (složený) objekt
    - část celku = komponentní (složkový) objekt
  - Vlastnosti kompozice
    - **složený objekt (většinou) neexistuje bez svých komponent**
    - **komponenta (komponentní objekt) může být součástí nejvýše jedné kompozice**
    - implicitní násobnost každé složky je 1
    - asociace kompozice nemívá název
    - kompozice bývají heterometrické (tj. komponenty patří do různých tříd)
    - při rušení celku se ruší jeho složky (příp. jsou z kompozice vyňaty)
  - Zaznačíme ji takto

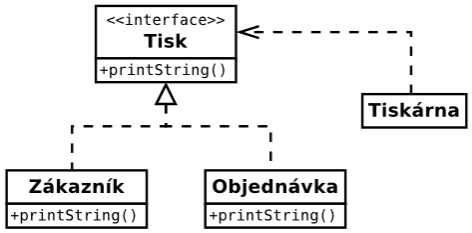


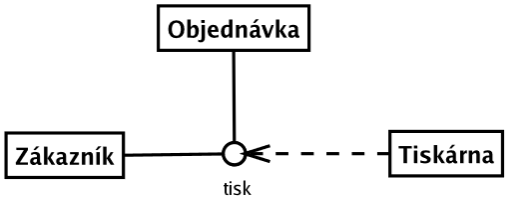
- **Závislost tříd**
  - vyjadřuje jiné různé vztahy mezi objekty či třídami
  - typ závislosti se označuje pomocí stereotypů
  - Nejběžnější typ stereotypu – používání <<use>>
  - A (klient) → B (dodavatel)
    - metoda třídy A potřebuje argument třídy B
    - metoda třídy A vrací hodnotu třídy B
    - metoda třídy A používá objekt třídy B, ale ne jako atribut



- závislost bez uvedeného stereotypu se považuje za používání
- 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
  - …

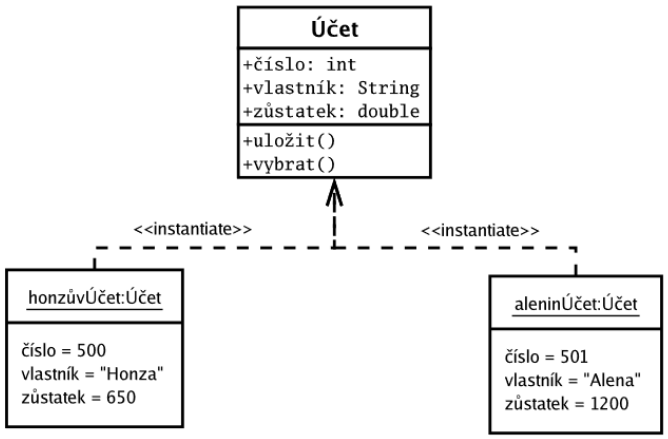
- **Realizace tříd**
  - **Rozhraní**
    - množina operací, které určují chování objektu
      - pouze definuje, co objekt umí (nabízí), nedefinuje jak
      - způsob provedení operací závisí na jejich implementaci třídami
    - lze modelovat jako speciální prvek
    - zjednodušení návrhu, omezuje počet vazeb
  - **Realizace**
    - vztah mezi třídou a rozhraním
    - třída implementuje všechny operace (metody) z daného rozhraní
    - třída používající rozhraní pak umí používat i jeho implementační třídy



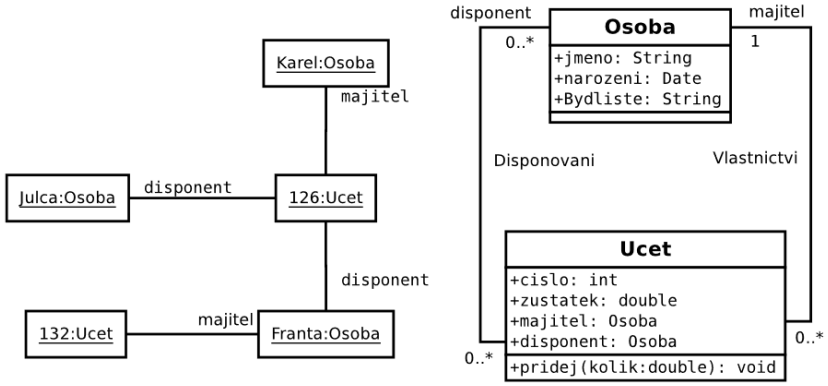


- **Diagram objektů**
  - je úzce svázán s diagramem tříd

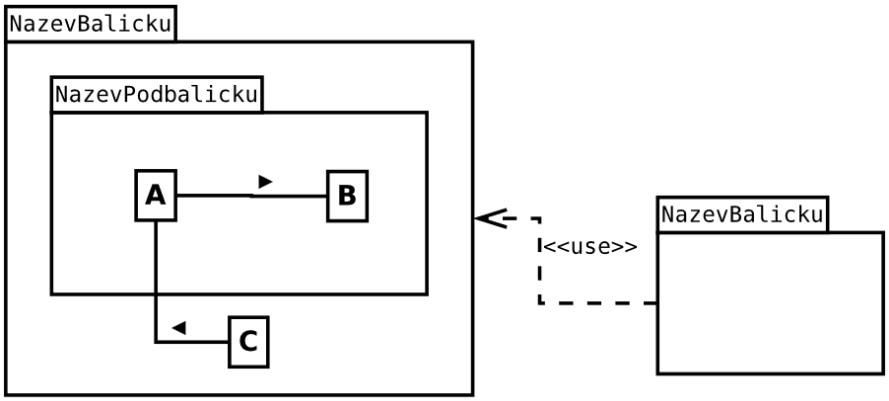
- znázorňuje objekty a jejich relace v určitém čase
- relace jsou dynamické (nemusí trvat po celou dobu existence objektů)



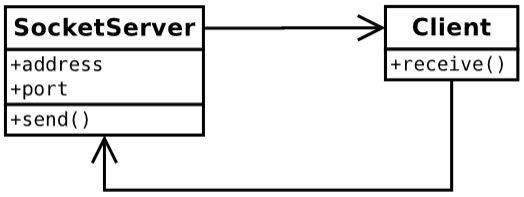
- mezi objekty existuje spojení (spojení = instance vztahu asociace)



- **Diagram seskupení (balíčků) (Package Diagram)**
  - seskupení sémanticky (významově) 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



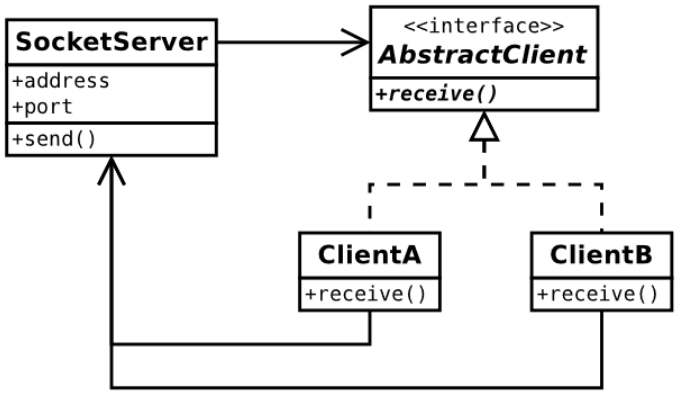
- **Principy objektově orientovaného návrhu**
  - **Problémy spojené se špatným návrhem**
    - změna v softwaru je náročná a vyžaduje úpravy na mnoha místech
    - změna způsobí problémy v jiných, mnohdy nesouvisejících částech softwaru
    - vyčlenit část softwaru pro znovupoužitelnost je náročnější než tuto část vytvořit znovu
  - **Principy objektově orientovaného návrhu architektury**
    - předkládají vhodné postupy pro návrh architektury
    - vhodné z pohledu údržby a rozšiřitelnosti architektury systému
  - **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 ≠ metoda; metody jsou implementovány, aby byla splněna zodpovědnost
  - **Open Closed Principle (OCP)**
    - třída by měla být otevřená pro rozšíření ale uzavřená pro modifikace
    - třída by měla být rozšiřitelná bez nutnosti modifikace kódu



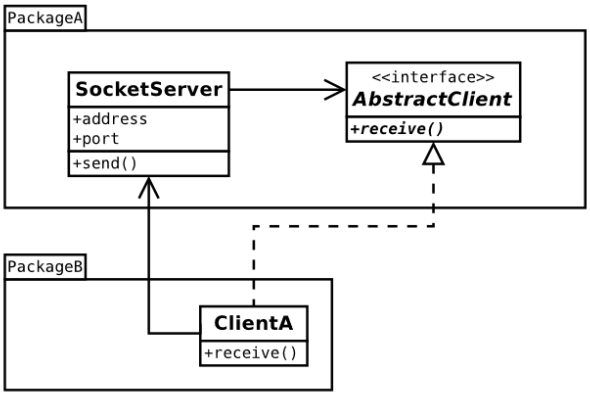
  - **Dependency Inversion Principle (DIP)**
    - závislost na abstraktním ne na konkrétním
    - závislosti by měly směřovat jedním směrem, a to od konkrétního k abstraktnímu
    - závislosti by měly směřovat ke společným rozhraním a abstraktním třídám
    - **Důsledky**
      - redukce závislosti v kódu
      - abstraktní rozhraní se mění mnohem méně než konkrétní implementace ⇒ zavislý kód se nemusí měnit tak často
      - snadná možnost nahradit jednu implementaci za jinou

  - **Problém**

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



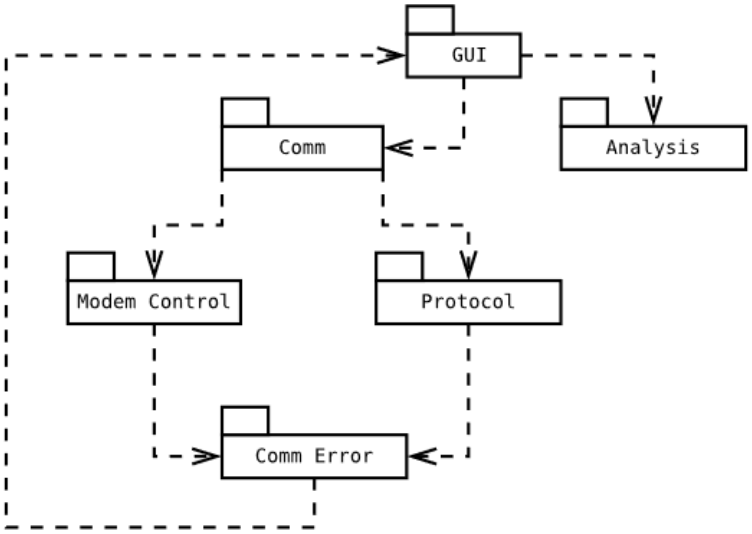
-
- **Balíčky**
  - třídy a rozhraní řešící komunikaci dáme do jednoho balíčku
  - třídy obsluhující klienta patří do jiného balíčku



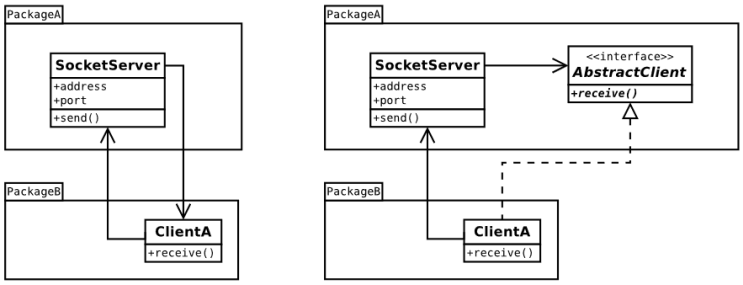
  -

- **Principy návrhu balíčků (komponent)**
  - Release Reuse Equivalency Principle (REP)
    - granularita znovupoužitelnosti je shodná s granularitou uvolnění nové verze
  - 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 usnadňují práci při údržbě

- **Acyclic Dependencies Principle (ADP)**
  - závislosti mezi balíky nesmí tvořit cykly
  - minimální počet závislostí mezi balíky ⇒ jednodušší údržba a uvolňování nových verzí (menší počet závislých balíků pro testování)
  - závislosti s cykly ⇒ velký počet závislých balíků



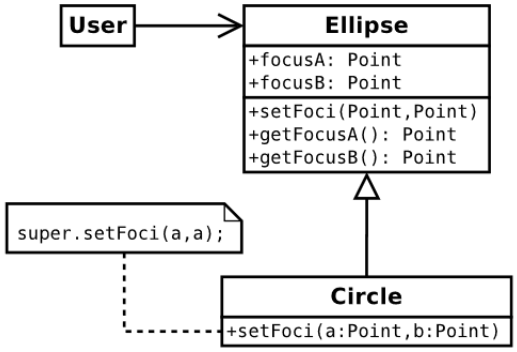
-
- Balíčky
  - aplikací DIP jsme odstranili cyklickou závislost (ADP)



-

- **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

- **Liskov Substitution Principle (LSP)**
  - odvozené třídy by měly být zaměnitelné za bázové třídy
  - uživatelé bázové třídy by měli být schopni pokračovat bez chybného chování i při nahrazení odvozenou třídou
  - Příklad kružnice / elipsa
    - kružnice je elipsa se stejnými ohnisky



-
- LSP – Příklad kružnice / elipsa
    - kružnice je konzistentní sama se sebou
    - elipsa je konzistentní sama se sebou
    - tyto objekty jsou však používané 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;
```
- LSP – Příklad kružnice / elipsa
    - kružnice není zastupitelná za bázovou 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

- **Do not Repeat Yourself (DRY)**
    - neopakujte stejný kód na různých místech
    - problémy s modifikací a udržovatelností
    - opakující se kód ⇒ samostatná metoda
    - soubor opakujících se metod ⇒ vytvoření obecnější třídy