OMO

2 - Objektově orientovaný přístup

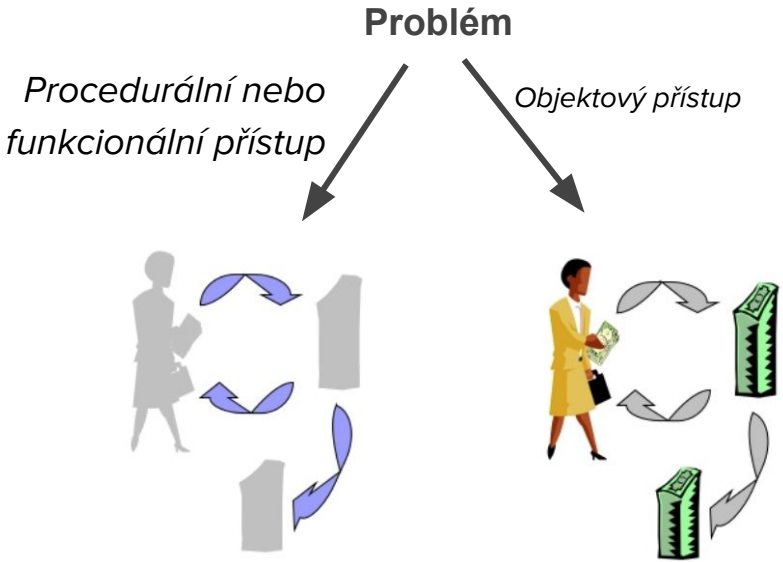
- Objekty, třídy a hierarchie tříd
- Interface a abstraktní třídy
- Dědičnost
- Message passing
- Class diagramy a příklady systémů modelovaných pomocí OOP
- Volba správného přístupu
- Rozdíl mezi asociací, agregací a kompozicí
- Dědičnost versus kompozice
- Polymorfismus
- SOLID
- Správná reprezentace

Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, *david.kadlecek@cz.ibm.com*

Verze 22.10.2018

Objektový přístup

Problém se snažím řešit tak, že ho kompletně strukturuji do objektů, které mezi sebou komunikují. Objekty intuitivně volím tak, aby co nejvíce odpovídaly objektům z reálného světa.



- výběr peněz
- vklad peněz
- převod peněz mezi účty

- zákazník
- účet
- peníze

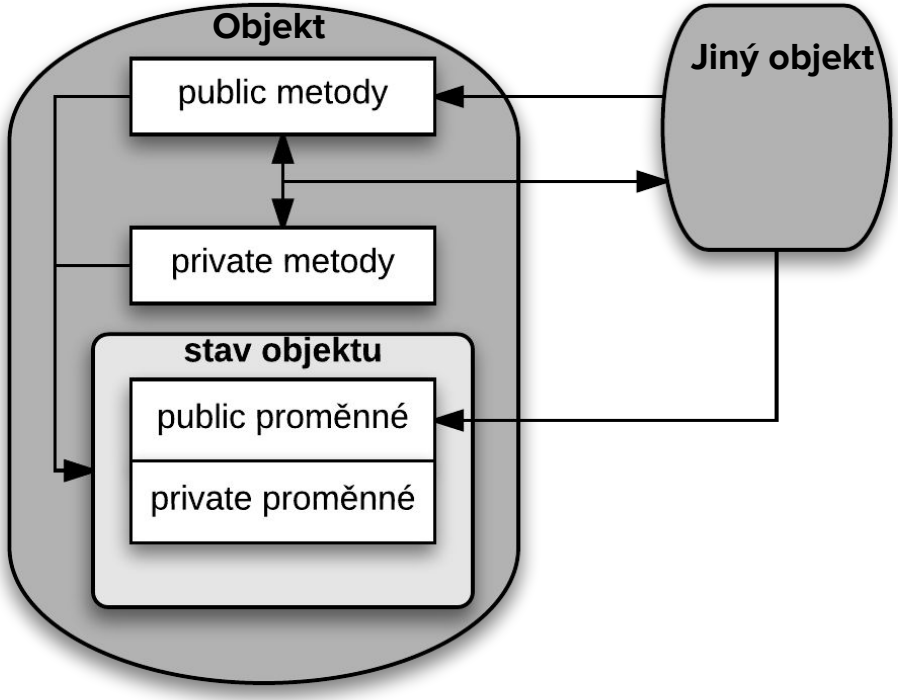
Jaký přístup je lepší?

Záleží na problému, který řeším

Objekt

Objekt do sebe zapouzdřuje ***proměnné***, které reprezentují jeho stav a **metody**, které s tímto stavem pracují (vracet hodnotu, upravit hodnotu ...)

Uvnitř objektu má všechno přístup ke všemu. Z vnějšího světa lze volat pouze veřejné metody (***public***) a napřímo pracovat pouze s veřejnými proměnnými (narozdíl od ***private***).



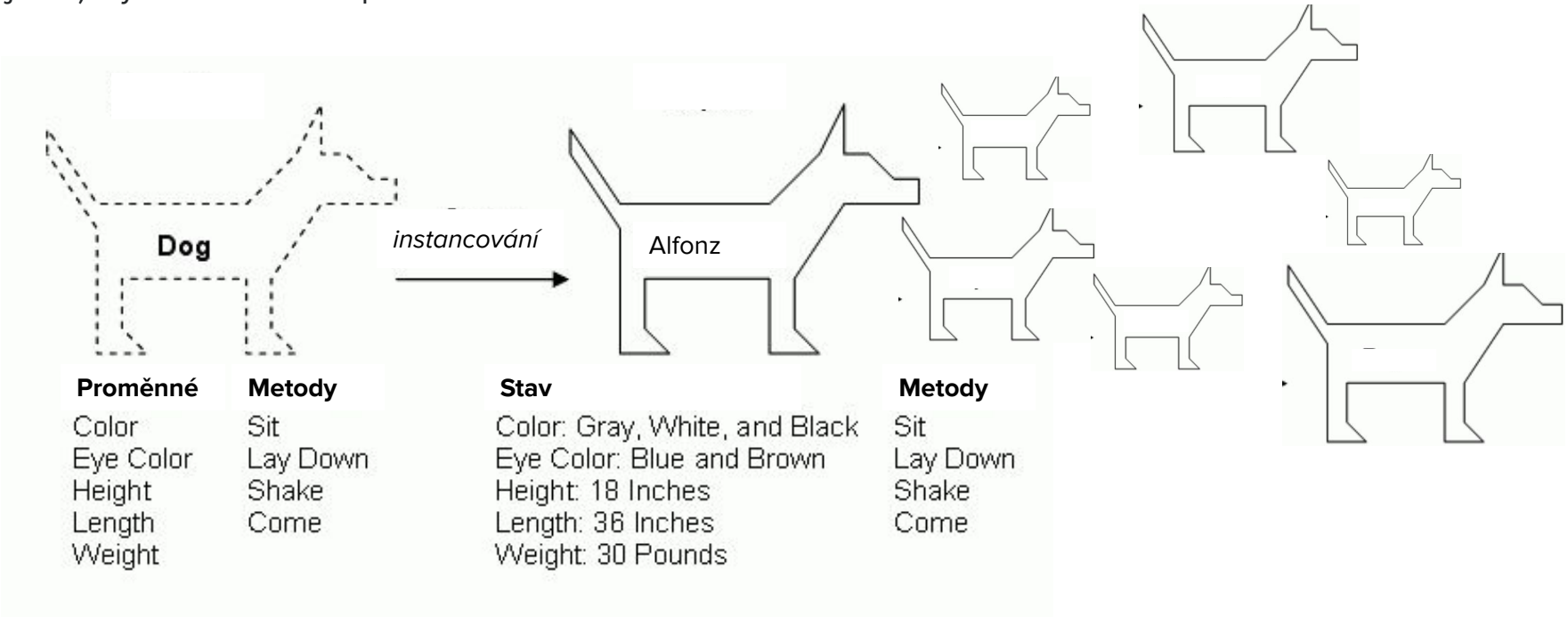
Program Karel - Karel stojí na nějaké pozici a když chce, tak se usmívá. Dětský programátor může Karla posouvat dopředu, dozadu a řídit jestli se usmívá. Při návrhu nechci, aby programátor mohl přímo nastavovat pozici Karla, ale pomocí jednoduchých programovacích pokynů ho posouval.

```
public class Karel {
   private int position;
   public boolean isSmiling;
   public int getPosition() {
       return position;
   }
   public Karel(int position){
       this.position = position;
   }
   private void updatePosition(int step){
       position = position + step;
   }
   public void stepForward() {
       updatePosition(1);
   };
   public void stepBackward() ...
}
```

```
  public class BabyProgrammer {
     public static void main(String[] args){
         Karel myKarel = new Karel(10);
         myKarel.isSmiling = false;
         myKarel.stepForward();
         myKarel.stepForward();
         myKarel.stepBackward();
         myKarel.isSmiling = true;
         System.out.println("Position is " +
myKarel.getPosition());
     }
  }
```

Třída

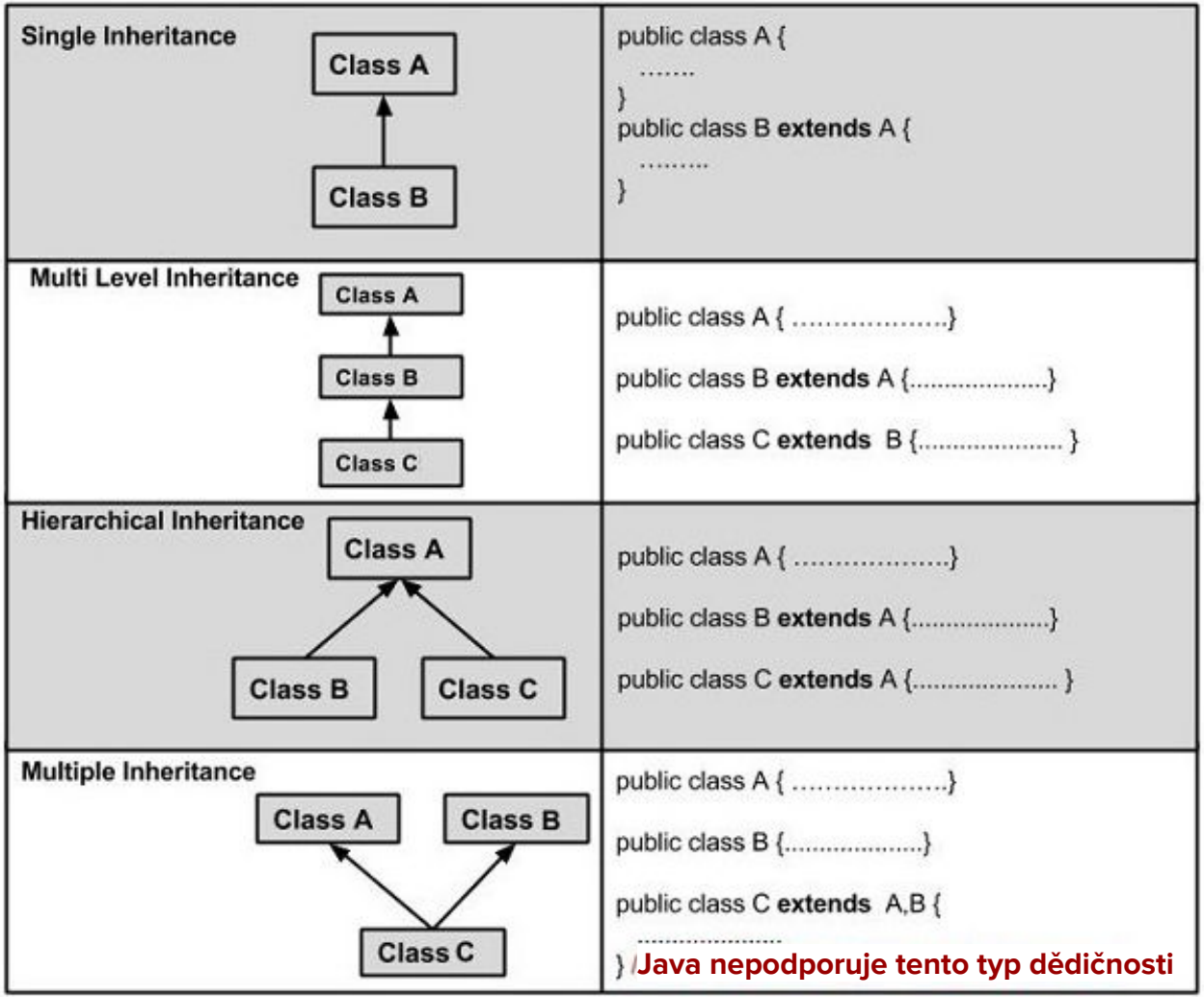
Třída je “**předpis” pro tvorbu a chování objektů** - instancí této třídy. Z každé třídy mohu vytvořit *N* instancí (objektů). Ty se od sebe liší pouze stavem.



Dědičnost

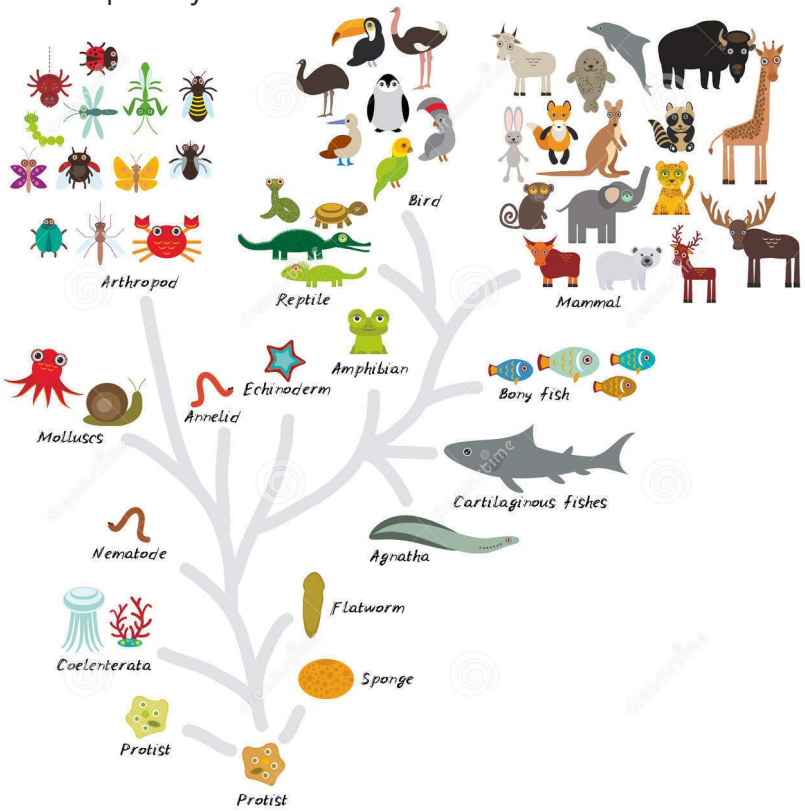
Při dědičnosti třída potomka přebírá vlastnosti předka

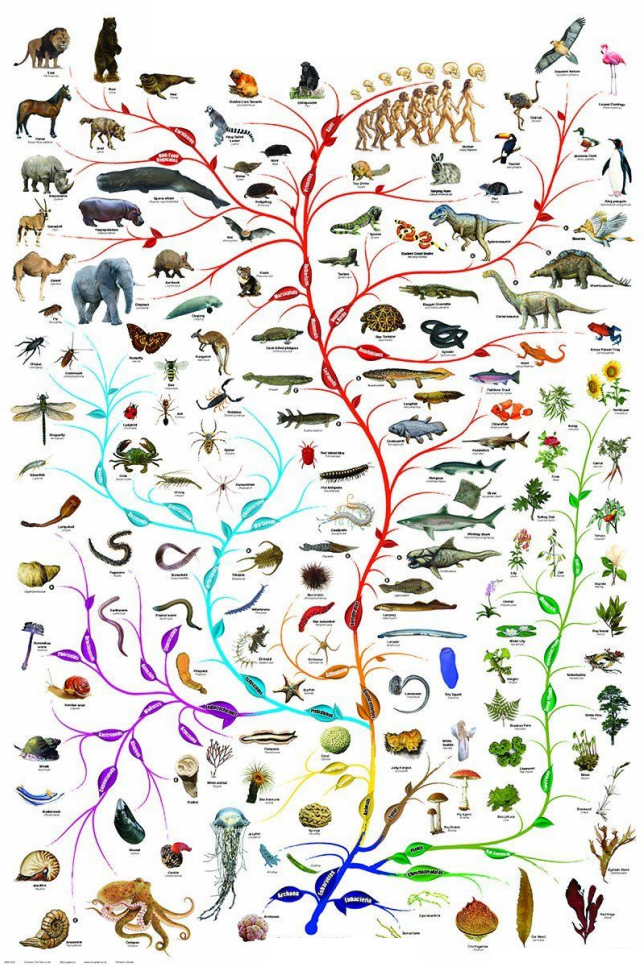
Potomek může přepoužívat proměnné a metody předka pokud je nepředefinuje



Hierarchie tříd

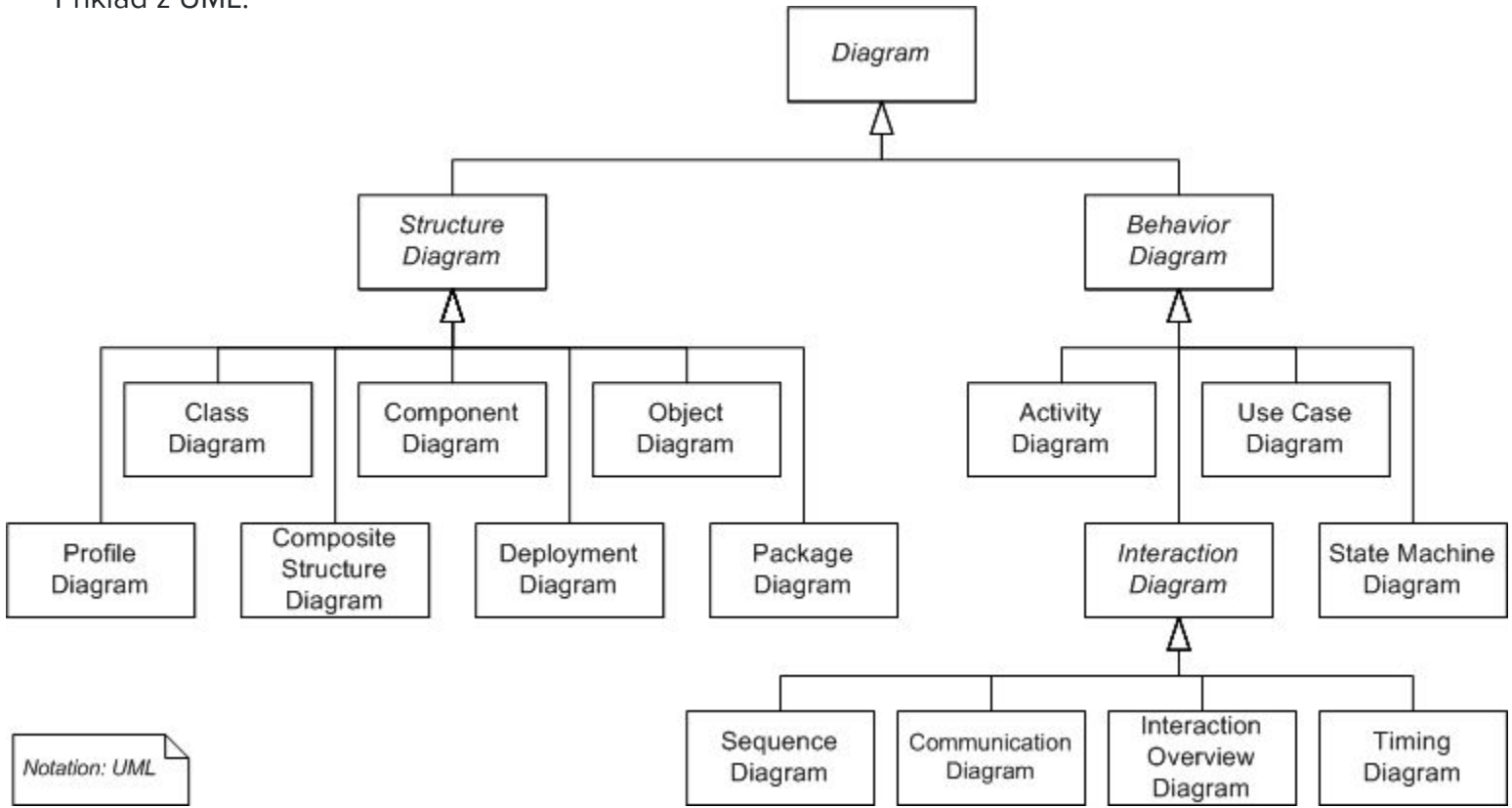
Příklad z přírody:





Hierarchie tříd

Příklad z UML:



Interface

Interface (neboli rozhraní) je “předpis” pro to, jaké metody musí třída implementovat (realizovat).

Třída musí implementovat všechny metody - ne jen některé => interface tedy definuje kompaktní skupinu spolu souvisejících funkcionalit a předdefinovává rozhraní kterým budu k objektu přistupovat.

```
Interface Bicycle {

void changeCadence(int newValue);

void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);
}
```

```
public class SpecializedVengeBicycle implements Bicycle{
  int cadence = 0;
  int speed = 0;
  int gear = 1;
 public void changeCadence(int newValue) {
      cadence = newValue;
  }
  public void changeGear(int newValue) {
      gear = newValue;
  }
  public void speedUp(int increment) {
      speed = speed + increment;
  }
  public void applyBrakes(int decrement) {
      speed = speed - decrement;
  }
  void printStates() {
      System.out.println("cadence:" + cadence + " speed:" + speed + " gear:" + gear);
 }
}
```

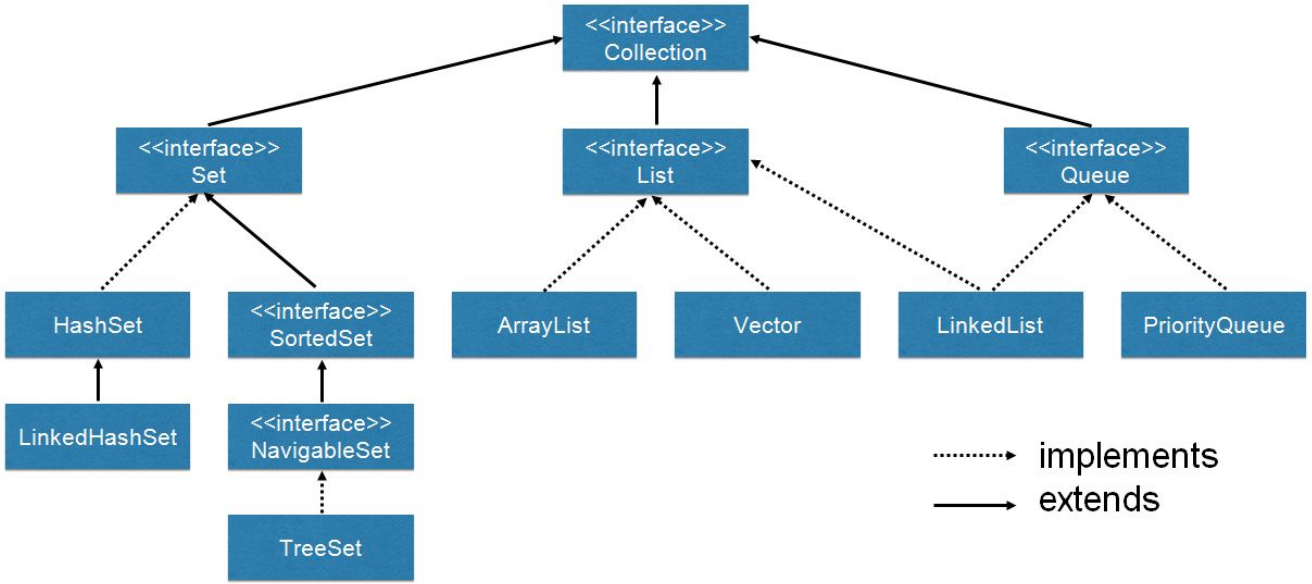
Interface

Implements - třída implementuje interface (realizuje jeho metody)

Extends - interface extenduje (dědí) vlastnosti rodičovského interface

Třída může implementovat více interface najednou

Příklad rozhraní v Java Collection API



Abstraktní třída

Abstraktní třída je taková, kterou **nelze instanciovat (nelze od ní vytvořit objekt)**

Uvnitř abstraktní třídy jsou implementované metody, které pak přebírají potomci této abstraktní třídy

Uvnitř abstraktní třídy mohu definovat abstraktní metody, které nemají implementaci. Tedy jako bych definoval rozhraní, které pak musí implementovat potomci této abstraktní třídy

```
public abstract class AbstractEmployee {
    private String name;
    private String address;
    private int number;

    public AbstractEmployee(String name, String address, int number) {
        this.name = name;
        this.address = address;
    }

    public abstract double computePay();
    public abstract int getNumber();

    public void mailCheck() {
        System.out.println("Mailing check to"+this.name+" " + this.address);
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String newAddress) {
        address = newAddress;
    }
}
```

Rozdíl mezi abstraktní třídou a interface

Interface - všechny proměnné jsou automaticky *public static final* a metody *public*.

Abstraktní třída - lze definovat také proměnné, které nejsou *static* a *final* a metody mohou být *public, protected, private*

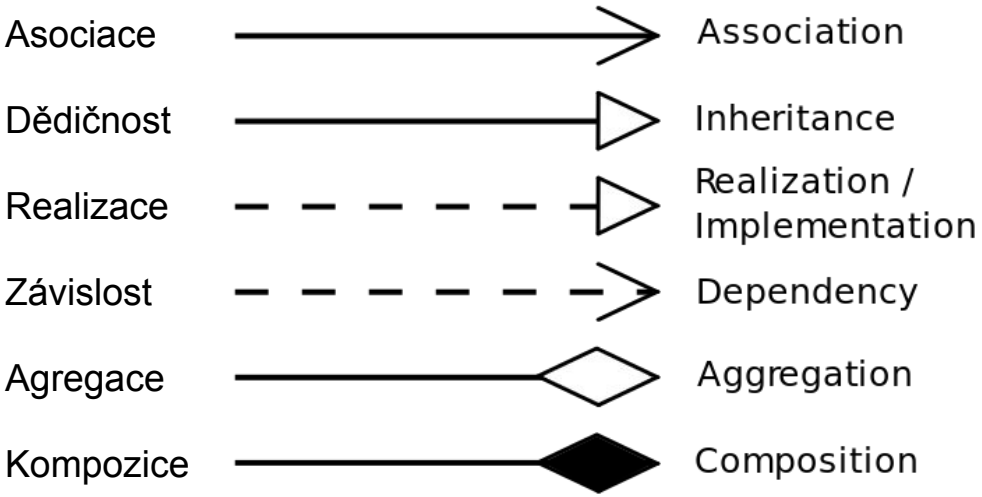
***Abstraktní třídu použiji, jestliže*:**

1. Chci sdílet kód mezi více třídami a neexistuje “neabstraktní” předek těchto tříd
2. Třídy jsou mezi sebou úzce spjaté - sdílí mezi sebou mnoho proměnných a metod
3. Potřebujeme předepsat i metody, které neslouží k vnější komunikaci s objektem, tedy *private* a *protected*.
4. Chci předefinovat proměnné, které nejsou *static* a *final*.

***Interface použiji, jestliže*:**

1. Vytvářím vnější rozhraní k objektům - vytvářím veřejné API ke své komponentě
2. Chci, aby i objekty z jiné class hierarchie implementovaly stejné rozhraní
3. Předepisuju pouze metody a nikoliv jejich implementaci - mým cílem není, aby třídy sdílely implementaci
4. Chci předepsat třídě, aby implementovala **metody z více rozhraní**

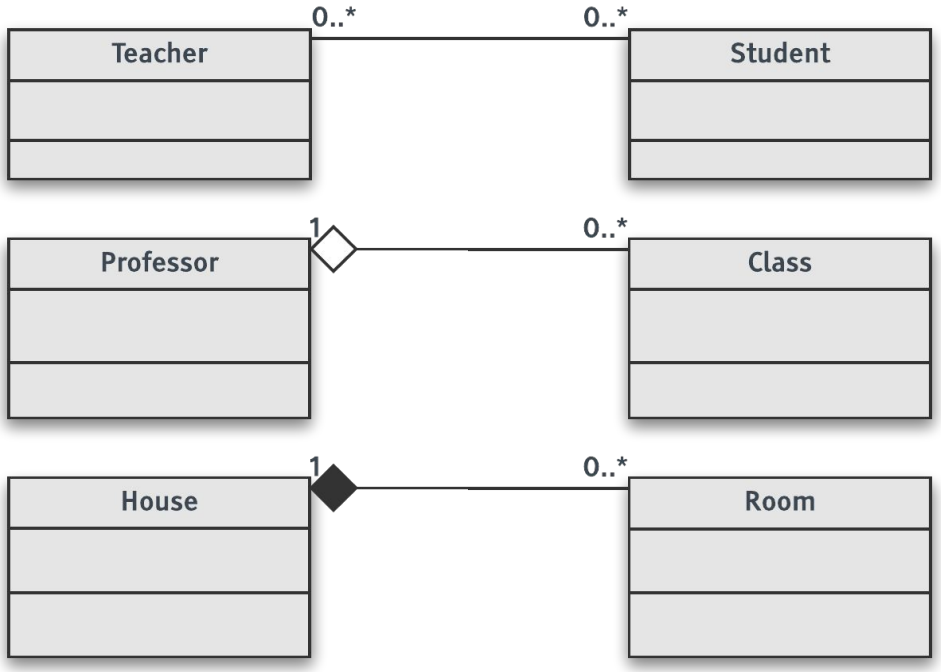
Vztah mezi dvěma objekty



Rozdíl mezi agregací, kompozicí a asociací

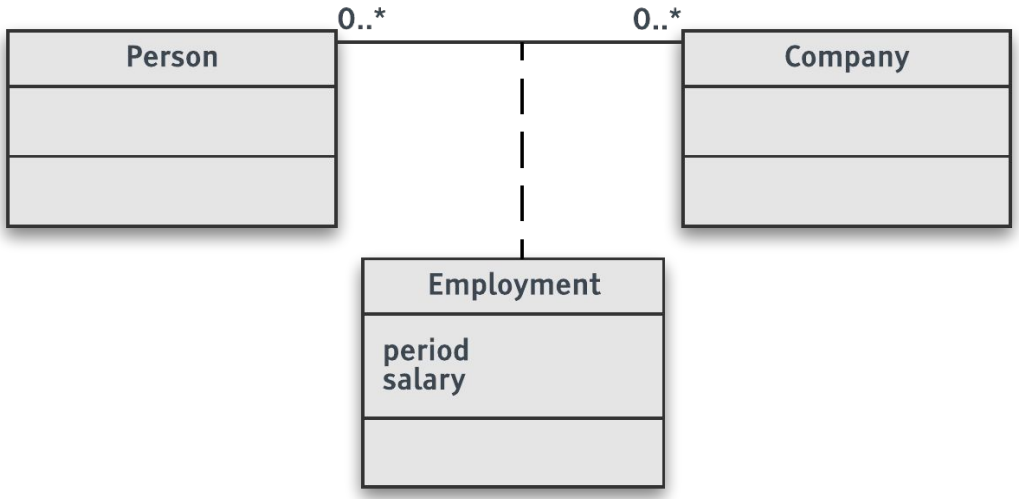
Liší se od sebe primárně sílou vazby

- ***Asociace*** - objekty mají zcela nezávislý životní cyklus, realizována jako proměnná držící referenci na instanci nebo proměnná na vstupu metody, jakákoliv multiplicita.
- ***Agregace*** - objekty mají zcela nezávislý životní cyklus, vlastněný objekt nemůže mít dalšího vlastníka, realizována jako proměnná držící referenci na instanci. Nemůže vytvářet cykly, multiplicita 1:1 nebo 0:N.
- ***Kompozice*** - objekty mají svázaný životní cyklus, jeden objekt vlastní druhý a s jeho zánikem i ten druhý zaniká, realizována jako proměnná držící referenci na instanci nebo zanořená (inner) třída. Nemůže vytvářet cykly, multiplicita 1:1 nebo 0:N.

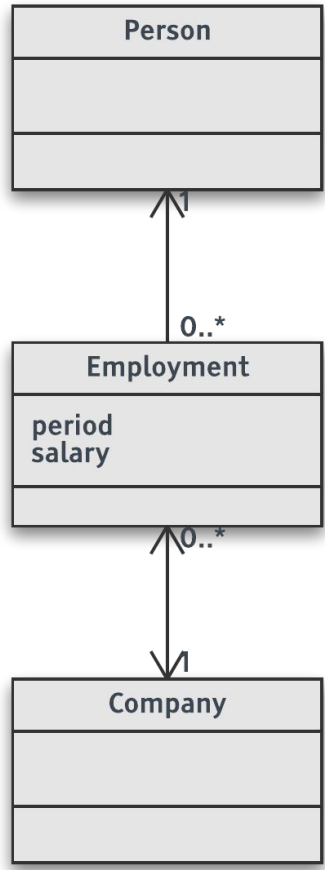


Asociační třída

Asociační třídu potřebuji v případě, že vazby mezi objekty za dvou různých tříd mají různý stav.



Asociační třída - Java realizace



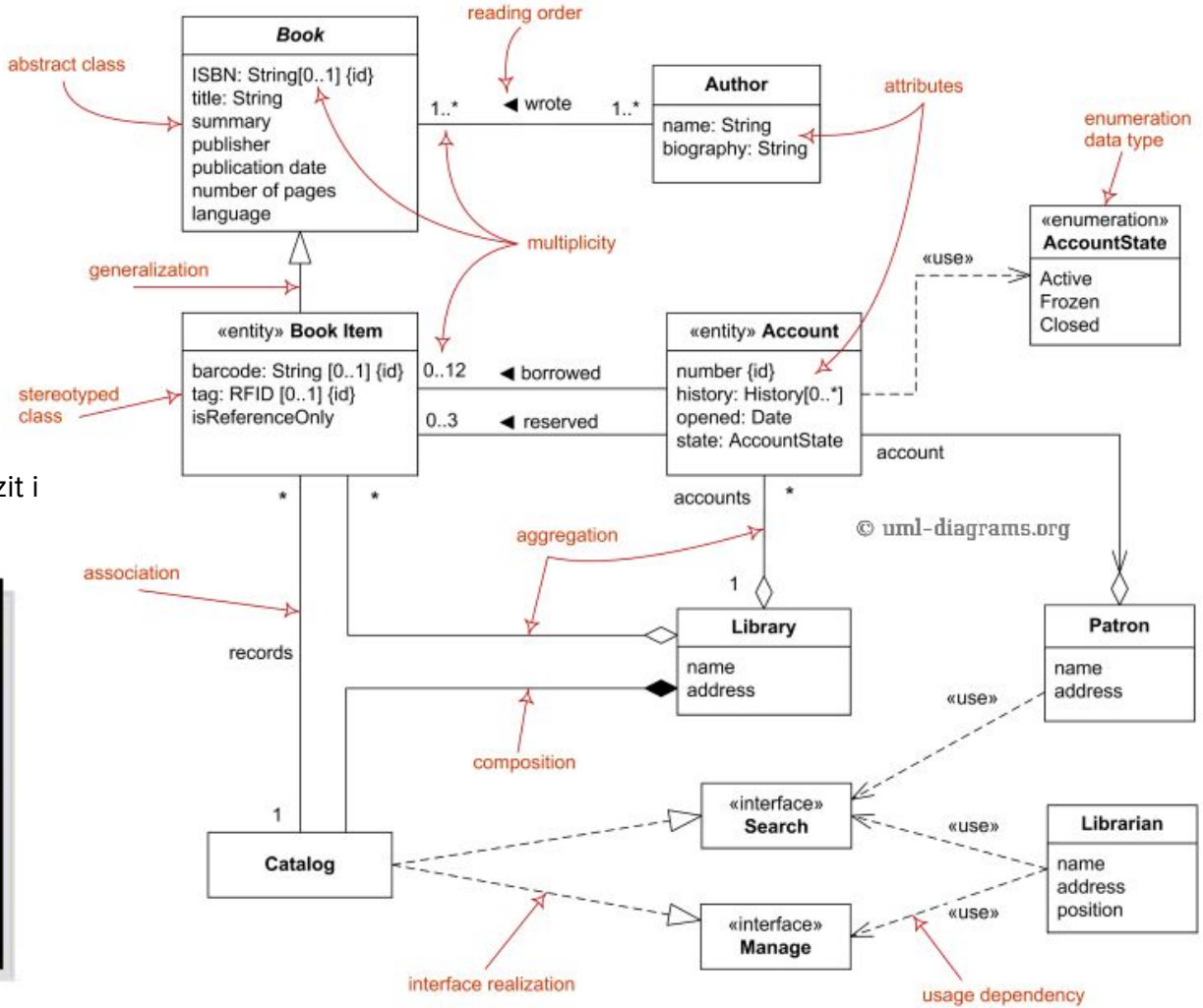
Co navigabilita? Je lepší obousměrná vazba nebo jednosměrná?

- Obousměrná vazba znamená, že při každé změně vazby z jedné strany musím upravit i druhý směr
- Jednosměrná vazba je jednodušší na správu, ale neumožňuje efektivně hledat z obou stran - jak zjistím v jakých všech objektech *Company* je *Person* zaměstnán?

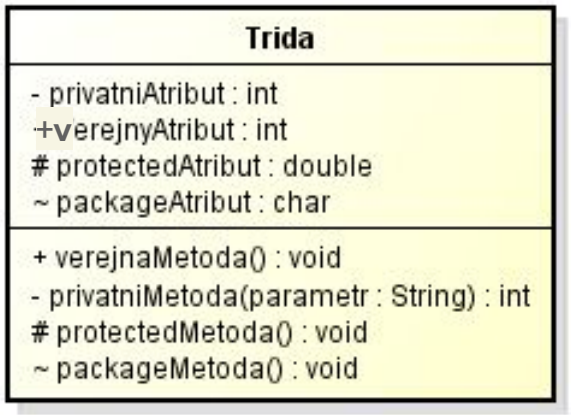
```
public class Person {
}

public class Employment {
   long salary;
   Interval period;
   Company company;
   Person person;
   public Person getPerson() {
       return person;
   }
   …
}

public class Company {
   private Set<Employment> employments;
   Set <Person> getEmployees(){
       Set<Person> employees = new HashSet<>();
       for(Employment employment: employments){
           employees.add(employment.getPerson());
       }
       return employees;
   }
…
}
```

Class diagram

Class diagram je UML diagram pro grafické zobrazení tříd, jejich vlastností a vztahů mezi nimi



Je možné jít do většího detailu a zobrazit i metody a úroveň přístupu

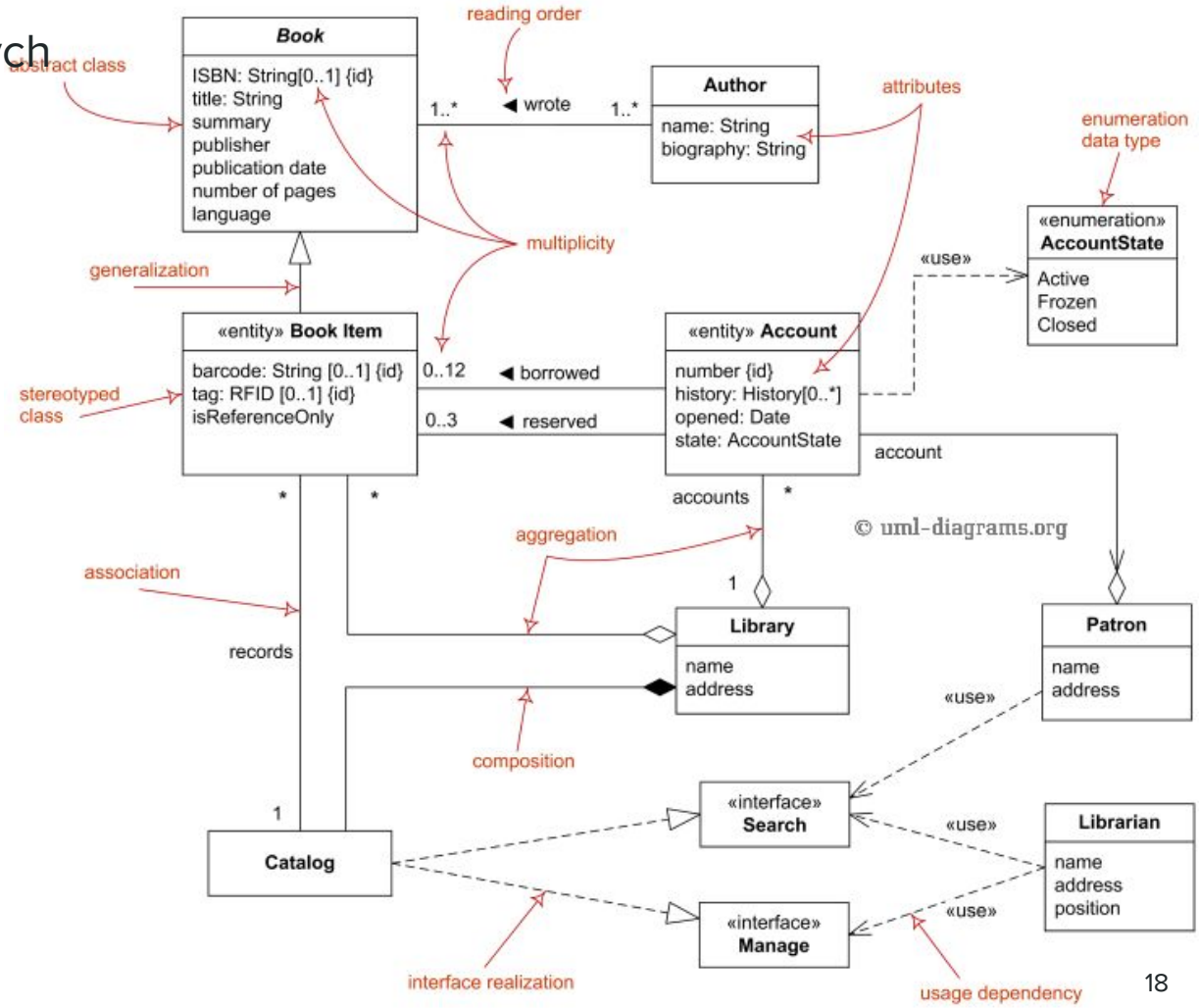


Příklady systémů modelovaných objektovým přístuem

Tento systém reprezentuje knihovnu:

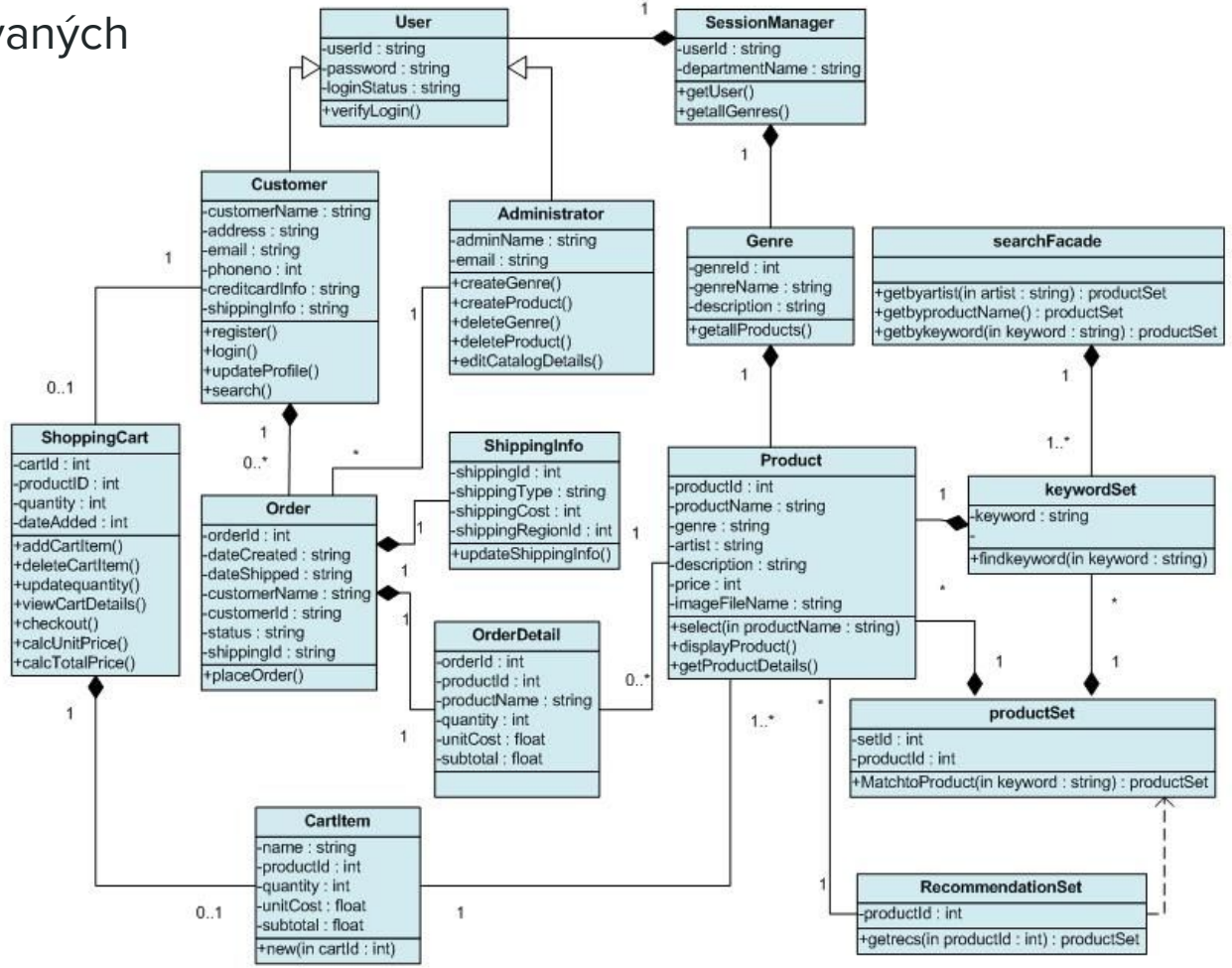
- Mám knihovnu, ta spravuje N účtů, N knih a disponuje jedním katalogem
- U účtů evidujeme v jakém jsou stavu a pro jakého člena jsou otevřeny
- V knihovně se půjčují knihy, abychom je mohli elektronicky evidovat a vyhledávat, tak jsou opatřeny čarovým kódem a RFID
- Ke katalogu můžeme přistupovat pomocí rozhraní pro vyhledávání a správu

Systém vhodný pro realizaci pomocí OOP



Příklady systémů modelovaných objektovým přístuem

Tento class model reprezentuje systém pro správu objednávek



Systém vhodný pro realizaci pomocí OOP

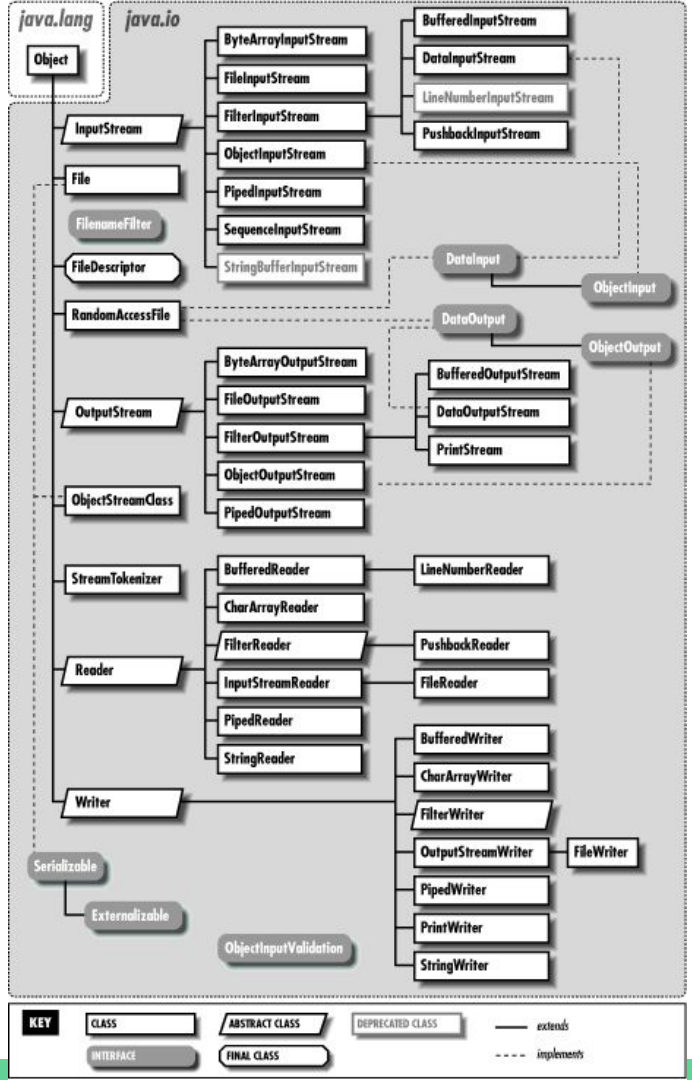
Příklady systémů modelovaných objektovým přístupem

Tento class diagram ukazuje třídy a rozhraní v package *java.io*. Ukazuje, že i Java knihovny jsou realizovány pomocí OOP.

Při debuggování Javy je zřejmá hierarchie tříd a modularizace, je jednoduše pochopitelné trasovat co se právě děje

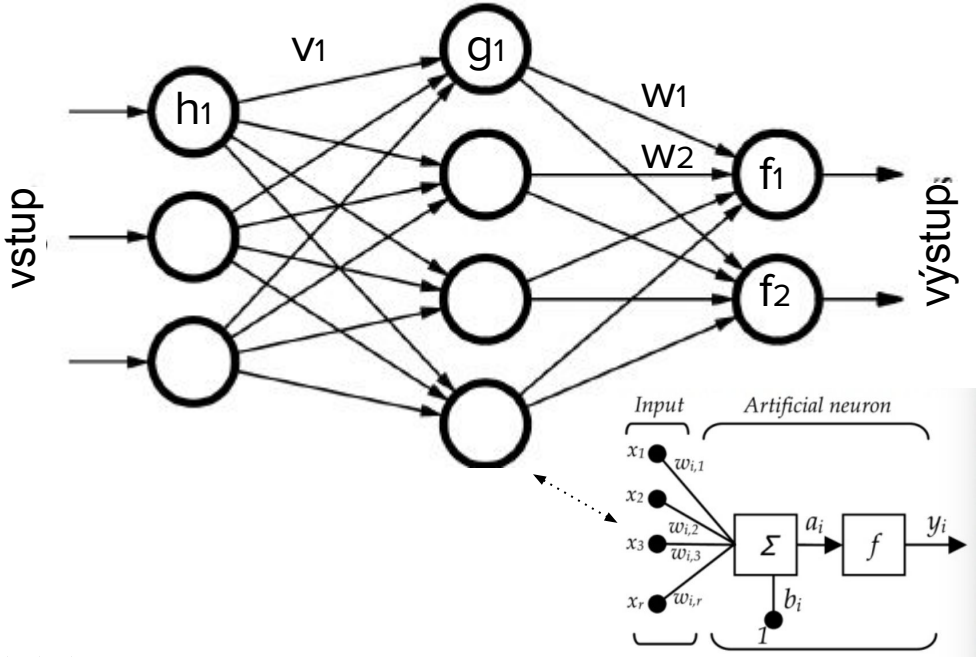
Negativem je, že neustále procházíte desítkami vrstev tříd a jejich předků, kde v každé metodě je pár řádků kódu (stack trace se nevejde ani na jednu stránku)

Java je realizovaná pomocí OOP

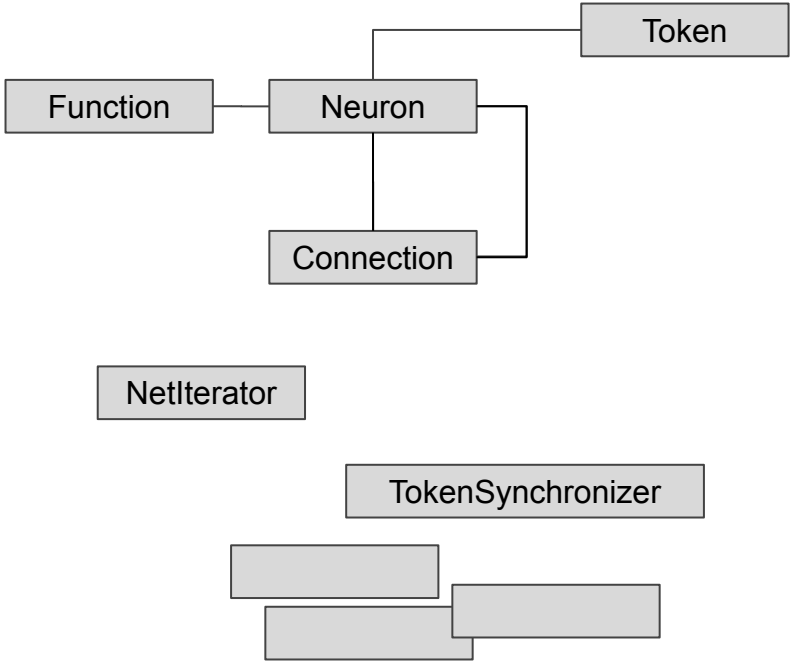


Jak budu modelovat neuronovou síť?

Chci realizovat aplikaci pro výpočet v neuronové síti (nebo jiném výpočetním grafu). Na vstup mi přichází signál, který se přenáší po vazbách mezi nody (neurony). V neuronu se sečtou signály ze všech příchozích vazeb (s příslušnými váhami) a na tento součet se aplikuje funkce. Pak signál pokračuje k dalším neuronům až dorazí na výstup.

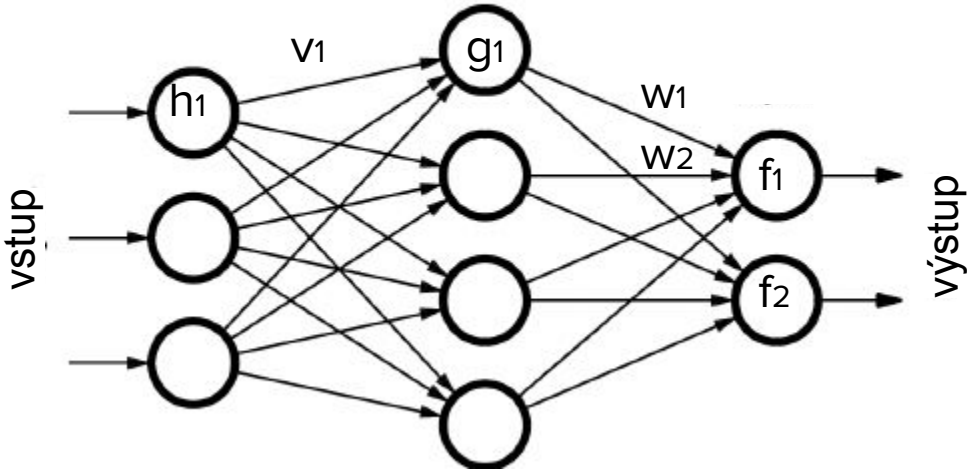


Animace:



... a přidávám další třídy, abych byl schopný systém realizovat pomocí OOP. A to jsem ještě nenapsal ani řádku kódu

Jak budu modelovat neuronovou síť?



=> Systém nevhodný pro realizaci pomocí OOP

Versus:

```
public double f(double in){
   return Math.atan(in)
}
```

Vystup1 = f (w1 * g1(v1 * h1 (...)) + w2 *g2(...) ...)

Vystup2 = f (...

... kde výpočet hodnot na výstupu realizuji pomocí pomocí pár řádků kódu a bez nutnosti složité objektové reprezentace

Komunikace mezi objekty

Objektové orientovaný přístup je definovaný jako zapouzdření proměnných a metod do objektů, které si mezi sebou posílají zprávy. Tzv. **message passing** je předávání zpráv mezi dvěma objekty. Příkladem je komunikace mezi objekty v Smalltalk.

Jaká je analogie z reálného života?

- *Malý Karlík se rozhodne, že napíše dopis své babičce.*
- *Vezme papír, vezme tužku, napíše dopis a odnese ho na poštu. Pak se vrátí domu a jde spát*
- *Dopis přijde babičce do její schránky*
- *Babička když každé ráno pouští psa Alíka, tak kontroluje svou schránku*
- *Jednoho rána je celá radostí bez sebe, ve schránce je dopis od jejího jediného vnoučka, vnouček ji prosí o jeho oblíbené sušenky, tak se babička sebere a ihned začne péct sušenky*

Jaká je realizace v Java pomocí standardního message passingu?

- *Hlavní program zavolá na objektu Karlik metodu sendMessage() a předá jí referenci na objekt Babicka*
- *Metoda sendMessage() v objektu Karlik zavola synchronně metodu receiveMessage() na objektu babicka, z této metody se vola další metoda bakeCookie()*
- *Objekt Karlik čeká než babička dopeče buchty, hlavní program čeká na Karlika*

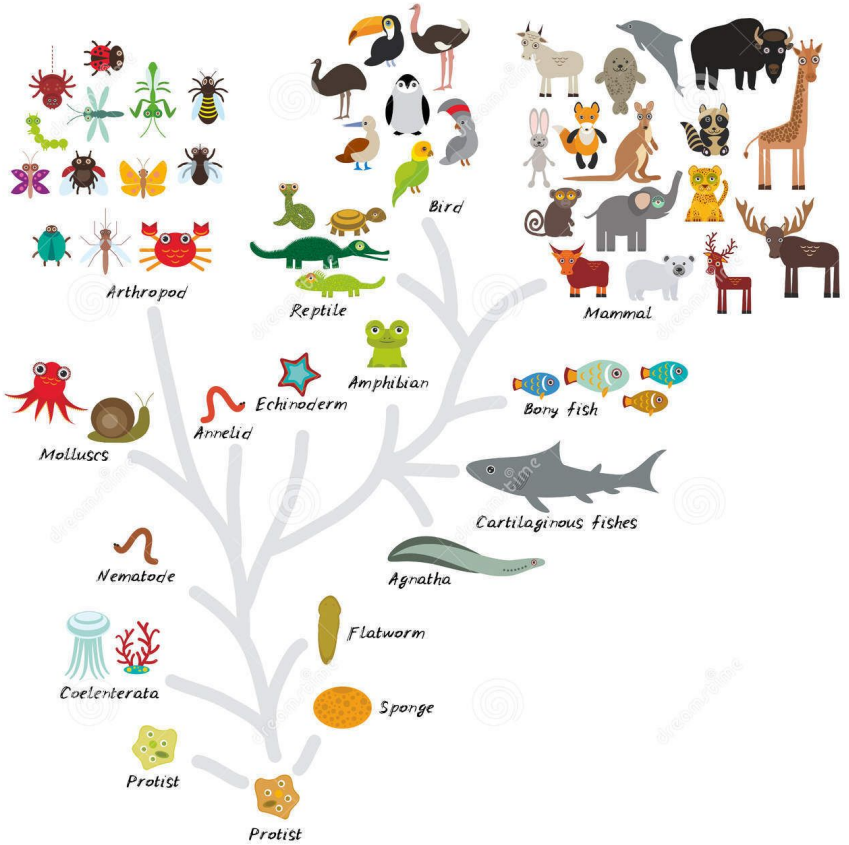
Java ve skutečnosti neimplementuje opravdový message passing mezi objekty =>

- Synchronní komunikace mezi objekty se děje jednoduše přes provolání metody druhého objektu, thread je blokován
- Asynchronní posílání zpráv si musíte doprogramovat nebo využít nějakou existující nadstavbu (např. jms, kafka,...)

Je lepší dědičnost nebo kompozice?

Mám následující class hierarchii =>

... ne příliš podobná zvířata jsou od sebe v class hierarchii vzdálena



Je lepší dědičnost nebo kompozice?

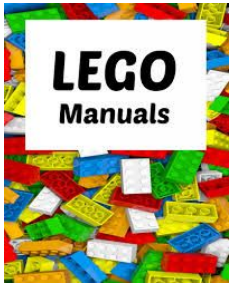
<= ... a chci implementovat toto

... co se stane, když budu chtít přenést vlastnosti z jednoho místa hierarchie tříd do jiné části u hodně komplexního systému?

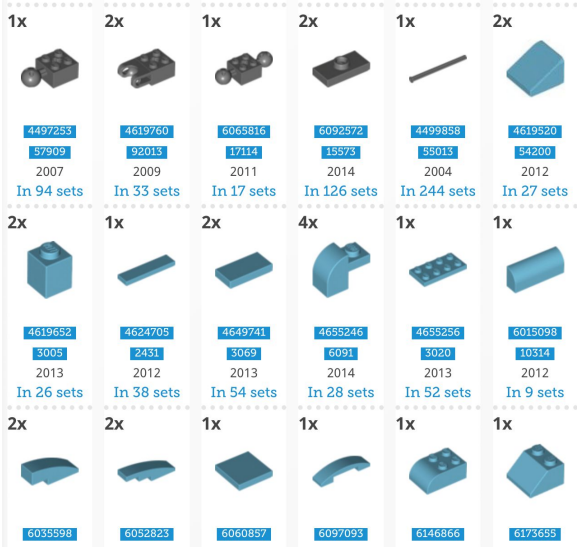
Je lepší dědičnost nebo kompozice?

Co je Lego?

Specifikace podle které dělam kompozici



Kostky, ze kterých dělám kompozici

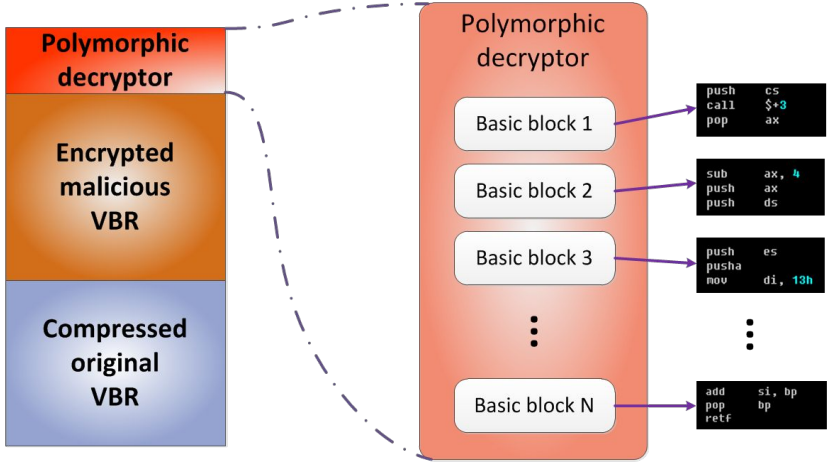


Je lepší dědičnost nebo kompozice?

- Dědičnost má výhodu v tom, že zavádí pravidla a minimalizuje duplicity v kódu
- Dědičnost je nicméně extrémně silná vazba, kterou zavádím do svého systému. Strukturální zásahy do hierarchie tříd ve chvíli, kdy už mám implementován komplexní systém, jsou extrémně pracné
- Pro nějaké problémy nelze rigidní hierarchii tříd ani sestavit
- Při návrhu je dobré si dopředu dobře rozmyslet, které struktury a pravidla jsou natolik pevné, že je můžu fixovat pomocí dědičnosti.
- Tam, kde vím, že budu v budoucnosti potřebovat flexibilitu, tak radši volím kompozici
- Kompozice má nevýhodu, že někdy končí duplicitami v kódu a je mnohem více benevolentnější - tedy “hloupý programátor” může udělat značné škody

Polymorfismus

Polymorfní virus - virus, který má v různých časech nebo v různých svých generacích odlišnou formu. Dosahuje to tím, že do svého kódu vkládá náhodně instrukce, které nemění funkcionalitu viru, nicméně slouží k maskování. Virus navíc sám sebe zakryptuje.



VBR (Volume Boot Record) - Inicializační sektor na disku ze kterého se začíná nahrávat operační systém

Polymorfismus v Javě (a obecně v OOP) znamená, že:

- v rámci té samé třídy mám metody, které mají stejné jméno (jiné parametry), ale jiné chování
- V třídě potomka a předka mám metody, které mají stejné jméno (stejné parametry), ale jiné chování

Polymorfismus

Overloading

Vytvoření více metod se stejným jménem metody, ale různými parametry. Různými parametry je zde míněno jiný počet parametrů nebo jiné typy parametrů

- metody se stejnými parametry nemůžou mít různé návratové typy
- děje se mezi metodami v rámci té samé třídy

Overriding

Předefinování metody předka metodou potomka s identickými parametry a návratovým typem

- je možné upravit access level, aby byl méně restriktivní
- návratový typ může být potomkem původního návratového typu
- potomek nemůže vyhazovat novou výjimku
- nelze dělat override statické nebo final metody

Polymorfismus - overloading

Různé počty parametrů

```
class Adder{
   static int add(int a,int b){return a+b;}
   static int add(int a,int b,int c){return a+b+c}
}
class TestOverloading1{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(11,11,11));
   }}
```

Různé typy parametrů

```
class Adder{
   static int add(int a, int b){return a+b;}
   static double add(double a, double b){
     return a+b;}
}
class TestOverloading2{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(12.3,12.6));
   }}
```

Různé návratové typy

```
class Adder{
   static int add(int a, int b){return a+b;}
   static double add(int a, int b){return a+b;}
}
class TestOverloading2{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(12,12));
   }}
```

=> Compile Time Error: method add(int,int) is already defined in class Adder

Polymorfismus - overriding

Co se děje bez overridingu

```
class Vehicle{
   void run(){
      System.out.println("Vehicle is running");
      }
}

class Bike extends Vehicle{

   public static void main(String args[]){
       Bike obj = new Bike();
       obj.run(); //=> Output:Vehicle is
running
   }
}
```

Po úpravě

```
class Vehicle{
   void run(){
      System.out.println("Vehicle is running");
   }
}

class Bike2 extends Vehicle {
   void run() {
       System.out.println("Bike is running");
   }
   public static void main(String args[]) {
       Bike2 obj = new Bike2();
       obj.run(); //=> Output:Bike is running
   }
}
```

Polymorfismus - overriding

Od Javy 5 je možné změnit při overloadingu i návratový typ z předka na potomka - tzv. *kovariantní typ*

```
class A{
   A get(){return this;}
}

class B1 extends A{
   B1 get(){return this;}
   void message(){
      System.out.println("Covariant return type");
   }

   public static void main(String args[]){
       new B1().get().message();
   }
}
```

SOLID - jak psát dobře udržovatelný OOP kód

Single Responsibility Principle

Třída má pouze jeden účel a jednu zodpovědnost a všechny její metody by měly sloužit k plnění tohoto účelu. Proč? Čím méně bude třída plnit účelů, tím bude méně důvodů do této třídy zasahovat. Např. je vhodné rozdělit formátování a generování reportů do různých tříd.

Open/Closed Principle

Třídy by měly být otevřené pro rozšiřování a uzavřené pro modifikaci. V existujících třídách by měl probíhat pouze bug fixing, ale nová funkcionalita by měla přicházet do potomků třídy. Důvodem je opět minimalizace zásahů do již hotového kódu

Liskov Substitution Principle

Objekt je vždy možné nahradit objektem z třídy potomka. Kód pak nemusí kontrolovat s jakým konkrétním podtypem pracuje, Důvodem je, abychom nemuseli v kódu provádět nepříjemné kontroly typu a řešit různé side efekty.

Interface Segregation Principle

Více klient specifických rozhraní je lepší než jedno víceúčelové rozhraní. Důvodem je, že to více specifických rozhraní vede k menšímu couplingu. Např. PersistenceManager implementuje rozhraní *DBReader* a *DBWriter*

Správná reprezentace

- Absence struktury (zde objektové reprezentace) je menší zlo než špatná struktura
  - Pozor zejména na nesprávné relace, jejich směr a multiplicitu
- Správná objektová reprezentace je co nejbližší reálnému světu ve kterém žijeme
- Pokud neimplementujeme reálný systém, tak by reprezentace měla být co nejbližší systému, který realizujeme.

Když programujete tuto počítačovou hru, tak po přečtení knihy na OPP budete intuitivně vytvářet složitou class hierarchii, která reprezentuje typologii "potvor" a typologii překážek. Pak začnete vyhodnocovat pozici SuperMaria podle absolutní pixel pozice přečtené z obrazovky. Obojí je špatně. Místo složité typologie vám stačí obálka (šířka a výška) předmětu, id ikony/textury pro zobrazení a flagy definující chování. Místo odečítání pozice z pixelů na obrazovce realizujete uvnitř programu fyzikální model, který počítá aktuální pozice a kolize s předměty; a zobrazení je jen projekce do aktuálního zobrazovacího zařízení v aktuálním rozlišení.