Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

PAZ1C

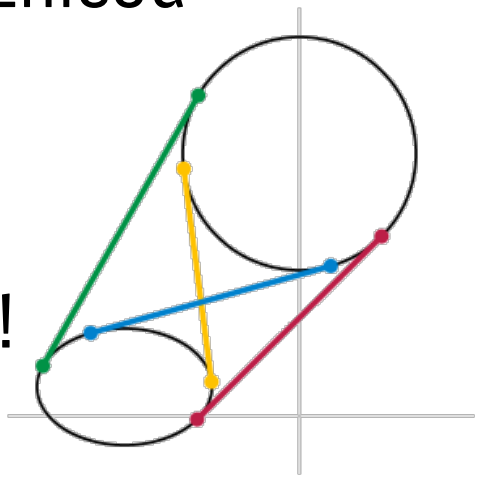
Róbert Novotný
robert.novotny@upjs.sk

Problémový prípad: kružnica vs elipsa

- notoricky známy problém pri návrhu:

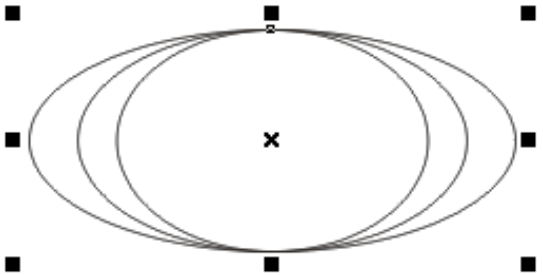
  kružnica-elipsa (alebo štvorec-obdĺžnik)

- overenie kompozície: „kružnica nemá elipsu", „elipsa nemá kružnicu" ➔ ***kompozícia je nezmysel***
- overenie dedičnosti: „každá elipsa je kružnicou"
  - nedáva zmysel
- overenie dedičnosti naopak: „každá kružnica je elipsou" ➔ **dáva zmysel**, ale!



Zamyslime sa nad kontraktom

- dodajme do kontraktu schopnosť naťahovať sa do šírky
- výška sa musí zachovať

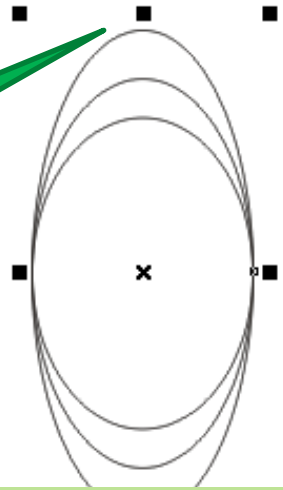


Obrázok z vektorového editora. Ťahaním za držadlo môžeme zväčšovať šírku so zachovaním výšky

```
class Elipsa {
  ...
  void zmeňPolosE(int dĺžka) {…}
}
```

Zamyslime sa nad kontraktom

- do kontraktu navyše dajme schopnosť naťahovať sa do **výšky**
- šírka sa musí zachovať



Obrázok z vektorového editora. Ťahaním za držadlo môžeme zväčšovať výšku so zachovaním šírky

```
class Elipsa {
  void zmeňPolosE(int dĺžka) {…}
  void zmeňPolosF(int výška) {…}
}
```

Kružnica vs elipsa


```
class Elipsa {
  void zmeňPolosE(int dĺžka) {…}
  void zmeňPolosF(int dĺžka) {…}
}
```

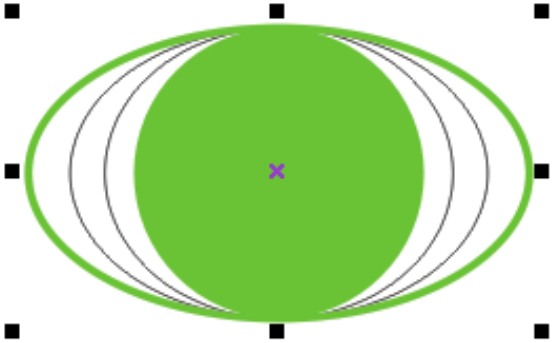
```
class Kružnica extends Elipsa {
  // zdedia sa metódy pre polosi
}
```

```
class Kružnica extends Elipsa {
  void zmeňPolosE(int dĺžka) {…}
  void zmeňPolosF(int dĺžka) {…}
}
```

- potrebujeme prekryť metódy pre polosi

Prekrytie metód v elipse – možnosť 1

- metódy neprekryjeme, priamo ich zdedíme



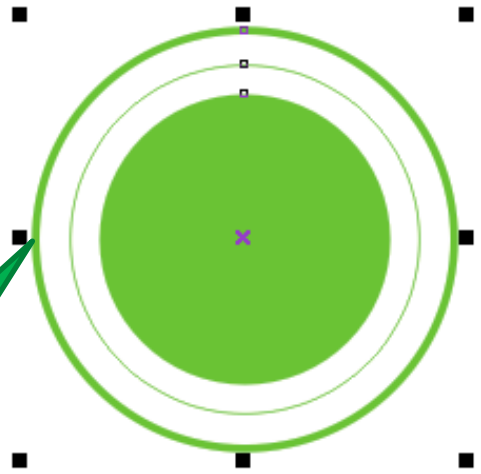
- lenže tým môžeme z kružnice spraviť elipsu
- budeme mať objekt typu **Kružnica**, ktorý nebude kružnicou

„to je dosť blbé"

Prekrytie metód v elipse – možnosť 2

- prekryjeme metódy tak, aby dodržala „kružnicovosť"
- teda so zmenou veľkosti jednej polosi zmeníme aj veľkosť druhej polosi
- lenže...
  - ťahaním za držadlo šírky spôsobíme aj ťahanie výšky!
  - užívateľ je v šoku!
- a zároveň...

Nedodržali sme kontrakt!



Nelogickosť v kóde

PAZ1C

```
Elipsa elipsa = new Elipsa();
elipsa.setPolosE(2);
elipsa.setPolosF(4);
System.out.println(elipsa.getPolosE());
System.out.println(elipsa.getPolosF());
```

```
2
4
```

```
Elipsa kružnica = new Kružnica();
kružnica.setPolosE(2);
kružnica.setPolosF(4);
System.out.println(kružnica.getPolosE());
System.out.println(kružnica.getPolosF());
```

```
2
2
```

Elipsa sa správa polymorfne, ale nastávajú nečakané vedľajšie efekty!

Nedodržanie kontraktu

- ak v kontrakte **Elipsy** povieme, že naťahovanie do šírky zachová výšku, musí to platiť aj v podtriedach
- **Kružnica** však tento kontrakt nevie dodržať.
- Dedičnosť nemá zmysel!

Ďalšie problémy

- kružnica však nepridáva žiadnu špeciálnu schopnosť
- práve naopak: kružnica je obmedzením elipsy
  - „kružnica je elipsa, ktorej polosi majú rovnakú dĺžku"
- elipsa potrebuje viac stavov než kružnica
  - elipsa: dve premenné (polos *e*, polos *f*)
  - kružnica: stačí jedna (*priemer*)

Zásada!
Oddedená trieda by mala ponúkať
správanie rodiča plus niečo navyše.

Liskovovej substitučný princíp (1987)

Ak pre každý objekt o1 typu T1 existuje objekt o2 typu T2 taký, že pre všetky programy P využívajúce T2 platí, že po nahradení objektu o2 objektom o1 sa správanie P nezmení, potom T1 je podtypom T2

- ak v programe nahradíme triedu podtriedami, správanie sa zachová.
- ak nahradíme inštancie elíps inštanciami kružníc, správanie sa zrejme poruší

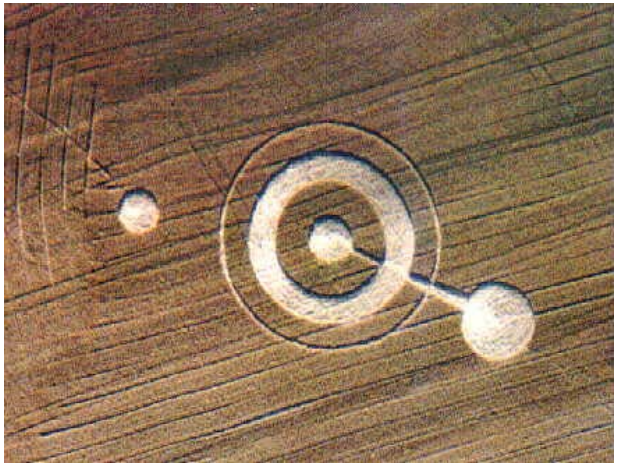
Liskovovej substitučný princíp (1987)

- v LSP je vágny pojem „správanie"
  - musíme sami určiť, čo znamená, že „správanie sa nezmení"
  - typicky: dodržanie kontraktu
- jedna vec je dedičnosť na **syntaktickej** úrovni
  - dodržiavanie hlavičiek metód
- druhá vec je **sémantika dedičnosti** (zmysel), teda *funkčnosť*, teda *správanie*
- `Kružnica extends Elipsa` je korektná v matematickom zmysle, ale v zmysle OOP sa nedá namodelovať
  - majú totiž odlišné **správanie** (a OOP je hlavne o správaní)

Problém je v meniteľnosti

- problém spočíva v **meniteľnosti inštancií**
  - keby sme zabránili zmene atribútov inštancie, problém by sa vyriešil
  - raz vytvorená kružnica bude navždy kružnicou

Riešenie: zrušíme settre, atribúty možno nastaviť *len* v konštruktore.



Problém je v narušení zapúzdrenosti

- problém spočíva v narušení zapúzdrenosti
  - oddedená trieda môže **meniť** interný stav rodičovskej triedy priamo

Riešenie: zrušíme hierarchiu **Kružnica – Elipsa**. Budú to dve nezávislé triedy bez akéhokoľvek vzťahu.

Formálnejšie zásady pre Liskovovej princíp

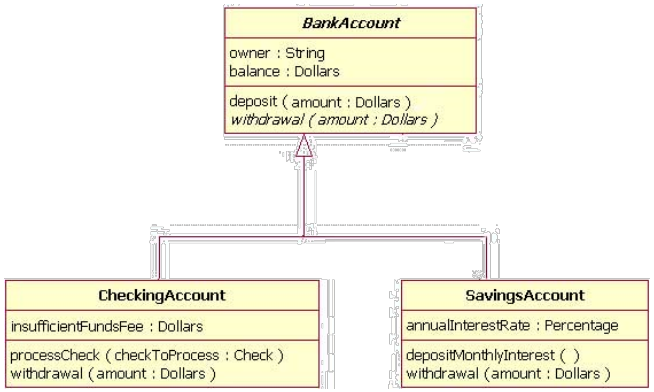
- **prepodmienky** (preconditions) **nemožno** v podtriede **zosilniť**
  - v podtriede **Kružnica** zrazu vyhlásime, že polos *e* sa musí rovnať polosi *f*
- **postpodmienky** (postconditions) **nemožno** v triede **zoslabiť**
  - v triede **Elipsa** platí v metóde zmeňPolosF() postpodmienka `novápolosE == predošlápolosE`
  - lenže metóda zmeňPolosF() v **Kružnici** túto podmienku zoslabuje
- **invarianty** musia ostať nezmenené
  - tvrdenie platné počas celého behu programu
  - „veľkosť multimnožiny nikdy nepresiahne *x*"

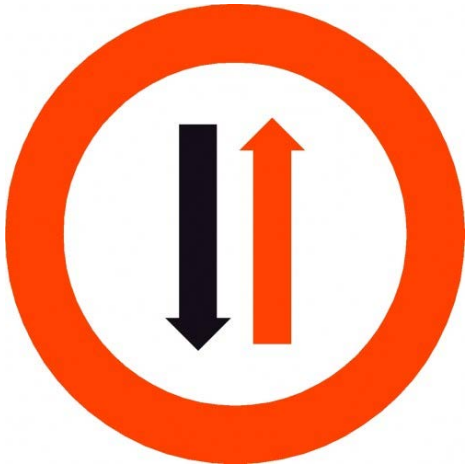
Ďalšia zásada pre Liskovovej princíp

- musí platiť **„history constraint (rule)"**

Ak máme stav predka, ktorý nemožno meniť settermi, potomok si nemôže dodať settery, ktoré toto obmedzenie zrušia.

- v opačnom prípade vieme narušiť invariant





Zásady pre Liskovovej princíp

Ak máme stav predka, ktorý nemožno meniť settermi, potomok si nemôže dodať settery, ktoré toto obmedzenie zrušia.

```
class PevnýBod {
  private int x, y;
  PevnýBod(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class Bod extends PevnýBod
  void setX(int x) {…}
  void setY(int y) {…}
}
```

Metóda narušuje *history rule*, umožňuje modifikovať rodičovský stav.

Iný príklad narušenia Liskovovej princípu

```
class Účet {
  int stav = 100;
  boolean uzatvor() {
    return (stav > 100);
  }
  ...
}
```

```
class TermínovanýÚčet
   extends Účet
{
  boolean uplynulaPerióda;
  boolean uzatvor() {
    return (stav > 100)
      && uplynulaPerióda;
  }
```

false

```
Konto k = new Konto();
Konto tk = new TermínovanéKonto();
k.uzatvor();
tk.uzatvor();
```

termínované konto sa má správať rovnako ako bežné konto! ***Narušenie LP***

Liskovovej princíp je hrôza!

- ale veď to úplne popiera dedičnosť!
- prečo som sa to učila, keď je to zbytočne?
  - smrť dedičnosti! naspäť k Pascalu!
- **dedičnosť nie je zlá**, len ju treba používať s rozvahou

Základný kritický bod dedičnosti
Dedičnosť narúša zapúzdrenie!
(Inheritance breaks encapsulation!)

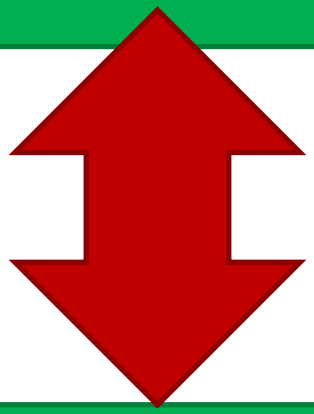
Dedičnosť narúša zapúzdrenie

- Zapúzdrenie = **skrývanie informácií**
  - trieda je čierna skrinka
  - skrýva implementačné detaily
- Klient uzatvára **kontrakt** s triedou, ktorý je reprezentovaný verejnými metódami
- Oddedená trieda je však tiež len klient rodičovskej triedy a má sa riadiť kontraktom

Dve protibežné zásady

Dedičnosť

Trieda zdedí správanie a stav od rodiča



Zapúzdrenie

Trieda môže meniť stav inej triedy len skrz špecifikované metódy

Ešte jeden problém s dedičnosťou

PAZ1C

- vieme, že dedičnosť narúša zapúzdrenie
- dedičnosť **vyžaduje** znalosti o správaní rodičovskej triedy

Riešenie

Uprednostňujte kompozíciu pred dedičnosťou

Dizajn = balansovanie

- Návrh tried = balansovanie medzi dvoma zásadami
- Teoretický ideálny stav
  - všetky inštančné premenné sú privátne
  - prístup k stavu rodiča len cez verejné metódy
  - všetci dodržiavajú Liskovovej princíp
- V praxi: **ťažko dosiahnuteľné**
- Kvôli jednoduchosti sa zavádzajú konštrukcie, ktoré pri správnom používaní vedú ku kompromisu
  - ***protected*** premenné a metódy

Otvorené a zatvorené

Open-Closed Principle
Triedy majú byť otvorené voči rozširovaniu, ale uzavreté voči zmenám.

-- „Bertrand Meyer – Object Oriented Software Construction (1988)"

Open-Closed Principle

Softvérový modul má byť otvorený voči rozširovaniu, ale uzavretý voči zmenám.

- **open for extension** (otvorené voči rozširovaniu)
  - možno rozširovať funkcionalitu
  - pridávať nové správanie / stav
- **closed for modification** (uzavreté voči zmenám)
  - zdrojový kód modulu nemožno meniť
  - jediný výnimka: oprava chýb

Open-Closed Principle

- novú funkcionalitu možno dosiahnuť cez
  - pridávanie nových tried
  - dedičnosť
  - prekrývanie
  - znovupoužitie existujúcich tried
- toto je ideálna predstava, často to nie je možné

Strašne zlý hudobný príklad


```
class Hudobník {
  void zahraj(Gitara g) {
    System.out.println("Tidli-fidli");
  }
  void zahraj(Trúba t) {
    System.out.println("Tururu");
  }
}
```

- ak chceme, aby hudobník vedel hrať na nový nástroj

Strašne zlý hudobný príklad


```
class Hudobník {
  void zahraj(Nástroj n) {
    if("gitara".equals(n.getTyp()) {
      System.out.println("Tidli-fidli");
    }
    if("trúba".equals(n.getTyp()) {
       System.out.println("Tururu");
    }
  }
}
```

Čo ak chceme naučiť hrať hudobníka na nový nástroj?

```
class Nástroj {
  private String typ;
  public Nástroj(String typ) { this.typ = typ; }
  // gettre, settre…
}
```

Strašne zlý hudobný príklad

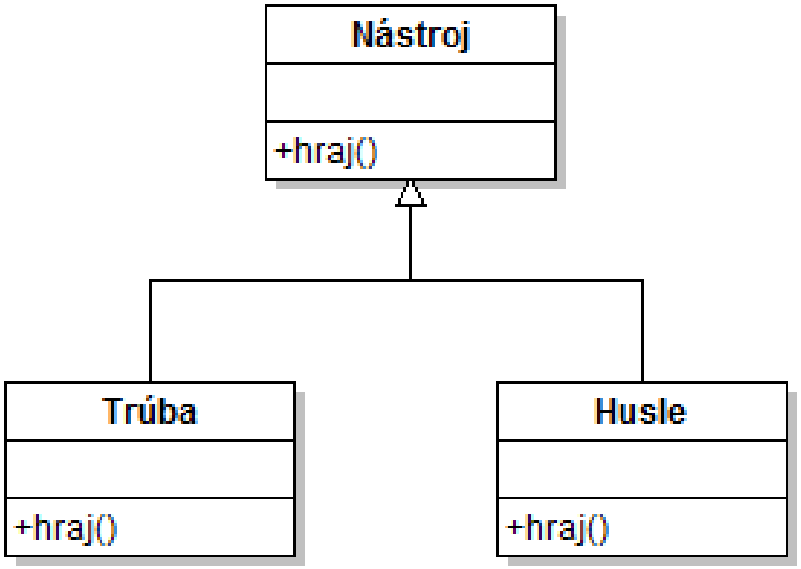

```
class Hudobník {
  void zahraj(Nástroj n) {
    if("gitara".equals(n.getTyp()) {
      System.out.println("Tidli-fidli");
    }
    if("trúba".equals(n.getTyp()) {
       System.out.println("Tururu");
    }
  }
}
```

- jediná možnosť rozšírenia
  - oddediť
  - skopírovať celú metódu
  - dodať ďalší cyklus

Trieda je uzavretá voči modifikácii.

Postupnosť vylepšení

- Namiesto odlíšenia typu nástroja pomocou stavovej premennej **zavedieme hierarchiu dedičnosti**
- Nástroj bude sám vedieť, ako sa na ňom hrá




```
class Husle extends Nástroj {
  String hraj() {
    return "Tidli-fidli";
  }
}
```

Postupnosť vylepšení

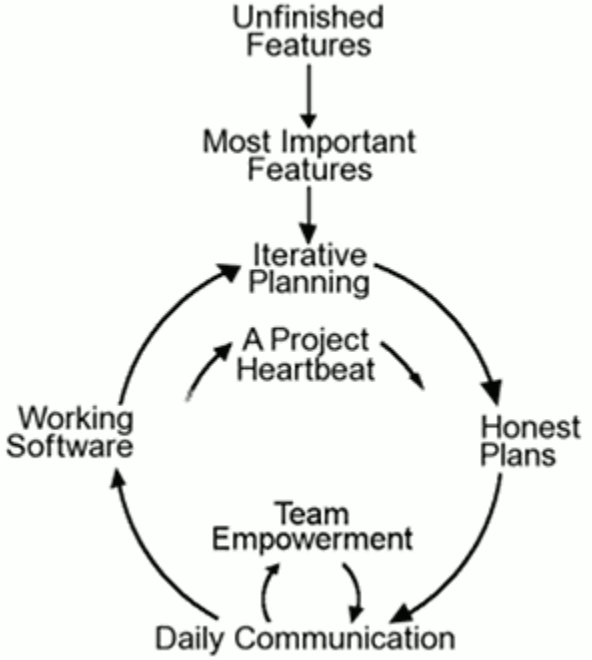

```
class Hudobník {
  void zahraj(Nástroj n) {
    System.out.println(n.hraj());
  }
}
```

```
Hudobník oldfield = new Hudobník();
h.zahraj(new Gitara());
h.zahraj(new Husle());
```

- trieda je otvorená voči zmenám
- dodanie schopnosti hrať na nový nástroj
  - dodanie triedy dediacej od Nástroja
  - zavolanie metódy **zahraj()**

Návrh tried = diktátorstvo s ľudskou tvárou

- pri návrhu tried treba identifikovať miesta, ktoré budú chcieť programátori modifikovať
- prípadne kde je priestor na doda funkcionality
- toto chce často **prax/cit/predvídavosť**
- možno sa riadiť niekoľkými zásadami



Zásady pre návrh

1. Je lepšie všetko na začiatku obmedziť a potom postupne povoľovať, ako naopak.
2. Urobte tú najjednoduchšiu vec, ktorá vyzerá, že bude fungovať.

   „Vo chvíli, keď dostaneme niečo na obrazovku, vieme sa na to pozrieť. A ak potrebujeme niečo viac, vieme to tam vždy dodať." (Kent Beck)

3. YAGNI (You Ain't Gonna Need It)

   *„Veci implementujte len vtedy, ak ich **naozaj** potrebujete a nie vtedy, ak len **predvídate**, že ich budete potrebovať."*