

Programovanie, algoritmy, zložitosť / ÚINF/PAZ1C

PAZ1C

Róbert Novotný
robert.novotny@upjs.sk

27. 10. 2010

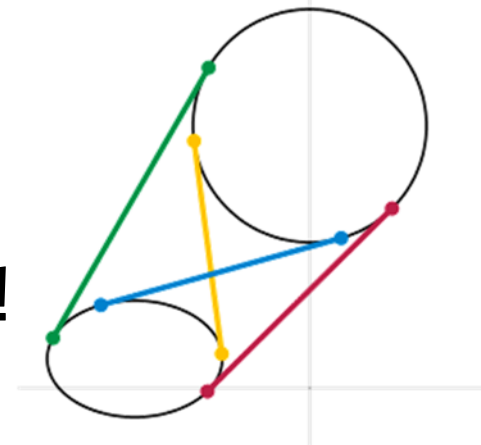
Problémový prípad: kružnica vs elipsa

PAZ1C

- 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!

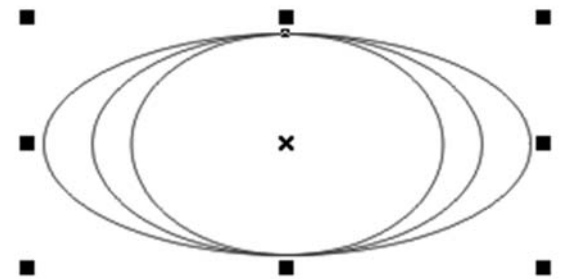


Zamyslime sa nad kontraktom

PAZ1C

- 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



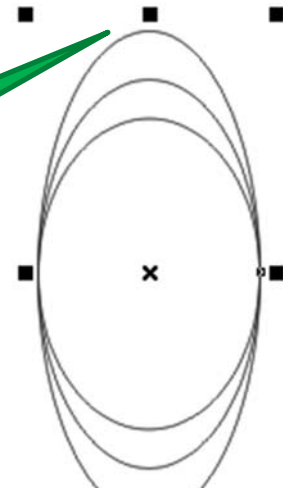
```
class Elipsa {  
    ...  
    void zmenPolosE(int dĺžka) {...}  
}
```

Zamyslime sa nad kontraktom

PAZ1C

- do kontraktu navyše dajme schopnosť natahovať 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

PAZ1C

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

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

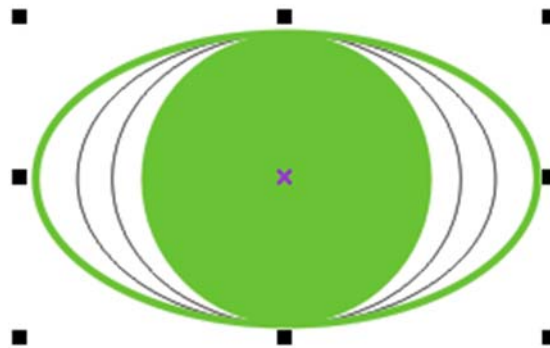
```
class Kružnica extends Elipsa {  
    void zmeňPolosE(int dĺžka) {...}  
    void zmeňPolosF(int dĺžka) {...}  
}
```

- potrebujeme prekryť metódy pre polosy

Prekrytie metód v elipse – možnosť 1

PAZ1C

- metódy neprekryjeme, priamo ich zdedíme



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

„to je dosť blbé“

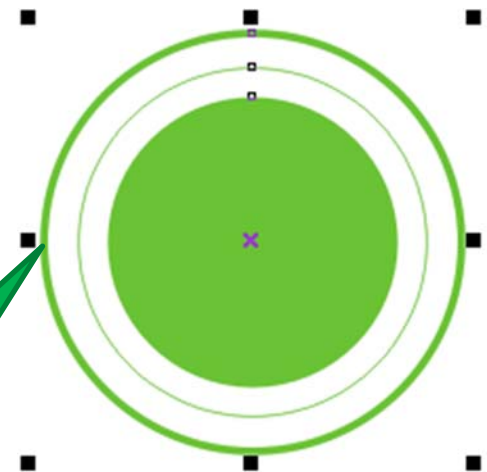
Prekrytie metód v elipse – možnosť 2

PAZ1C

- 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!

Ťahaním za držadlo sa zväčšuje šírka i výška!



Nelogickosť v kóde

PAZÍC

```
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

PAZ1C

- 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

PAZ1C

- kružnica však nepridáva žiadnu špeciálnu schopnosť
- práve naopak: kružnica je obmedzením elipsy
 - „kružnica je elipsa, ktorej polosy 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)

PAZ1C



Ak pre každý objekt o_1 typu T_1 existuje objekt o_2 typu T_2 taký, že pre všetky programy P využívajúce T_2 platí, že po nahradení objektu o_2 objektom o_1 sa správanie P nezmení, potom T_1 je podtypom T_2

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

Liskovovej substitučný princíp (1987)

PAZ1C

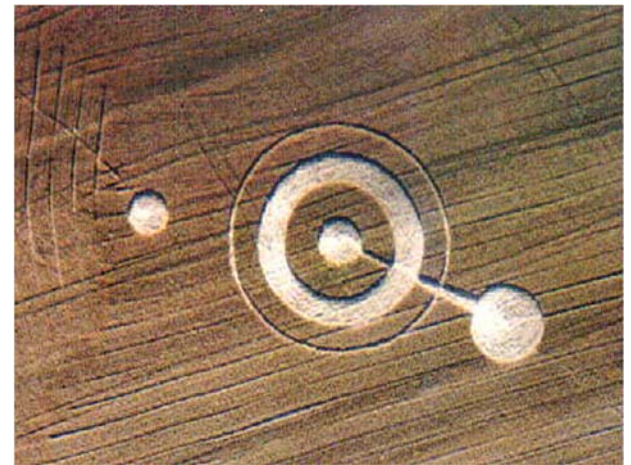
- v LSP je vágny pojem „správanie“
 - musíme sami určiť, čo znamená, že „správanie sa nezmení“
 - typicky: dodržanie kontraktu
- jedna vec je dedičnosť na **syntaktickej** úrovni
 - dodržiavanie hlavičiek metód
- druhá vec je **sémantika dedičnosti** (zmysel), teda *funkčnosť*, teda *správanie*
- Kružnica 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

PAZ1C

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

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



Problém je v narušení zapúzdrenosti

PAZ1C

- 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

PAZ1C

- **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 “

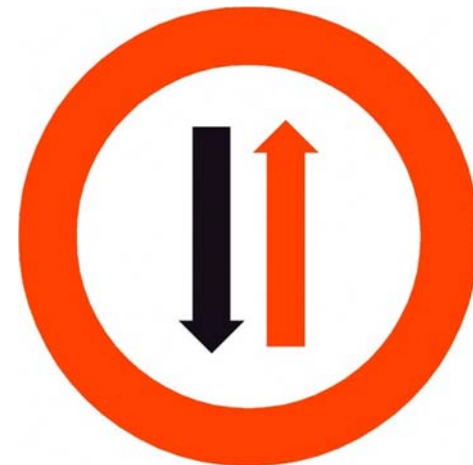
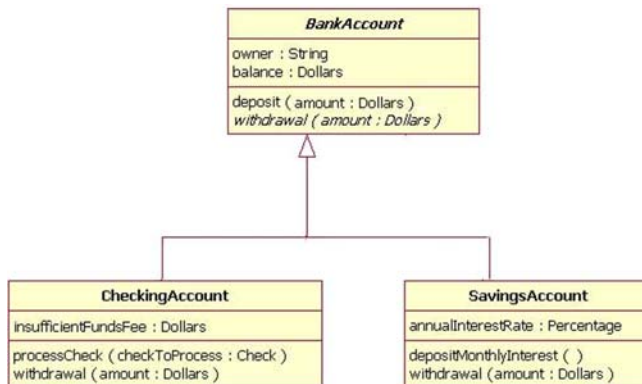
Ďalšia zásada pre Liskovovej princíp

PAZ1C

- 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

PAZ1C

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

PAZ1C

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

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

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!

PAZ1C

- ale ved' 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 zapúzdenie!
(Inheritance breaks encapsulation!)

Dedičnosť narúša zapúzdrenie

PAZ1C

- 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

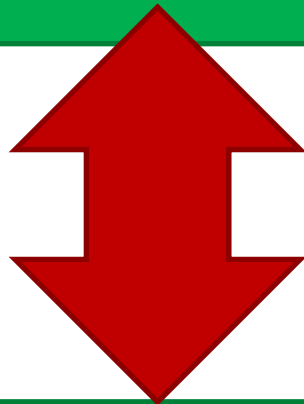


Dve protibežné zásady

PAZ1C

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

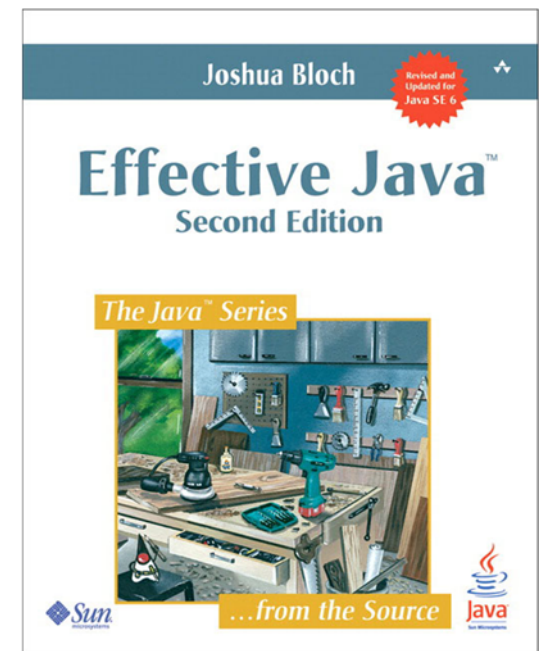
Riešenie

Uprednostňujte kompozíciu pred dedičnosťou

Kolekcie a ich tragédia

PAZ1C

- Joshua Bloch – *Effective Java*, 2. vydanie (2008)
 - krátka, ale geniálna kniha o zákutiach Javy
- príklad:
 - urobme množinu, ktorá si bude evidovať počet pridaných prvkov
 - „do tejto množiny bolo pridaných X prvkov“



Návrh kódu a logika uvažovania

PAZ1C

- oddedíme z **HashSet**
- prekryjeme metódu **add()**
 - pripočítame jednotku
 - zavoláme rodičovskú metódu, ktorá pridá prvok
- musíme prekryť aj **addAll()** (pridanie kolekcie do množiny)
 - pripočítame toľko, koľko je prvkov v množine
 - zavoláme rodičovskú metódu, ktorá pridá prvky

Návrh kódu a logika uvažovania

PAZ1C

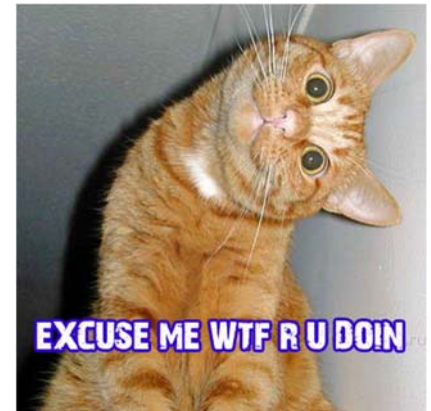
```
public class InstrumentedHashSet<E> extends HashSet<E> {  
  
    private int početPridaní = 0;  
  
    public boolean add(E e) {  
        početPridaní++;  
        return super.add(e);  
    }  
  
    public boolean addAll(Collection<? extends E> c) {  
        početPridaní += c.size();  
        return super.addAll(c);  
    }  
}
```

Použitie triedy

PAZ1C

```
Set<String> s = new InstrumentedHashSet<String>();  
s.addAll(Arrays.asList("Baldrick", "Edmund", "Queenie"));
```

- lenže ak zistíme počet pridaných prvkov, zistíme, že máme výsledok **6**
- **prečo?** nik nevie
- pozrieme do zdrojákov!
 - ešteže ich Sun zverejňuje...
- vinník: metóda **addAll()** v **java.util.AbstractCollection**



Návrh kódu a logika uvažovania

PAZ1C

```
public boolean addAll(Collection<? extends E> c) {  
    boolean modified = false;  
    Iterator<? extends E> e = c.iterator();  
    while (e.hasNext()) {  
        if (add(e.next())) modified = true;  
    }  
    return modified;  
}
```

Metóda **addAll()** volá metódu **add()**!
Hm!

- započíta sa to dvakrát – raz v prekrytej metóde **addAll()**, ktorá zavolá našu prekrytú metódu **add()**

Čuduj sa svete, dokumentácia!

PAZÍC

addAll

```
public boolean addAll(Collection c)
```

Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)

This implementation iterates over the specified collection, and adds each object returned by the iterator to this collection, in turn.

Note that this implementation will throw an `UnsupportedOperationException` unless `add` is overridden (assuming the specified collection is non-empty).

Specified by:

[addAll](#) in interface [Collection](#)

Parameters:

`c` - collection whose elements are to be added to this collection

Returns:

`true` if this collection changed as a result of the call.

Throws:

[UnsupportedOperationException](#) - if this collection's `addAll` method does not support this operation.

[NullPointerException](#) - if the specified collection is null.

See Also:

[add\(Object\)](#)

Z dokumentácie možno odvodiť toto správanie.

Čo ak dokumentácia nie je?

Návrh kódu a logika uvažovania

PAZ1C

```
public boolean addAll(Collection<? extends E> c) {  
    boolean modified = false;  
    Iterator<? extends E> e = c.iterator();  
    while (e.hasNext()) {  
        if (add(e.next())) modified = true;  
    }  
    return modified;  
}
```

Metóda **addAll()** volá metódu **add()**!
Hm!

- započíta sa to dvakrát – raz v prekrytej metóde **addAll()**, ktorá zavolá našu prekrytú metódu **add()**

Dedičnosť je bezpečná, ak...



- dedíte od tried v rovnakom balíčku
- máte dosah na implementáciu rodičovských tried
- dedíte od tried, ktoré boli explicitne navrhnuté na oddedenie a majú o tom dokumentáciu
- implementujete interfejs alebo tvoríte interfejs, ktorý dedí od iného interfejsu

Dediť od náhodných tried krížom cez balíčky môže viesť k nečakaným problémom!

Riešenie problému

PAZ1C

- zrušiť pripočítavanie v metóde `addAll()`
 - lebo sme si prečítali dokumentáciu
 - lenže my sa spoliehame na implementačný detail, ktorý sa môže zmeniť, a potom máme problém
- iné riešenie: prekryť si metódu `addAll()` po svojom
 - prelez kolekciu, pridaj prvok cez (prekrytú) metódu `add()`
 - lenže časom môžeme zistiť, že kopírujeme kód z metódy rodičovskej triedy
- rozumné riešenie je použiť **delegáciu** (tretí spôsob skladania tried)

Nástrel delegácie

PAZÍC

```
public class InstrumentedSet<E> implements Set<E> {  
    private final Set<E> s;  
  
    public InstrumentedSet(Set<E> s) { this.s = s; }  
  
    public void clear() { s.clear(); }  
  
    public boolean contains(Object o) {  
        return s.contains(o);  
    }  
    public boolean isEmpty() {  
        return s.isEmpty();  
    }  
    // .. ďalšie metódy  
}
```

objekt, na
ktorý budeme
delegovať
volania

v metóde delegujeme
volanie na príslušný
objekt

Nástrel delegácie - pokračovanie

PAZ1C

```
public class InstrumentedSet<E> implements Set<E> {  
    // private final Set<E> s;  
    // pokračovanie z predošlého snímku  
  
    private int početPridaní = 0;  
  
    public boolean add(E e) {  
        početPridaní++;  
        return s.add(e);  
    }  
  
    public boolean addAll(Collection<? extends E> c) {  
        početPridaní += c.size();  
        return s.addAll(c);  
    }  
}
```

V metóde delegujeme volanie na príslušný objekt. Je to analógia **super**.

Použitie triedy

PAZ1C

```
Set<String> set  
    = new InstrumentedSet<String>(new HashSet<String>());
```

- množina bude delegovať volania na objekt typu **HashSet**
- množina **set** sa správa v metódach **add()** a **addAll()** podľa nášho želania, ostatné metódy sú delegované na **HashSet** v parametri konštruktora



Výhody a nevýhody delegácie

PAZ1C

- Simulujeme dedičnosť
- Nemusíme sa spoliehať na neznáme správanie rodičovskej triedy
- Môžeme dodať funkcionality pre počítanie pridaných prvkov do ľubovoľnej triedy, ktorá implementuje **Set**
- Nevýhody:
 - treba vygenerovať kód pre delegovanie
 - rozumné IDE to spraví za nás
 - kód vyzerá na pohľad nepríjemne, ale je len zjavný

Dizajn = balansovanie

PAZ1C

- 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 dosiahnutelné**
- Kvôli jednoduchosti sa zavádzajú konštrukcie, ktoré pri správnom používaní vedú ku kompromisu
 - *protected* premenné a metódy

Otvorené a zatvorené

PAZ1C

Open-Closed Principle

Triedy majú byť otvorené voči rozširovaniu, ale uzavreté voči zmenám.



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

Open-Closed Principle

PAZ1C

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

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

Open-Closed Principle

PAZ1C

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



Strašne zlý hudobný príklad

PAZ1C

```
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

PAZIC

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

Čo ak chceme naučiť hrať hudobníka na nový nástroj?

Strašne zlý hudobný príklad

PAZ1C

```
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");  
        }  
    }  
}
```

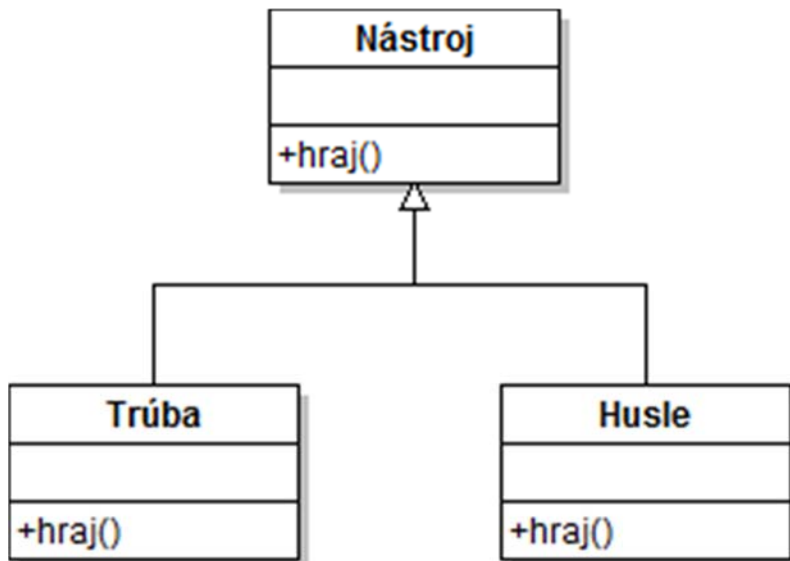
- jediná možnosť rozšírenia
 - oddediť
 - skopírovať celú metódu
 - dodať ďalší cyklus

Trieda je uzavretá voči modifikácii.

Postupnosť vylepšení

PAZ1C

- 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

PAZ1C

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

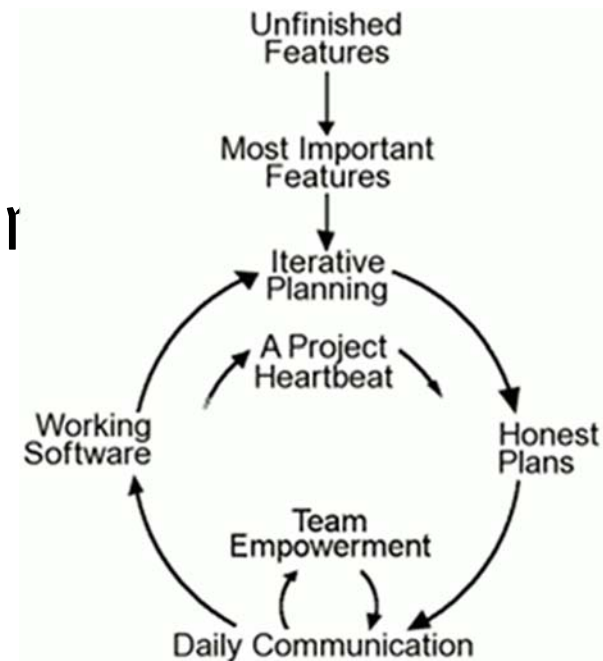
```
Hudobník oldfield = new Hudobník();  
h.zahraj(new Gitara());  
h.zahraj(new Husle());
```

- 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

PAZ1C

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



Zásady pre návrh

PAZ1C

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.“*