<embed/it>

Úvod do objektového návrhu

PV168
23. 2. 2016 & 28.2.2016
Petr Adámek

Obsah

- Organizace předmětu
- Osnova předmětu
- Nejdůležitější pravidlo
- Rekapitulace ostatních důležitých zásad
- Úvod do objektové dekompozice

Organizace a osnova předmětu

- **Organizace předmětu**
  - https://is.muni.cz/auth/el/1433/jaro2017/PV168/op/Organizace.html
- **Osnova předmětu**
  - https://is.muni.cz/auth/el/1433/jaro2017/PV168/index.qwarp

Nejdůležitější pravidlo

KISS

Základní pravidla pro tvorbu udržovatelného kódu

- Přečíst si Effective Java ☺
- Důsledně preferovat jednoduchý a přehledný kód, vyhýbat se předčasné optimalizaci
- Dodržovat konvence pro zápis kódu (např. identifikátory)
- Používat srozumitelné identifikátory
- Psát kód tak, aby nebyly potřeba komentáře
- Vyhýbat se duplicitnímu kódu
- Používat existující knihovny a Java Core API, nevynalézat znovu kolo
- Nezneužívat dědičnost, používat ji pouze pro modelování vztahu generalizace/specializace
- Používat vyjímky pouze pro ošetření chyb nebo nestandardních situací
- Být defenzivní, vždy kontrolovat dodržení kontraktu

Zdroje

Effective Java (2nd Edition)

Joshua Bloch

http://amazon.com/dp/0321356683/

Objektová dekompozice

- **Dekompozice je rozklad problému na dílčí podproblémy, které se pak řeší samostatně. V případě programování se jedná zejména o rozdělení programu do modulů, objektů a metod.**

- Dobrá dekompozice je důležitou součástí dobrého návrhu programu.

- **Princip *Rozděl a panuj.***

Jak se pozná dobrá dekompozice

- **Dobře provedená dekompozice:**
  - Je jednoduchá (vyhýbáme se zbytečně komplikovaným návrhům a neřešíme problémy, které řešit nemusíme).
  - Každá komponenta (třída nebo metoda) je co nejjednodušší (ale aby dávala smysl).
  - Mezi komponentami je co nejméně závislostí.
  - Neobsahuje duplicitní kód
  - Využívá výhody zapouzdření (skrývání složitosti).
  - Používá správně definovanou hierarchii výjimek.

Jak se pozná dobrá dekompozice

- **Další tipy pro dekompozici**
  - Pokud je to možné, používejte neměnitelné třídy.
  - Pro definici typů, služeb a potenciálně vyměnitelných nebo obecných komponent používejte rozhraní.
  - Mezi komponentami je co nejméně závislostí.
  - Omezte použití dědičnosti (často je vhodnější kombinace kompozice nebo agregace a delegování).
  - Klíčem je využití existujících komponent (knihovny, Java Core API, jakarta-commons, frameworky, apod.)

Jak na dekompozici

- **Snažíme se identifikovat jednotlivé části problému, které je vhodné modelovat pomocí tříd.**

- **U databázových aplikací obvykle narazíme na:**
  - třídy s charakterem entit (Student, Car, Course, apod.);
  - třídy reprezentující operace či sady operací (StudentCatalog, QueryParser, StorageManager).

Entity

- ***Entita*** **je třída reprezentující nějaký objekt, který existuje v problémové domény. Objekt může být konkrétní (auto, osoba, faktura) i abstraktní (úkol, dovednost, předmět).**

- **Každá entita by měla mít (pouze):**
  - ID (primární klíč);
  - příslušné atributy a k nim get/set metody;
  - bezparametrický konstruktor;
  - metody equals() a hashCode();
  - vhodná bývá i metoda toString(), případně compareTo().

Entity

- **Při vytváření dodržujeme následující zásady:**
  - ID (primární klíč) by mělo být syntetické, nevolíme atributy z problémové domény (např. rodné číslo, RZ).
  - Až na výjimky nevytváříme jiný než bezparametrický konstruktor.
  - Entita slouží jako schránka na data, nikdy nesmí obsahovat aplikační logiku.

Příklad entity

```
public class Person {

    private Long id;
    private String name;

    public Long getId()                { return id; }
    public void setId(Long id)         { this.id = id; }
    public String getName()            { return name; }
    public void setName(String name)  { this.name = name; }

    public boolean equals(Object o) {
        if (this == o) { return true; }
        if (getId() == null) { return false; }
        if (o instanceof Person) {
            return getId().equals(((Person) o).getId());
        } else {
            return false;
        }
    }

    public int hashCode() { return id==null?0:id.hashCode(); }
}
```

Závislosti mezi komponentami

- **Třída A závisí na třídě B pokud:**
  - Metody třídy A používají třídu nebo rozhraní B
  - Třída A rozšiřuje třídu B
  - Třída A implementuje třídu B
  - Potom třídu A nelze použít bez třídy B. Podobně to funguje i na ostatních úrovních (metody, komponenty, moduly, apod.)

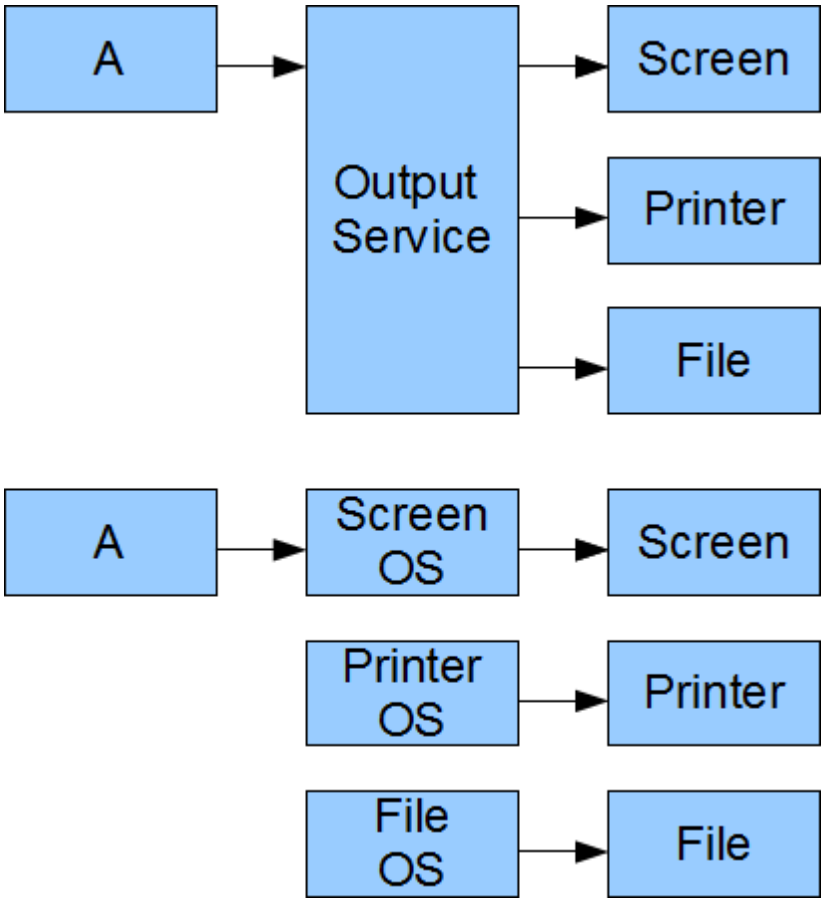
- **Závislosti mohou být tranzitivní**

Problémy se závislosti

- **Závislosti:**
  - komplikují údržbu kódu (změny jsou složité);
  - brání znovupoužitelnosti kódu;
  - indikují chybu v dekompozici (pokud je jich moc).

- **Zásadní chybou jsou cyklické závislosti:**
  - nikdy se nesmí objevit;
  - vždy jsou důsledkem špatné dekompozice;
  - vždy se dají odstranit.

Příklad problematických závislostí



Vazby mezi entitami

- **Co musíme vzít v úvahu:**
  - Druh vazby (asociace, agregace, kompozice).
  - Kardinalitu (1:1, 1:N, M:N)
  - Přítomnost referencí jako atributu v třídě entity (jednosměrné, obousměrné, bez přímé reference)

Přítomnost referencí v entitě

Ne vždy je dobrý nápad do třídy entity vkládat atributy pro všechny vazby, které daná entita má. Můžeme se dostat do situace, kdy kvůli tranzitivním závislostem budeme zbytečně načítat velké množství nepotřebných dat.

Proto je někdy užitečné modelovat vazbu jako jednosměrnou (je přístupná pouze z jedné entity a z druhé nikoliv), nebo atributy pro přístup k asociovaným entitám úplně vypustit. Přístup k asociovaným entitám pak zajistíme pomocí vhodné metody v servisní vrstvě.

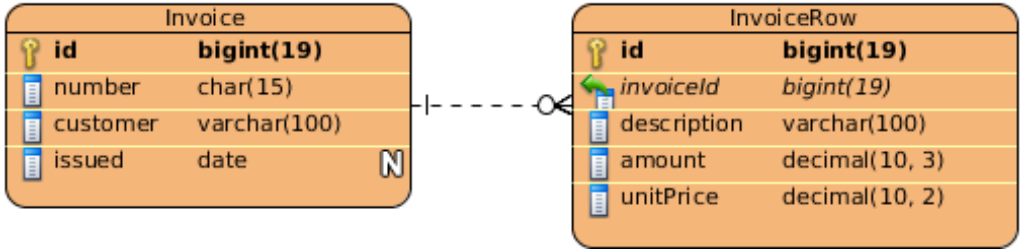
Jak řešíme kardinalitu

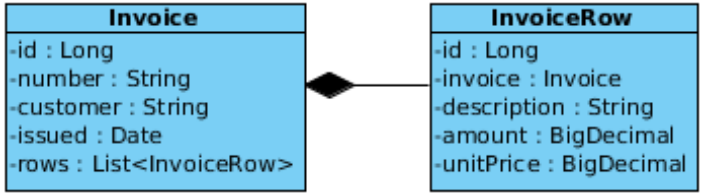
- **Na úrovni datového modelu**
  - 1:N – do tabulky entity s kardinalitou N přidáme cizí klíč, který bude odkazovat na primární klíč druhé entity.
  - M:N – musíme přidat skrytou vazební tabulku nebo vazební entitu.
  - 1:1 – podobně jako pro vztah 1:N
  - Pokud je daná vazba nepovinná, u sloupce s příslušným cizím klíčem povolíme hodnotu NULL

Druh vazby

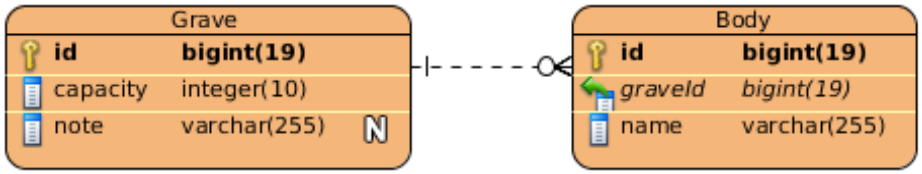
- Agregace i kompozice jsou speciálním případem asociace a proto se modelují a implementují stejným způsobem.
- Druh vazby však může být vodítkem pro to, zda zahrnout reference do entity.
- Pro rozhodnutí o jaký typ vazby se jedná vizte např. blogspot René Steina.

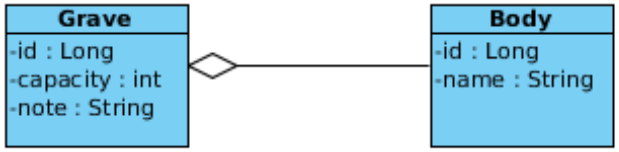
Kompozice, vztah 1:N, obousměrná vazba



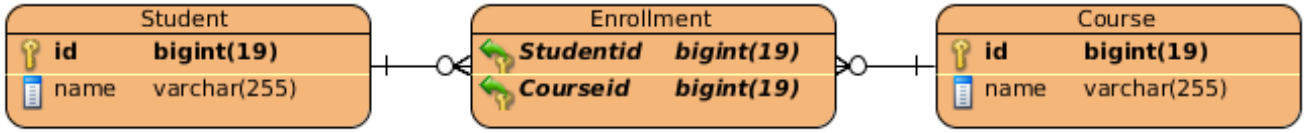


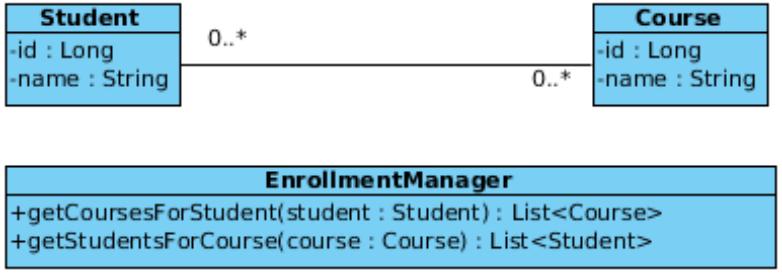
Agregace, vztah 1:N, bez referencí



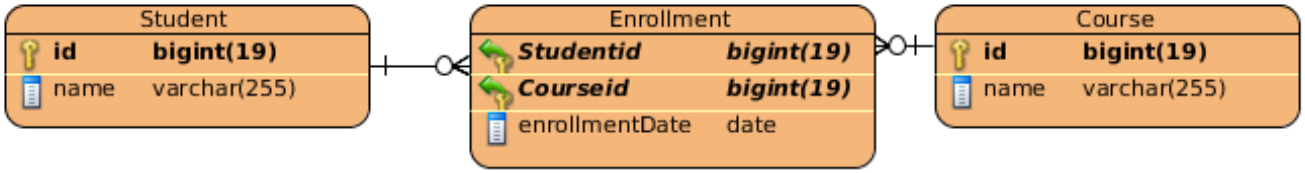


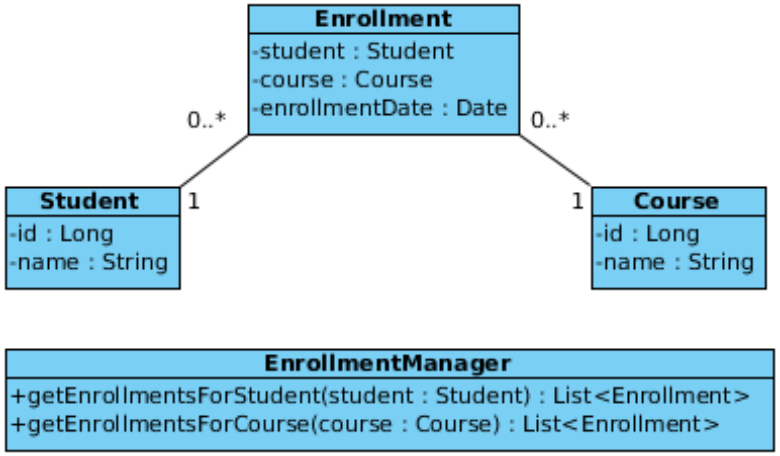
Asociace, vztah M:N, skrytá vazební tabulka, bez referencí





Asociace, vztah M:N, s vazební entitou, jednosměrné vazby





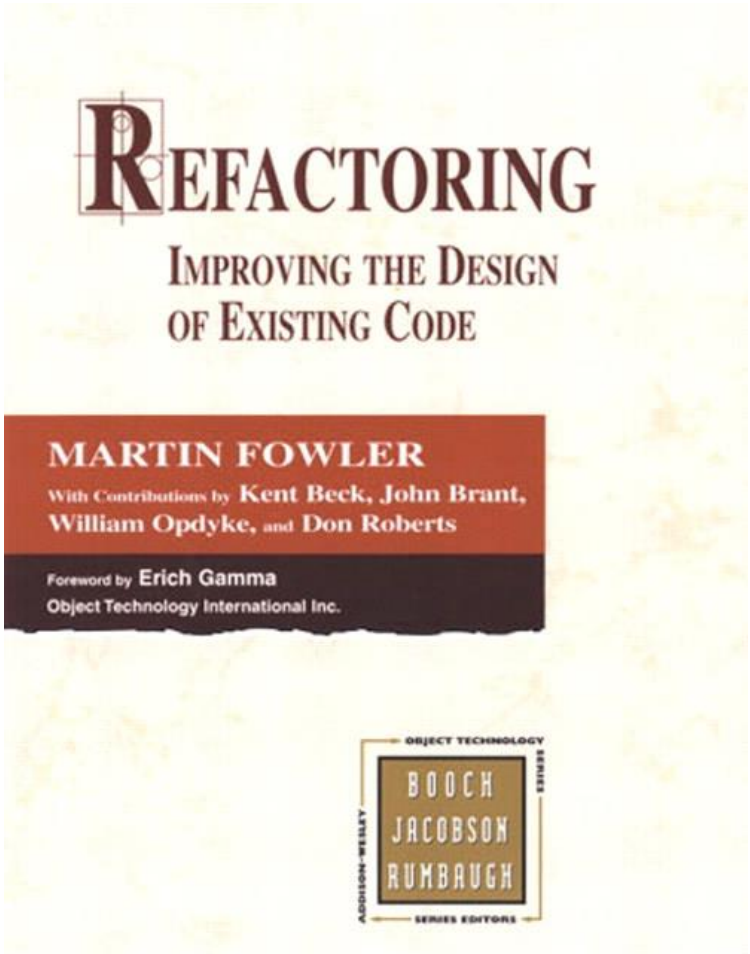
Refaktoring

- Refaktoring je změna struktury kódu, aniž by došlo ke změně funkčnosti.
- Málokdy se podaří vše navrhnout správně hned napoprvé, případně může dojít v průběhu vývoje ke změně či upřesnění požadavků. Proto je občas potřeba strukturu kódu změnit.
- Řada agilních metodik (TDD, XP, apod.) používá refaktoring jako jeden ze základních nástrojů.

Zásady při refaktoringu

- Při refaktoringu nikdy nepřidáváme novou funkcionalitu, pouze měníme strukturu kódu.
- Klíčem k úspěchu jsou jednotkové testy

Zdroje



Refactoring: Improving the Design of Existing Code

Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

http://amazon.com/dp/0201485672/

Závěr