1. Obecně o informačních systémech

1. Popište, co se rozumí pojmem informační systém, co řeší a uveďte příklady

•Informační systém je propojení informačních technologií a lidských aktivit směřující k zajištění podpory procesů v organizaci.

• V širším slova smyslu se jedná o interakci mezi lidmi, procesy a daty. Informační systém je určen ke zpracování (získávání, přenos, uložení, vyhledávání, manipulace, zobrazení) informací.

• Počítačové informační systémy poskytují podporu pro zvýšení efektivity práce s informacemi

• Snížení nákladů na lidskou práci, snížení chybovosti, zvýšení efektivity, automatizace

•Typy agend: Ekonomická • Personální • Skladová • Dokumentová

2. Popište, co se rozumí pojmem doména informačního systému a uveďte příklady.

•Doménou rozumíme skupinu souvisejících „věcí" z pohledu zákazníka či uživatele.

• Do domény patří data, procesy, uživatelé apod., které jsou charakterizovány specifickými pojmy.

• Používáním těchto pojmů určujeme, v jaké doméně se pohybujeme (např. bankovnictví používáme pojmy jako účet, transakce…)

3. Na jaké otázky si musíme odpovědět při realizaci informačního systému? Uveďte příklady odpovědí na jednotlivé otázky v kontextu nějakého konkrétního informačního systému.

• **CO?** Jde o informace ve smyslu dat.

• **JAK?** Jde o procesy prováděné s informacemi.

• **KDE?** Na jakých místech se pracuje s informacemi.

• **KDO?** Kdo, v jaké roli a v jakém kontextu pracuje s informacemi.

• **KDY?** Kdy, s jakými událostmi a na základě jakých impulzů se s informacemi pracuje

• **PROČ?** Jde o cíle a pravidla, jak těchto cílů dosáhnout.

Příklad:

Co? Systém pro realizaci výpůjček v půjčovně kol.

Jak? Uživatel po registraci a přihlášení si může vytvořit výpůjčku. Systém kontroluje dostupnost kol.

Kde? Přístup bude pomocí webové aplikace. Data budou ukládaná do MSSQL databáze.

Kdo? Uživatel si může vytvořit výpůjčku. Správce může rovněž vytvořit výpůjčku, a navíc schvaluje výpůjčky od Uživatele.

Kdy? Nejčastěji bude impulz od Uživatele při vytvoření výpůjčky.

Proč? Systém má zjednodušit a usnadnit, některé kroky jako jsou ručně psané záznamy. Jde hlavně o to ušetřit čas a peníze.

4. Z jakých pohledů (míry abstrakce) se můžeme dívat na informační systém? A v jakých rolích?

• **Rámec, rozsah, vize** (strategické rozhodnutí, vizionář, stratég)

• **Byznys model** (osoba odpovědná za proces, vlastník)

• **Systémový model** (architekt, návrhář)

• **Technologický model** (návrhář, technolog)

• **Detailní reprezentace** (specialista, programátor)

• **Provoz** (vyčleněný pracovník, správce, uživatel)

5. Co se rozumí architekturou informačního systému a co zahrnuje?

• Základní organizace softwarového systému zahrnující jeho **komponenty**, jejich vzájemné **vztahy** a **vztahy s okolím systému**, principy **návrhu** takového systému a jeho **vývoj.**

• Architektura informačního systému leží na vyšší úrovni abstrakce tak, že zahrnuje

•pohled na aplikační doménu (tj. „pohled zákazníka"),

•pohled vývojáře na globální strukturu systému a chování jeho částí, jejich propojení a synchronizace,
•pohled na přístup k datům a toky dat v systému,
• fyzické rozmístění komponent a další.

6. Jaké jsou rozdíly mezi statickou, dynamickou a mobilní architekturou informačního systému? Uveďte příklady.
   • **statická architektura** – Zachycuje pouze pevnou strukturu softwarového systému bez možnosti změn. Je daná návrhem a nemění se v průběhu běhu programu.
   • **dynamická architektura** – Oproti statické podporuje i dynamický vznik a zánik komponent a vazeb za běhu systému dle pravidel daných návrhem.
   • **mobilní architektura** – Rozšiřuje dynamickou architekturu o mobilní prvky, které přesouvají komponenty a vazby za běhu systému podle stavu výpočtu.
7. Co obsahuje struktura každého informačního systému? Uveďte příklady.
   • **Komponenty** – části dekomponovaného systému s daným rozhraním.
   • **Konektory** – komunikační kanály pro propojení komponent s daným rozhraním.
   • **Konfigurace** – konkrétní způsob vzájemného propojení komponent pomocí konektorů.
8. Popište, co se rozumí pojmem komponenta informačního systému. Uveďte příklady.
   Softwarová **komponenta** je softwarový balík, služba nebo obecně modul zajišťující určitou funkčnost, a tedy zapouzdřující funkce a data.
9. Jaký je rozdíl mezi architekturou a návrhem informačního sytému?
   • **ARCHITEKTURA** se zabývá především technickými (jinými než funkčními) a částečně funkčními požadavky, zatímco **NÁVRH** vychází z čistě funkčních požadavků.
   • Proces definice **ARCHITEKTURY** využívá zkušenosti, heuristiky a postupná upřesnění a zlepšení. Vyžaduje vysokou míru abstrakce pro **NÁVRH** rozdělení logiky systému do samostatně fungujících částí (s přesně definovanými kompetencemi).
10. Jak se správně postupuje při stanovení architektury a návrhu informačního systému?
    • **Dekompozice**
        • Identifikace systémových požadavků
        • Dekompozice systému do komponent
        • Přidělení požadavků k jednotlivým komponentám
        • Ověření, že všechny požadavky byly přiděleny

11. Které kompetence obsahuje informační systém? Uveďte příklady těchto kompetencí.
    • Komunikace s uživatelem (prezentace informací, předání požadavků)
    • Zpracování informací a jejich (dočasné) uchování.
    • Trvalé uchování informací (dat).
    • => Třívrstvá architektura

Navíc

Životní cyklus • Vize • Analýza • Logický návrh • Technologický návrh • Vývoj • Nasazení a provoz

• **NÁVRH** (design) popisuje systém rozdělený do logických částí, tedy **JAK** funguje a s **ČÍM** pracuje (třídy, tabulky, komponenty, služby a vztahy mezi nimi).

• **NASAZENÍ** (deployment) popisuje **KDE** systém běží (na jakém HW, SW platformě…).

2. DOMÉNOVÁ LOGIKA

1. Co se rozumí pojmem návrhový vzor? Co každý vzor obsahuje? Uveďte příklady.

   •Vzory jako připravené návody •To co opakovaně funguje •Nikdy se nevyskytují o samotě •typicky ukazují vztahy a interakce mezi třídami a objekty

   • Výstižné jméno (název)
   • Problém (Co)
   • Souvislosti (Kdo, Kdy, Kde, Proč)
   • Řešení včetně UML diagramů (Jak)
   • Příklady včetně zdrojových kódů

2. Popište podstatu třívrstvé architektury. Jaký je rozdíl mezi fyzickou a logickou třívrstvou architekturou?

   •Třívrstvá architektura je nejznámějším případem vícevrstvé architektury. Odděluje vrstvy tak aby na sobě nebyly závislé. Hovoří o logické organizaci kódu. Lineární architektura

   •**Prezentační vrstva**

   Zobrazuje informace pro uživatele, většinou formou grafického uživatelského rozhraní, může kontrolovat zadávané vstupy, neobsahuje však zpracování dat.

   •**Doménová vrstva** (též Business Logic)

   Zde leží jádro aplikace, její logika a funkce, výpočty, zpracování dat a jejich dočasné uchování.

   •**Datová vrstva**

   Tuto vrstvu tvoří nejčastěji databáze, která data trvale uchovává, zpřístupňuje a zaručuje jejich konzistenci.

   Fyzická Hovoří o tom, kde ten kód běží. Fyzické vrstvy jsou místa, kde jsou rozmístěny a běží logické vrstvy. Např **Web server** je prezentační vrstva kde je html,css,javascript. **Application server** doménová vrstva vyvíjená např. v Ruby, Django, ASP.NET. **Database server** datová vrstva databáze jako je MySql,Oracle…

3. Co je podstatou vzoru MVC? V čem se liší od třívrstvé architektury?

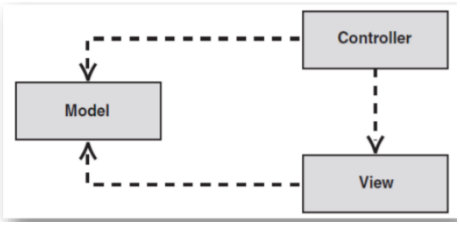
   • Oddělení vrstev na logické úrovni •Minimalizace závislosti-modifikace některé vrstvy má minimální vliv na ostatní •Obvykle neřeší přístup k datům •Je nutno chápat jako velmi obecný (a správný) návrhový/architektonický koncept. •Oproti třívrstvé architektury není lineární.

   •**Model** – Doménově specifická reprezentace informací, s kterými aplikace pracuje

   •**View** – Data reprezentované modelem převádí do zobrazovatelné podoby pro uživatele

   •**Controller** – Reaguje na události nejčastěji od uživatele a zajišťuje změny v Model nebo View

   Vzory blízké MVC: Observer a Data Binding (řešení založená na událostech), Model View ViewModel, Model View Presenter

4. Co řeší skupina návrhových vzorů pro doménovou logiku?

Vzory pro doménovou logiku (Domain Logic Patterns) popisují, jak rozdělit aplikaci do vrstev, dále popisují architekturu v rámci vrstvy aplikační logiky.

5. Popište podstatu jednotlivých vzorů pro doménovou logiku (Transaction script, Domain model, Table module, Service layer).
6. V čem se od sebe liší jednotlivé vzory pro doménovou logiku? Kdy, kde a proč je použít/nepoužít? Uveďte příklady.

Transaction script

•Organizuje doménovou logiku procedurami, kde každá procedura vyřizuje jeden požadavek z uživatelského rozhraní.

•Procedury si samé zajistí přístup k datům(databáze), vykonaní potřebné výpočty a vrátí výsledek do prezentační vrstvy.

•Tento vzor popisuje řešení pro malé problémy, ve kterých se požadavky uživatele promítají do relativně jednoduchého kódu.

Použití: Tam kde je málo logiky

Domain Model

•Objektový model domény, který zahrnuje chování i data.

Dva různé styly: • **Jednoduchý** – pracuje s jednoduchými logikami a třídy odpovídají tabulkám v databázi • **Úplný** – pracuje s komplexními logikami a model tříd se liší od databázového modelu

Použití: Vhodné pro už složitější systémy. Je dobré jej použít při rozsáhlé, měnící se logice.

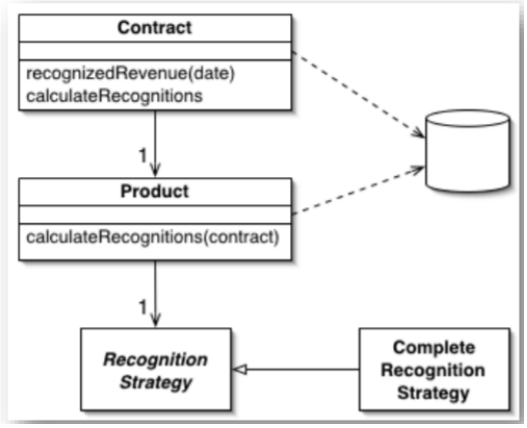


Table module

•Obstarává byznys logiku pro všechny řádky v databázové tabulce nebo pohledu.

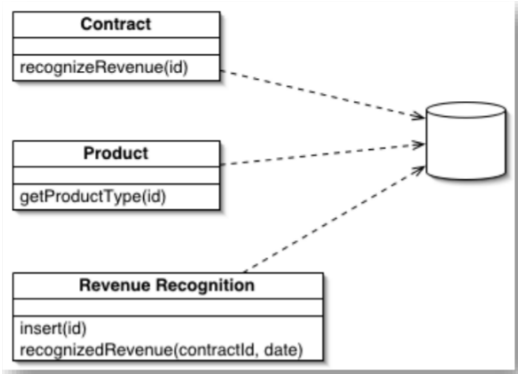
• Jedna instance třídy dle vzoru Table Module odpovídá jedné tabulce v DB.

Na rozdíl od domain model nemá ponětí o identitě objektu, se kterým pracuje. Proto pro získání objektu musíme použit např. takovou metodu:

anEmployeeModule.getAddress(long employeeID)

Použitelný s Table Data Gateway

Použití:U aplikaci často využívajících tabulkovou orientaci např DataGridView



Service Layer

•Definuje hranici aplikace, stanovuje množinu dostupných operací a koordinuje odpovědi aplikace pro každou operaci.

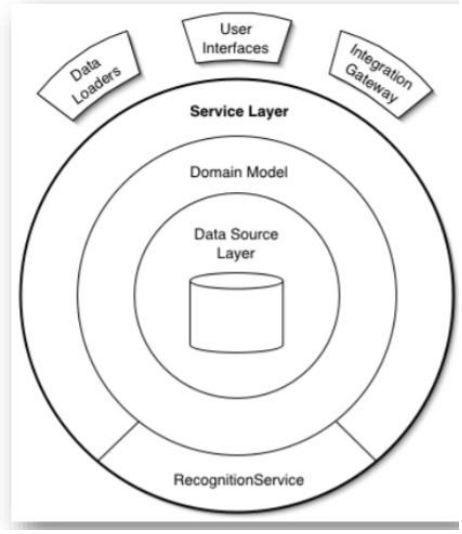
•Je to rozhraní, vůči kterému říkám, jaké mám požadavky a dostávám nějaké strukturované odpovědi, přičemž netušíme, jak je daný problém vyřešen.

Service Layer ukrývá byznys logiku

• **Doménová logika**, která se týká problémové oblasti (např. výpočetních strategií)

• **Aplikační logika**, která souvisí s odpovědnostmi informačního systému (zajištění tzv. workflow).

Je dobré ji **použít** pro jednoduché volání metod logiky. Pro přístup k mnoha klientům.

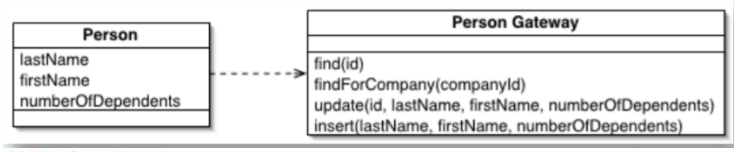


3. DATOVÉ ZDROJE

1. Co společně řeší skupina návrhových vzorů pro práci s datovými zdroji?
   •Základní požadavek je nezávislost doménové logiky na logice přístupu k datům
   •Vzory pro datové zdroje popisují, jakým způsobem komunikuje aplikační logika s databází.
   •Umístit do aplikační logiky kód pro komunikaci s databází má spoustu nevýhod, tyto vzory popisují, jak tento problém řešit.
2. Popište podstatu jednotlivých vzorů pro práci s datovými zdroji (Table data gateway, Row data gateway, Active record, Data mapper).
3. V čem se liší jednotlivé vzory pro práci s datovými zdroji? Kdy, kde a proč je použít/nepoužít?
4. Se kterými vzory pro doménovou logiku by se mohly použít některé ze vzorů pro práci s datovými zdroji a proč? Uveďte příklady

Table Data Gateway

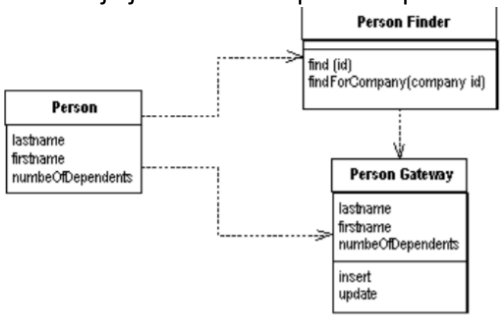
•Je objekt, který se chová jako brána do databázové tabulky a jedna instance této třídy se stará o všechny řádky této tabulky.
•Rozdíl oproti Table module:
Nemá na starosti doménovou logiku, pouze komunikuje s databází.
Použití: Je dobré jej použít při jednoduché doménové logice, s transaction script a table module.+Záměna SQL logiky Pro Domain model je lepší použít Data Mapper, protože poskytuje lepší izolaci.




```
public class Person{
    private int id;
    private String firstName;
    private String lastName;
    private int numberOfDependents;
}

public class PersonGateWay{
    public RecordSet find(int id){
    }

    public void update(int id, String firstName, String lastName, int numberOfDependents){
    }
    public void insert(String firstName, String lastName,int numberOfDependents){
    }

}
```

Row Data Gateway

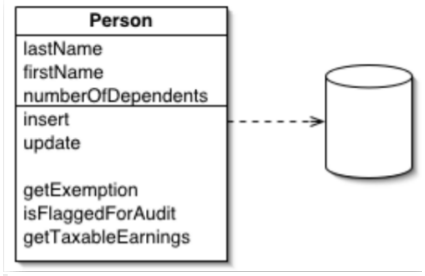
•Objekt, který slouží jako brána k jednomu záznamu ve zdroji dat. Na každý řádek připadá jedna instance.
•Gateway zde znamená objekt, který zapouzdřuje přístup k externímu systému nebo zdroji.
•Tento objekt neobsahuje metody doménové logiky. Pokud ano (zejména doménovou logiku), objekt se stane vzorem Active Record.
Použití: Široce se používá se vzorem Transaction Script. Kde je jednoduchá doménová logika. (Nepoužívat s Domain Model.) V tomto případě pěkně rozděluje přístupový kód k databázi a umožňuje jeho snadné opětovné použití různými transakčními skripty.




```
class PersonGateway{
   private String lastName;
   private String firstName;
   private int numberOfDependents;
   //gets sets

    public void update() {}
    public Long insert() {}
}

class PersonFinder{
     public PersonGateway find(Long id) {}

}
```

Active Record

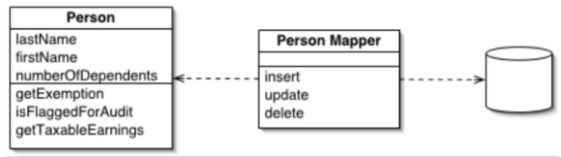
•Objekt, který obaluje řádek v databázové tabulce nebo zobrazení, zapouzdřuje přístup k databázi a přidává doménovou logiku k těmto datům.
•Podstatou aktivního záznamu je doménový model, v němž třídy velmi přesně odpovídají struktuře záznamu v databázi.
•Každý Active record je zodpovědný za ukládání a načítání do databáze a také za jakoukoli doménovou logiku, která pracuje s daty.
Použití: Složitější doména, ale s jednoduchými operacemi přímo mapovanými na tabulky. S table data gateway. Nevýhoda netriviální mapování do DB.




```
public class Person {
   private int id;
   private String lastName;
   private String firstName;
   private int numberOfDependents;
   // getter and setter methods

   public Long insert() {}
   public void update() {}

    public Money getExemption() {
      // Business logic
   }
   public boolean isFlaggedForAudit() {
      // Business logic
   return true;
   }

   public String getTaxableEarnings() {
      // Business logic
   }
}
```

Data Mapper

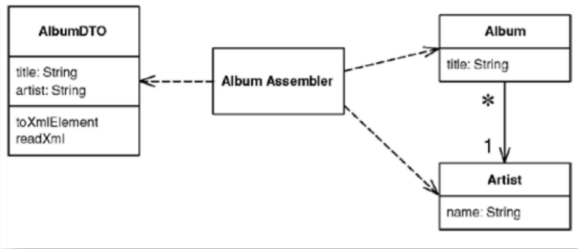
•Je to vrstva „mapperů“, která přesunuje data mezi objekty a databází, zatímco je udržuje nezávislé na ostatních a „mapperu“ samotném.
Použití: Je dobré jej použít pro nezávislou podobu domény a databáze. S doménovým modelem, složitou doménovou logikou. Použití s Domain Model, protože poskytuje lepší izolaci. -je komplikovanější
•Jednoduchým případem by byla třída Person a Person Mapper. Pro načtení osoby z databáze by klient zavolal metodu find na mapperu. Mapovač použije vzor Identity Map, aby zjistil, zda je osoba již načtena; pokud ne, načte ji.



```
public class Person {
   private int id;
   private String lastName;
   private String firstName;
   private int numberOfDependents;
}
public class PersonMapper{

    public bool update(Person person){
    }
    public bool insert(Person person){
    }

}
```

Data Transfer Object (DTO)

Objekt, který přenáší data mezi procesy, aby se snížil počet metod.



4. OBJEKTOVĚ-RELAČNÍ CHOVÁNÍ

1. Pro každý ze čtyř návrhových vzorů pro práci s datovými zdroji si promyslete a napište kousíček zdrojového kódu, ze kterého bude poznat, o který vzor jde.
2. Co společně řeší návrhové vzory pro objektově-relační chování?

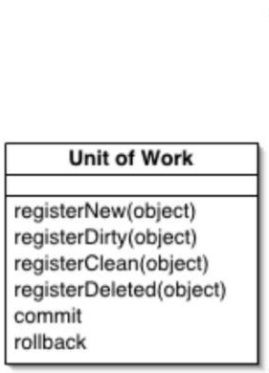
   Řeší způsob ukládání změněných dat do databáze. Při složitějším výpočtu muže být obtížné uhlídat, která data byla změněná a musí být uložena do databáze. Tyto vzory přicházejí v úvahu, pokud máme větší množství záznamů, nad kterými se pracuje.
3. Kdy bychom měli zvážit použití vzoru Unit of Work a proč? Na příkladu vysvětlete, jak jej použít.

   •Spravuje nebo udržuje seznam objektů dotčených nějakými bussiness transakcemi a koordinuje zápis změn a řešení problémů.

   •Jestliže mám v paměti objekt, který se za dobu své životnosti měnil (voláním funkcí typu vytvoř se, změn položku, přičti hodnotu, vytvoř položku… a nakonec smaž se), tak není potřeba promítat každou změnu objektu do DB, když se stejně nakonec smazal.

   •Místo hloupého vykonávání každého z těchto úkonů a postupného promítání změn do DB se změna uloží na vyžádání.

   •**Kdy použít**? Když můžeme odložit ukládání objektů do DB. Když se objekty mění často. Nepoužívat v kritických systémech



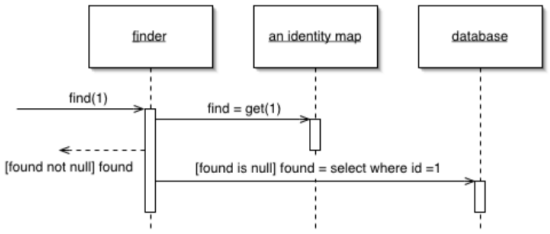

```
class UnitOfWork...

   public void registerNew(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
      Assert.isTrue("object not removed", !removedObjects.contains(obj));
      Assert.isTrue("object not already registered new", !newObjects.contains(obj));
      newObjects.add(obj);
   }
   public void registerDirty(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not removed", !removedObjects.contains(obj));
      if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
         dirtyObjects.add(obj);
      }
   }
   public void registerRemoved(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      if (newObjects.remove(obj)) return;
      dirtyObjects.remove(obj);
      if (!removedObjects.contains(obj)) {
         removedObjects.add(obj);
      }
```

4. Kdy bychom měli zvážit použití vzoru Identity Map a proč? Na příkladu vysvětlete, jak jej použít.

•Zajišťuje, že každý objekt bude načtený v paměti pouze jednou tak, že každý načtený objekt drží načtený v mapě. Pokud chceme na objekt odkázat, podívá se napřed do mapy a pokud tam není, díváme se do DB. Snižuje počet volání databáze.

•**Kdy použít**? • + Když nechceme, aby dva objekty nereprezentovaly stejný záznam v DB
Nepoužívat, když objekty nemění svůj stav

```
private Map people = new HashMap();
public static void addPerson(Person arg) {
    soleInstance.people.put(arg.getID(), arg);
}
public static Person getPerson(Long key) {
    return (Person) soleInstance.people.get(key);
}
public static Person getPerson(long key) {
    return getPerson(new Long(key));
}
```



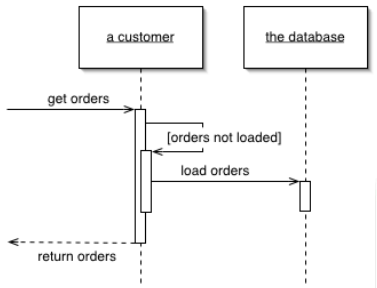
1. Požadavek nejprve přijde do metody **finder** s primárním klíčem.
2. Vyhledávač zkontroluje, zda **PersonMap** (IdentityMap) obsahuje objekt osoby pro daný primární klíč.
3. Pokud je objekt osoby v **PersonMap** (IdentityMap) nalezen, pak vrátí objekt osoby z **PersonMap** a nezasahuje do databáze.
4. Pokud objekt osoby není nalezen v **PersonMap** (IdentityMap), pak vyvolá SQL dotaz na databázi a získá záznam z databáze.

5. Kdy bychom měli zvážit použití vzoru Lazy Load a proč? Na příkladu vysvětlete, jak jej použít.

•Je to objekt, který neobsahuje všechna data, ale ví, jak zbylá data získat

•Umožňuje nenačítat zbytečná data, která by jen zatěžovala linku k DB. Pokud později některá z nenačtených dat k objektu budeme potřebovat, dodatečně se načtou.

•**Kdy použít**? + Když pro získání objektu z DB potřebujeme extra volání
Nepoužívat - Když nejsou objekty ve složitých kompozicích




```
class Supplier...

   public List getProducts() {
      if (products == null) products = Product.findForSupplier(getID());
      return products;
   }
```

6. Popřemýšlejte, se kterými vzory pro práci s datovými zdroji byste mohli společně použít některé ze vzorů pro objektově-relační chování a proč? Zkuste najít příklady.

5. OBJEKTOVĚ-RELAČNÍ STRUKTURY

1. V jakých situacích byste použili vzor Identity field a proč? A kdy naopak ne? Napište fragment kódu, ze kterého bude patrné, proč jste tento vzor využili.

•Udržuje ID z DB v objektu, který tím zajišťuje komunikaci mezi objektem v paměti a řádkem tabulky DB. Id v objektu je shodné s id v db.

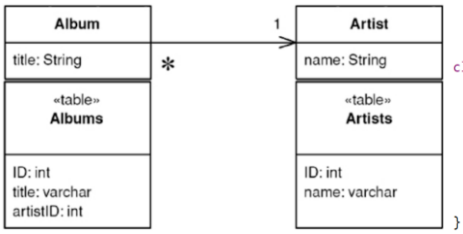
•Používat, pokud existuje mapování mezi objekty v paměti a řádky v databázi.

•Naopak nepoužívat, pokud takové mapování neexistuje.

```
class Person{
    private long id;
    private String name;
    private int age;
    private String gender;
    // standard getter and setter methods.
}
```

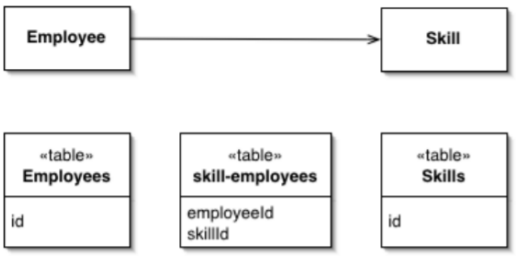
2. Jaký je rozdíl mezi vzory Foreign key mapping a Associate table mapping? Napište dva fragmenty kódu, ze kterých bude patrné, že jste použili tyto vzory.

•**FKM** Mapuje vazby mezi objekty pomocí cizího klíče odkazujícím na vazby mezi tabulkami.
•Používá se hlavně v asociacích one-one, one-many a many-one.
•Čili v tabulce kvůli vazbě mam v Albu ID Artist. Ale v paměti vazbu tvořím referencí na objekt, ne udržováním jeho ID.



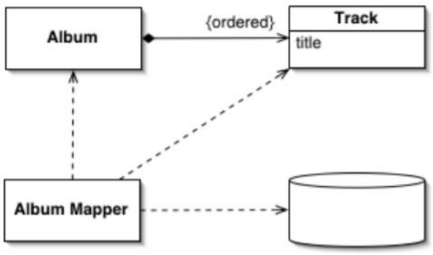

```
class Artist {
    private long id;
    private String name;
    public Artist(long ID, String name) {
        this.id = ID;
        this.name = name;
    }
class Album {
    private long id;
    private String title;
    private Artist artist;
    public Album(long ID, String title, Artist artist) {
      this.id = ID;
      this.title = title;
      this.artist = artist;
   }
}
```

•**ATM** Udržuje vazby jako tabulku s cizími klíči tabulek, které jsou spojeny vazbou. V kódu by se to řešilo seznamem objektů na obou stranách. V DB by to byla m:n vazební tabulka. Cílem je řešení m:n vazeb.
•Například mezi **employees** a **skills** existuje vztah m:n employee může mít více skills a skill může být spojena s více employees.




```
public class Student {

 private long studentId;
 private String studentName;
 private Set<Course> courses = new HashSet<Course>(0);

public Student(String studentName, Set<Course> courses) {
  this.studentName = studentName;
  this.courses = courses;
 }
}
```

3. V čem se liší vzor Dependent mapping od vzoru Data mapper? Kdy je vhodné jej použít?
•Vzor poskytující speciální třídu, která mapuje do databáze i objekty, které jsou asociované s daným objektom. U Data mapperu se každá třída mapuje zvlášť.
•Album Mapper mapuje objekty třídy Album. Ale Album je jediným vlastníkem objektů třídy Track. Album Mapper v tomto případě mapuje i objekty Track patřící tomuto albu.
•**Použití**: Používá se v případě, když máte objekt, na který odkazuje pouze jeden jiný objekt, k čemuž obvykle dochází, když má jeden objekt kolekci závislých objektů. Nesmí existovat reference na dependenci z žádného objektu kromě vlastníka.

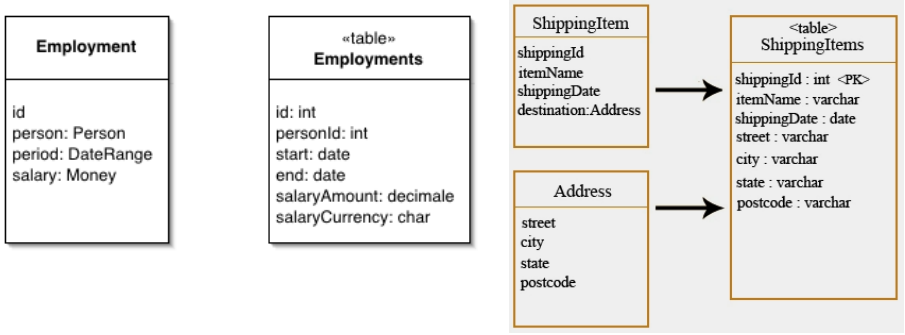


4. Napište fragment kódu, ze kterého bude patrné, že jste použili vzor Dependent mapping.

```
class Track {
   private String title;
   public Track(String title) {
      this.title = title;
   }
}
class Album {
   private List tracks = new ArrayList();
   public void addTrack(Track arg) {
      tracks.add(arg);
   }

   public void removeTrack(int i) {
      tracks.remove(i);
   }
}

class AlbumMapper {
     public void insertTrack(Track track, int seq, Album album){
      //insert
   }
}
```

5. Vymyslete alespoň dva příklady vhodné pro použití vzoru Embedded value. Co je podstatou tohoto vzoru? Proč a kdy je vhodné jej použít?

•Mapuje více tříd do jedné databázové tabulky. Zkrátka jde o to, že nemusí platit, že jedna třída = jedna tabulka. V tomto případě viz obrázek se tabulka Employments rozdělí v paměti na třídy



Person, DateRange a Money. Je to použitelné pouze tehdy, když je mezi objekty v paměti vazba 1:1.

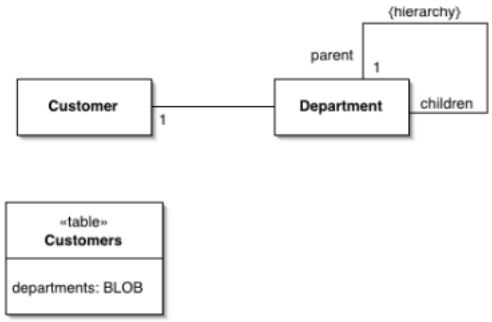
6. Vymyslete alespoň dva příklady vhodné pro použití vzoru Serialized LOB. Co je podstatou tohoto vzoru? Proč a kdy je vhodné jej použít?

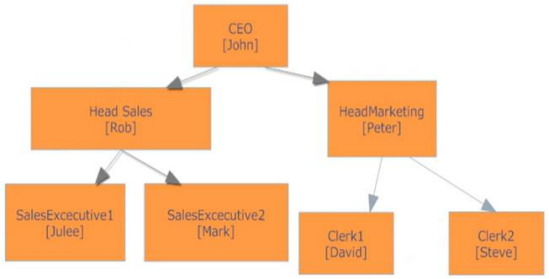
•Říká, že si složitou strukturu serializuje do jednoho LOBu (třeba pomocí XML) a ten uložím jakožto jeden objekt do jedné buňky tabulky.

•Department má v sobě složitou hierarchii oddělení, kterou serializuju do jednoho velkého objektu (LOB) a ten uložím do buňky tabulky Customers.

Praktické využití například při zálohování verzí. Aktuální data budou v jednotlivých objektech, kdežto zálohy se serializují do XML a uloží jako jeden objekt

•**Používá** se, když objekt nemá moc dat, ale má složitou strukturu (velmi mnoho vztahů mezi objekty), která by se jinak složitě mapovala do DB





6. MAPOVÁNÍ DĚDIČNOSTI A OBECNÉ VZORY

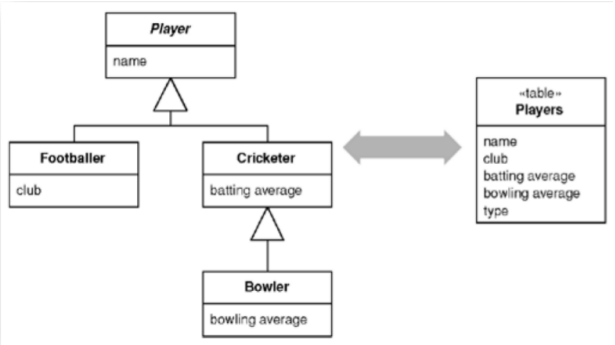
1. Popište podstatu vzorů Single / Class / Concrete Table Inheritance a v jakých situacích je vhodné je použít.

Single Table Inheritance

•Reprezentuje hierarchii dědičnosti tříd jako jedinou tabulku, která obsahuje sloupce pro všechna data v těchto třídách

•**Výhody**: •V databázi je pouze jedna tabulka, o kterou se musíme starat. •Při získávání dat se nepoužívají žádná spojení. •Změna hierarchie tříd nevyžaduje změnu v databázi.

•**Nevýhody**: •Pole jsou někdy potřeba a někdy ne, což může být matoucí pro lidi, kteří pracují přímo s tabulkami. •Sloupce používané pouze některými podtřídami zabírají zbytečně místo v databázi. •Jedna tabulka může být příliš velká, mít mnoho indexů a zamykání, vliv na výkon.

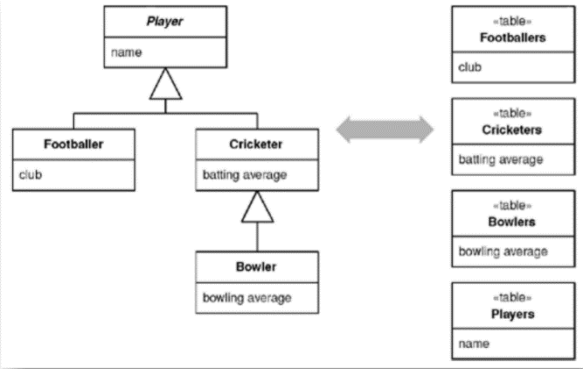


Class Table Inheritance

•Reprezentuje hierarchii dědičnosti tříd tak, že každá třída má vlastní tabulku

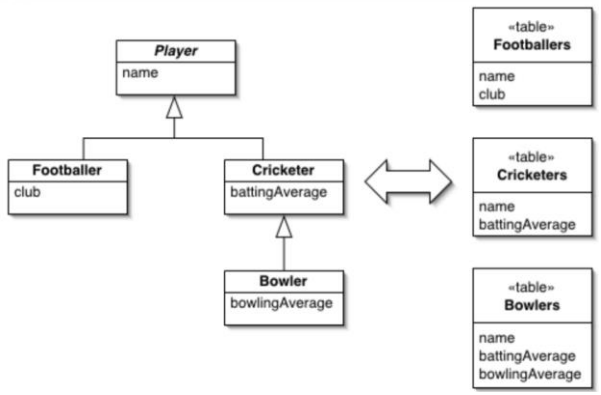
•**Výhody**: • Pro každý řádek jsou použité všechny sloupce, takže tabulky jsou přehlednější a neplýtváme místem. • Vztah mezi doménovým modelem dědičnosti a databází je přímočarý.

•**Nevýhody**: • Pro načtení objektu je třeba se pracovat s více tabulkami, což znamená použití spojení více dotazů v paměti. • Jakýkoli přesun dat v dědičné hierarchii nahoru nebo dolů způsobuje změny v databázi. • Tabulky nadtypů se mohou stát úzkým hrdlem, protože se k nim musí často přistupovat. • Vysoký stupeň normalizace databáze může ztížit pochopení pro ad-hoc dotazy



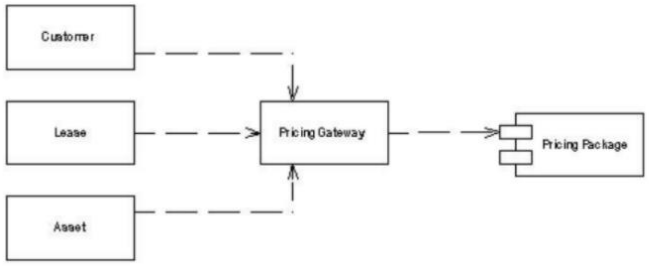
Concrete Table Inheritance

•Reprezentuje hierarchii dědičnosti tříd takovým způsobem, že co potomek (třída), to jedna tabulka v DB, jenž obsahuje atributy dané třídy + navíc atributy předků této třídy.
•**Výhody**: •Každá tabulka je samostatná a neobsahuje žádná prázdná pole. • Při čtení dat z konkrétních mapperů není třeba provádět žádná spojení. • Ke každé tabulce se přistupuje jen tehdy, když se přistupuje k dané třídě, což snižuje náklady na dotazování.
•**Nevýhody:** •Horší práce s abstraktními třídami. Pokud proměnné třídy přesuneme v jejich hierarchii nahoru nebo dolů, musíme změnit tabulky. •Pokud se změní proměnná nadtřídy, je nutné změnit každou tabulku, která ji má. •Polymorfismus (více tříd se stejným názvem, ale každá patří jinam) vyžaduje kontrolu všech tabulek, což vede k vícenásobným přístupům do databáze.

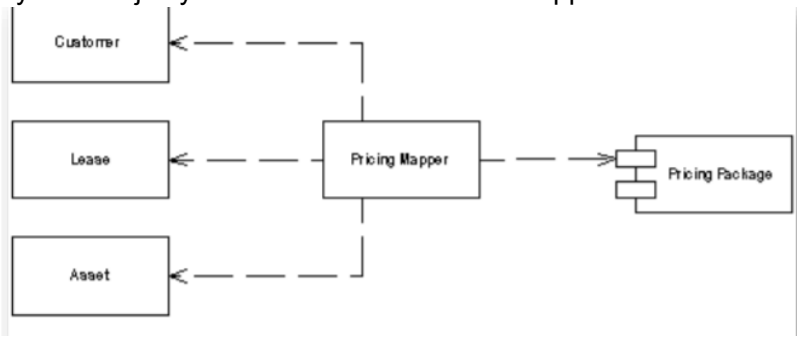


2. Napište fragment kódu, ze kterého bude patrné, že jste použili vzor Single Table Inheritance.
3. Jaký je rozdíl mezi vzory Class a Concrete Table Inheritance? Napište dva fragmenty kódu, ze kterých bude patrné, že jste použili tyto vzory.
   Class table inheritance má pro úplně každou třídu svou tabulku, zatímco concrete table inheritance pouze pro konkrétní třídy, tedy ne abstraktní
4. Vymyslete a popište příklad na využití vzoru Gateway.

- Objekt, ktorý predstavuje prístup do externého systému alebo zdroja
- Objekty vo "vnútri" systému využívajú gateway na prístup do externého zdroja
- *Table Data Gateway, Row Data Gateway*



5. Vymyslete a popište příklad na využití vzoru Mapper.
   Objekt zaisťujúci komunikáciu medzi dvoma nezávislými objektami. Mapper pre svoju činnosť využíva objekty na oboch stranách. Data Mapper.



6. Vymyslete a popište příklad na využití vzoru Layer Supertype. Napište fragment kódu využívající dědičnost, ze kterého bude patrné, že jste použili tento vzor.
   Máme mnoho tříd vzoru Data Mapper a všechny tyto třídy obsahují společné základní chování. V takovém případě můžeme vytvořit nadřazenou třídu, ze které budou naše mappery toto chování dědit.

```
public abstract class MapperSupertype<T> {
    public T FindById(int id) {}

    public bool Insert(T object) {}
}

public class PersonMapper<T> : MapperSupertype<T> {
    public T FindById(int id) {
        //...
    }

    public bool Insert(T person) {
        //...
    }
}

public class CarMapper<T> : MapperSupertype<T> {
    public T FindById(int id) {
        //...
    }

    public bool Insert(T car) {
        //...
    }
}
```

7. Vymyslete a popište příklad na využití vzoru Service Stub (Mock Object). Napište fragment kódu, ze kterého bude patrné, že jste použili tento vzor.
   •Dočasně nahrazuje závislost nad problémovými službami v průběhu vývoje a testování
   •Dočasná služba by měla implementovat stejné rozhraní jako skutečná, abychom mohli napodobit její reálné chování.

•Informační systém provádí operace v závislosti na čase dne. Abychom nemuseli čekat několik hodit na otestování těchto operací, implementujeme si dočasný objekt, který bude čas pouze simulovat, ale se kterým bude lépe manipulovatelné

7. JAK VYVÍJET V TÝMU

1. Popište, co se rozumí vodopádovým modelem a jaké jsou jeho výhody a nevýhody.
   • Původně sedm navazujících (nepřekrývajících se) fází:
   Specifikace požadavků, Návrh, Implementace, Integrace, Testování, Ladění, Instalace, Údržba.
   • Vodopádový model vyžaduje, aby se k následující fázi přikročilo pouze tehdy, pokud je ta předcházející kompletní a perfektně připravená.
   • **Výhody** (dvakrát měř a jednou řež)
      • Včasné odhalení chyb (vede k úsporám).
      • Vysoký důraz na dokumentování (bezproblémová záměna lidí).
      • Jednoduchost pro řízení (stabilita projektu).
   • **Nevýhody**
      • Je nemožné dovést jednu fázi životního cyklu softwarového produktu k dokonalosti předtím, než se přejde k další fázi.
      • Trvá se na rozhodnutích, která se mohou později ukázat jako nesprávná.
2. Popište, co se rozumí iterativním a inkrementálním vývojem. Uveďte příklad.
   • **Iterativní** návrh je metodologie založená na opakovaném procesu analýzy, návrhu, implementace (prototypování), testování a redefinici produktu.
   • **Přírůstkový** (inkrementální) model je založen na principu postupně budovaného produktu po přírůstcích založených na iterativním návrhu.
   • Vývoj kombinuje vlastnosti vodopádového modelu s vlastnostmi iterativního prototypování.
3. Co se rozumí UP (unified process), na jakých principech, charakteristikách a fázích je postaven? Uveďte příklady.
   • **Dva klíčové principy:** Iterovaný a inkrementální vývoj
   • **Tři klíčové charakteristiky**: Use-case-driven - projektový tým používá use cases (případy užití) ve všech fázích vývojových prací, od počátečního sběru požadavků až po tvorbu zdrojového kódu systému., Architecture-centric - Architektura systému stojí v centru celého vývoje, Risk-focused - Vývojářský tým by se měl zaměřit na řešení nejkritičtějších rizik (nejistot) v rané fázi životního cyklu projektu.
   • **Čtyři fáze**: zahájení, rozpracování, konstrukce, – nasazení do provozu.
   • UML, dokumenty
   • Rational Unified Process (RUP) • Microsoft Solutions Framework (MSF) • Oracle Unified Method (OUM) • Open Unified Process (OpenUP)
4. Jaké jsou čtyři základní charakteristiky agilního softwarového vývoje, které ho odlišují od vodopádového a dalších robustních přístupů?
   1. **Lidi a komunikace** jsou víc než jen procesy a nástroje
   2. **Funkční software** je víc než zdlouhavá a vyčerpávající dokumentace
   3. **Spolupráce se zákazníkem** je víc než vyjednávaní smlouvy
   4. **Radši reagovat na změny** než se držet plánu
5. Na jakých principech a praktikách je založeno tzv. extrémní programování?
   • Všechno, co je správné, se dělá naplno. Nelze z ničeho ustoupit
   Principy: • Komunikace • Jednoduchost • Zpětná vazba • Odvaha
   Praktiky: • **Business** praktiky (Plánování hry, Zákazník na pracovišti, Vydávání malých verzí, Metafora)

• **Týmové** praktiky (Párové programování, Společné vlastnictví kódu, Standardy kódu, Udržitelné tempo)
• **Programovací** praktiky (Neustálá integrace, Jednoduchý návrh, Refaktorování kódu, Testování)

6. Které nejdůležitější charakteristiky má SCRUM?

SCRUM patří mezi nejpoužívanější metodiku agilního řízení projektu. Vývoj prostřednictvím této metodiky probíhá v tzv. iteracích (sprint), jež obvykle trvají okolo dvou týdnů až jednoho měsíce. Tyto iterace jsou zakončeny setkáním (Sprint Review Meeting), kterého se účastní všechny zainteresované osoby včetně zákazníků, přičemž účast zákazníků není nutně vyžadována.

- vývoj v krátkých iteracích (2 až 4 týdny);
- fixní náklady a termíny;
- flexibilní určení výstupů;
- malé týmy;
- časté analýzy rizik a revize;
- Sprinty • Schůzky (denně)
- velký důraz na komunikaci a spolupráci mezi všemi zúčastněnými (včetně zákazníka)

Scrum Master: pracuje jako spojení mezi týmem a jakýmkoliv rušivým elementem zvenku. Role Scrum mastera je chránit tým a vytvářet prostředí, kde by členy týmu nikdo neměl rušit, podporovat tým.

Product Owner je vlastník produktu, který má určovat priority a rozhoduje o tom, na čem se bude dále pracovat. Má na starosti vizi projektu.

Scrum tým Týmová spolupráce je velice důležitá pro agilní vývoj. Ve Scrumu se tým bere jako nedělitelný celek, proto nezáleží na selhání jednoho člena, vždy za vše odpovídají všichni.

7. Popište proces typický pro tzv. testy řízený softwarový vývoj. Uveďte příklad.

• Napsat test • Spustit testy a ujistit se, že všechny neprojdou • Napsat vlastní kód • Kód automatickými testy prochází • Refaktoring • Opakování

8. DOMÉNOVĚ SPECIFICKÉ JAZYKY

1. Co je doménově specifický jazyk? Proč a kdy je dobré ho využít? Uveďte příklady.

• Programovací jazyk s omezenou expresivitou, zaměřený na konkrétní doménu
Usnadnění pochopení kódu, a proto je lze rychleji napsat, rychleji upravit a je méně pravděpodobné, že se v nich budou chyby.
Protože jsou DSL menší a srozumitelnější, umožňují neprogramátorům vidět kód, který řídí důležité části jejich aplikace.
Usnadňují komunikaci mezi programátory a zákazníky, což je často největší problém
• **SQL** jako vyjadřovací prostředek pro komunikaci s databází.
• **HTML** jako deklarativní jazyk použitelný pro popis toho, jak má vypadat webová stránka.
• Jazyk „knihovny“ programovacího jazyka určený pro práci se specifickými úlohami patřícími do jedné domény. Např. **DateTime** a podpora úloh s datem a časem.

2. Co je cílem využití doménově specifického jazyka a jaké vlastnosti by měl mít?
3. Jaký je rozdíl mezi externím a interním doménově specifickým jazykem? Uveďte příklady.

• External - Jazyk oddělený od hlavního jazyka aplikace, se kterou pracuje.
(např. XML, SQL, regulární výrazy)

• Internal - je zvláštní použití obecného programovacího jazyka. Při formulaci výrazů interního DSL se omezujeme na malou podmnožinu vlastnosti obecného programovacího jazyka.

4. S jakými problémy se můžeme setkat při návrhu doménově specifického jazyka? Uveďte příklady.

Jazyková různorodost

• Použití více jazyků při vývoji stěžuje práci na vývoji

• Je nutné znát více jazyků, ne jen jeden

Náklady na vytvoření

DSL může mít malé náklady. Ale pořád to jsou náklady na sestavení kódu a udržování kódu.

Uzavřenost jazyka

• Společnost, která stále vylepšuje vlastní DSL ho může tak vylepšit, až se z něho stane všeobecný PJ !

•Také to stěžuje novým zaměstnancům naučit se používat tento jazyk

Snížení abstrakce

• DSL umožňuje vyjádřit chováni domény jednodušeji, než by se to vyjádřilo všeobecným PJ . Tím odbouráváme nutnost přemýšlet na nizko úrovňových konstrukcích.