VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

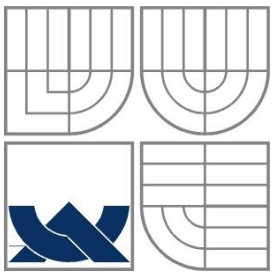
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OBJEKTOVĚ-RELAČNÍ MAPOVÁNÍ NA PLATFORMĚ PHP

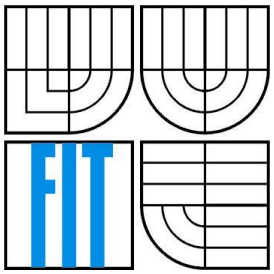
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Petr Mokruša
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OBJEKTOVĚ-RELAČNÍ MAPOVÁNÍ NA PLATFORMĚ PHP

OBJECT-RELATIONAL MAPPING ON PHP PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Petr Mokruša
AUTHOR

VEDOUCÍ PRÁCE — Ing. Radek Burget, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Práce se zabývá problematikou objektově-relačního mapovaní na platformě PHP. První část obsahuje obecný popis technologie ORM včetně návrhových vzorů pro ni určených. V další části jsou popsány dva nejvýznamnější ORM frameworky, Doctrine 2 a Propel. Doctrine 2 se práce věnuje více podrobněji, protože právě tento framework je použit v ukázkové aplikaci, která znázorňuje, jak technologii ORM integrovat do svého projektu a jak s ní efektivně pracovat.

Cílem této práce je seznámit čtenáře s technologii ORM a motivovat ho k použití ORM ve svých projektech.

Abstract

The thesis deals with an issue of object-relational mapping on the PHP platform. The first part contains general description of the ORM technology including design patterns meant for the technology. In the next part are described the two most important ORM frameworks - Doctrine 2 and Propel. The thesis is focused on Doctrine 2 as this framework is used in an exemplary application which represents how can be the ORM technology integrated to a project and how it can be effectively used.

The aim of this thesis is to introduce the ORM technology to readers and to motivate them to use the ORM technology in their projects.

Klíčová slova

Objektově-relační mapování, ORM, PHP, Doctrine 2, Propel, Kdyby\Doctrine.

Keywords

Object-Relational Mapping, ORM, PHP, Doctrine 2, Propel, Kdyby\Doctrine.

Citace

Petr Mokruša: Objektově-relační mapování na platformě PHP, bakalářská práce, Brno, FIT VUT v Brně, 2015

Objektově-relační mapovaní na platformě PHP

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Petr Mokruša

19.5.2015

Poděkování

Rád bych poděkoval svému vedoucímu za užitečné rady a vstřícnost při spolupráci.

Obsah

Obsah 1
1 Úvod 3
2 Objektově-relační mapování 4
2.1 Výhody použití ORM frameworku 5
2.2 Nevýhody použití ORM frameworku 6
2.3 Návrhové vzory 7
2.3.1 Návrhové vzory architektury 7
2.3.2 Návrhové vzory chování 10
3 Doctrine 2 14
3.1 Tři vrstvy Doctrine 2 14
3.2 Definice entit 15
3.3 Základní operace s entitami 18
3.4 Stavy entit a UnitOfWork 20
3.5 Implicitní a explicitní transakce 22
3.5.1 Implicitní transakce 22
3.5.2 Explicitní transakce 22
3.6 Základní asociace mezi entitami 22
3.7 dotazovací jazyk DQL 24
3.8 Query Builder 25
3.9 Kdyby/Doctrine 25
3.10 Integrace Doctrine 2 do Nette 26
4 Propel 27
4.1 Schéma databáze 27
4.2 Základní operace 28
4.2.1 Vytvoření a uložení objektu 28
4.2.2 Získávání dat z databáze 28
4.2.3 Aktualizace a mazáni dat 29
4.3 Základní vztahy 30
4.4 Instalace frameworku Propel 31
5 Vlastní aplikace 33
5.1 Vlastní aplikace a Doctrine 2 33
5.2 Class diagram datové vrstvy (entity) 35
5.3 Popis nejdůležitějších funkcí 36
5.3.1 Zpracování objednávky (dobropis, výdejka) 36

5.3.2 Seznamy 39
5.3.3 Přehledy a sestavy 40
5.4 Instalace aplikace 42
5.5 Zhodnocení použití ORM 42
6 Závěr 44
Literatura 45

1 Úvod

První část práce se věnuje obecně teorii, kterou je potřeba zvládnout pro správné pochopení fungování jednotlivých oblastí objektově-relačního mapování. Jsou zde shrnuty základní výhody a nevýhody ORM. Kdy je ORM vhodné používat a kdy naopak není. Dále jsou popsány významné návrhové vzory používané pro frameworky ORM.

Většina moderních informačních systémů je vyvíjena za pomoci objektově orientovaného programovacího jazyka. Příchod objektově orientovaného přístupu v programování s sebou přinesl myšlenku, kdy libovolný objekt v reálném světě může být reprezentován modelovým objektem. S příchodem PHP 5 a ustálení koncepce objektově orientovaného programování, začíná také vývoj frameworků pro objektově-relační mapování. Pro vývojáře je mnohem přirozenější reprezentovat v aplikaci objekt reálného světa a pracovat s ním jako s objektem, než jako s řádkem respektive množinou řádků v relační databázi. Do jisté míry se jedná o problém, který lze automatizovat, a tak vznikla programovací technika ORM.

Tato práce popisuje dva z nich, Doctrine 2 a Propel. Doctrine 2 jsem použil i pro svůj informační systém (ukázkovou aplikaci), a proto je zde popsán a rozebrán daleko podrobněji než Propel. Ukázková aplikace demonstruje jak jednoduše je možné integrovat ORM do svého projektu (v tomto případě do Nette) a jak jednoduše s ním pracovat.

2 Objektově-relační mapování

Objektově relační mapování (ORM) jde ortodoxně za myšlenkou OOP. Z databáze tedy místo pole řádků, jak je tomu u relačních databází, dostáváme rovnou pole objektů, se kterými můžeme pracovat běžnými prostředky jazyka.

V relační databázi je entita reprezentována jako řádek nebo množina řádků v databázových tabulkách. V objektově orientovaných jazycích je entita zpravidla reprezentována jako instance třídy. Tato rozdílná reprezentace dat vedla ke vzniku technologie ORM, která se stará o konverzi mezi relačními databázemi a objekty, které se používají v objektově orientovaných jazycích.

Díky tomu vývojář vidí tabulky v databázi jako kolekci objektů a v jazyce SQL nemusí s databází vůbec komunikovat. Do jisté míry je naprosto odstíněn od toho, že pracuje s relační databází. ORM přístup usnadňuje hlavně provádění častých databázových operací. Mezi tyto operace patří zejména operace čtení, zápis, mazaní a úprava dat. Dohromady jsou označovány jako CRUD operace (create, retrieve, delete, update).

ORM dále zajišťuje persistentní uchování dat a stará se o automatickou konverzi rozdílných datových typů mezi databázovým systémem a programovacím jazykem.

Metoda mapování relační databáze na objekty, má podporu téměř ve všech jazycích. S verzí PHP 5 a se stabilizací objektového programování začal také vývoj ORM frameworků pro PHP. Lze je dělit podle počtu operací, které nabízejí pro vývojáře. Některé frameworky nabízejí komplexnější přístup a přidávají různé nadstavby pro usnadnění vývoje rozsáhlejších systémů. Jedna z těchto komplexnějších knihoven je ORM knihovna Doctrine 2 [4]. Pro svůj projekt jsem zvolil právě knihovnu Doctrine 2 a proto bude později popsána podrobněji.

ORM frameworky jsou založeny na návrhových vzorech. Významným autorem v oblasti návrhových vzorů pro ORM je softwarový architekt Martin Fowler [2] [3], který vydal několik knih popisujících nejvýznamnější návrhové vzory pro ORM. Rozdělil je na návrhové vzory architektury a návrhové vzory chování.

Komplexnější ORM frameworky jsou vhodné pro aplikace, od kterých se očekává dlouhodobý vývoj a nasazení vícečlenného týmu nebo modulární aplikace jakou je například internetový obchod nebo redakční systém, u kterých je potřeba mít možnost připojovat nezávislé moduly.

Přínos ORM frameworků klesá se snižující se velikostí aplikace a pro jednoduchou stránku bez redakčního systému a s několika málo tabulkami ztrácí jejich nasazení smysl. ORM framework vhodně doplňuje návrhový vzor MVC (Model View Controler) a unit testy, což je základ pro vývoj robustní webové aplikace.

2.1 Výhody použití ORM frameworku

Abstraktní databázová vrstva:

PHP standardně nabízí PDO pro univerzální přístup k databázovým systémům. Nejednotný zápis SQL příkazů však nadále přetrvává. Například zápis omezení pro výběr prvních deseti záznamů z dotazu se pro MySQL a MSSQL liší. ORM frameworky např. Doctrine 2 nebo Propl nabízí takový zápis databázových dotazů, který je přenositelný mezi různými databázemi a to i včetně dotazů pro tvorbu nebo aktualizaci struktury tabulek.

Zapouzdření do objektů:

Možnost zapouzdřit související funkce přímo do objektu reprezentujícího řádek v tabulce velice usnadňuje čitelnost a testování kódu. Většina ORM frameworků nabízí předgenerované třídy pro každou tabulku databáze se základními metodami CRUD, gettery a settery. Tyto objekty je možné dál jednoduše rozšiřovat o vlastní kód, aniž by došlo při dalším generování základních tříd k jejich nežádoucímu přepsání.

Unit testy:

ORM frameworky mají v mnoha případech zajištěnou kvalitu kódu pomocí unit testů. Tyto testy jsou napsány takovým způsobem, aby bylo možné testovat i metody v rozšířených třídách. Pokud ORM framework podporuje databáze s paměťovým úložištěm, jsou unit testy velice rychlé a je zajištěno izolované prostředí pro každý běh testu.

Validátory hodnot objektu:

Hodnoty objektu (záznamu v dané tabulce) jsou nastavovány přes vygenerované gettery a settery nebo přes magické metody představené v PHP5. Proto je možné hodnoty před nastavením validovat případně vyčistit od nebezpečných znaků. ORM frameworky, které jsou generované z databázové struktury nebo popisných souborů (XML, YML nebo anotace), samy vygenerují základní validátory na povinné hodnoty a datové typy proměnných tříd.

CRUD generátor:

Základní operace se záznamy jsou hlavním účelem administračního systému. ORM framework je schopen na základě definice databázových objektů vytvořit uživatelské rozhraní pro tyto operace včetně validátorů a výpisu s filtrací. Vygenerované rozhraní je možné dále rozšiřovat a CRUD generátor může úspěšně fungovat jako základ rozsáhlých administračních systémů.

2.2 Nevýhody použití ORM frameworku

Učící křivka:

Největší nevýhodou ORM frameworku je nutnost naučit se ho používat. Čím více je framework komplexní, tím více dokumentace je potřeba zpracovat. Při výběru vhodného frameworku je nutné si předem ověřit dostupnost kvalitní dokumentace a zda existuje funkční komunita, která bude schopna podat pomocnou ruku v okamžiku, kdy se dostaneme mimo pole pokryté dokumentací.

Riziko výběru špatného frameworku:

ORM frameworků pro PHP je mnoho a výběr toho správného je mnohdy velký problém. Musíme si uvědomit, jaké funkce od něj očekáváme a zhruba jaký objem dat bude konečná aplikace obsahovat. ORM frameworky jsou poměrně mladé a stále se intenzivně vyvíjí a často přichází s novými funkcemi, proto je důležité vybrat framework se silnou komunitou a dostatečně ověřenou licencí, aby nebyly v dalším vývoji. Dále je dobré předem zjistit, jak framework pracuje s většími objemy dat a zda nenarazíme na paměťové limity.

Paměťová a procesorová náročnost:

ORM framework přidává do aplikace další vrstvu a v případě větších frameworků je vyšší paměťová náročnost znatelná. Při mapování záznamu tabulky na objekt dochází k takzvané hydrataci, každý objekt zabírá své místo v paměti a při načtení většího počtu záznamů můžeme narazit na paměťový limit pro běh jednoho skriptu. Navíc až do PHP 5.3 byly problémy s uvolňováním paměti, zejména u objektů s cyklickými vazbami. Nejedná se o nepřekonatelný problém, jen je vždy nutné s paměťovou náročností počítat. Větší množství pasivních záznamů (např. výpis položek) je lepší načíst do proměnné typu pole a pouze položky, na kterých budou volány funkce, načítat do objektů.

2.3 Návrhové vzory

Návrhových vzorů určených pro technologii ORM je velká spousta. Zde budou popsány jen ty nejvýznamnější a nejpoužívanější.

Návrhový vzor v softwarovém inženýrství představuje obecné řešení problému, které se využívá při návrhu počítačových programů. Nejedná se o knihovnu nebo část zdrojového kódu, kterou by jsme mohli vložit do našeho programu. Návrhový vzor popisuje řešení obecného problému nebo vytváří jakousi šablonu, která může být použita v různých situacích. Objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy.

Podle Martina Fowlera[1] se návrhové vzory, týkající se technologie ORM, dělí na návrhové vzory architektury a návrhové vzory chování. [2]

2.3.1 Návrhové vzory architektury

Návrhové vzory architektury popisují vazby mezi relačním databázovým úložištěm a objektovým modelem. Doménový model je souhrn tříd naplňující logiku vazeb, specifikuje atributy třídy, primární klíče, cizí klíče a další nutné vlastnosti, které přesně odpovídají tabulkám v databázi. Jeden objekt odpovídá jedné tabulce v databázi. Jednotlivé objekty, bychom pak měli vhodně rozšířit o logiku, kterou databáze nemohou zajistit.

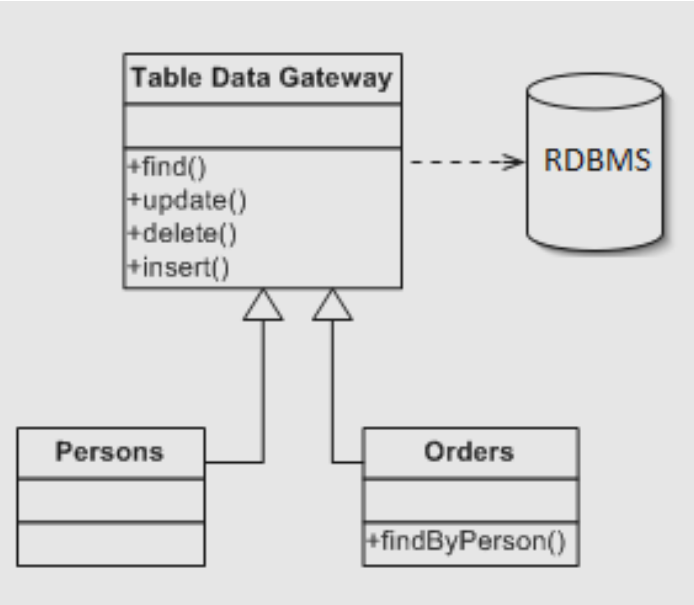
Mezi návrhové vzory architektury můžeme zařadit Table Data Gateway, Row Data Gateway, Active Record, Data Mapper.

2.3.1.1 Table Data Gateway

Tento návrhový vzor funguje jako brána, která zapouzdřuje databázové operace prováděné nad jednou tabulkou. Každá taková databázová tabulka je v aplikaci reprezentována samostatnou třídou. Tato třída pak obstarává CRUD operace s jednotlivými řádky tabulky. Metody, které v tabulce vyhledávají řádky podle primárního klíče nebo jakýchkoliv jiných kritérií by měli vždy vracet množinu řádků a to i v případě, že je výsledkem pouze jediný řádek. Odvozenou třídu lze doplnit o metody, které zjednoduší vyhledávání nad konkrétní tabulkou podle specifického kritéria.

[1] Martin Fowler je tvůrcem spousty návrhových vzorů používaných v ORM.

Zdroj obrázků popisovaných návrhových vzorů: http://martinfowler.com/eaaCatalog/

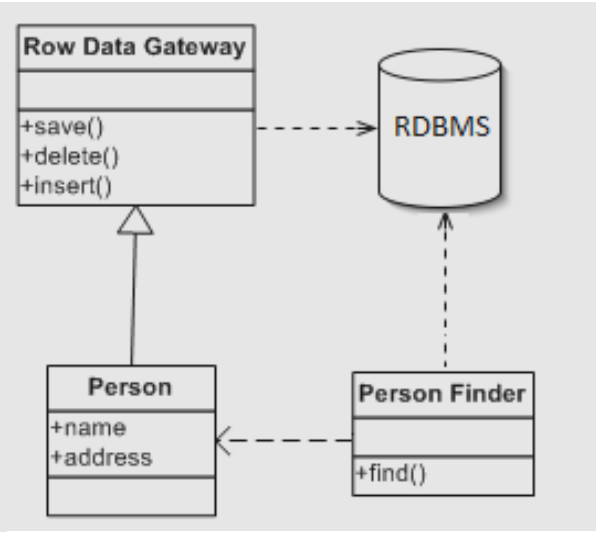


Obrázek č. 1 - Návrhový vzor Table Data

2.3.1.2 Row Data Gateway

Návrhový vzor Row Data Gateway funguje jako brána, která zapouzdřuje CRUD operace nad jedním řádkem databázové tabulky. Každý atribut dané třídy odpovídá danému sloupci v databázové tabulce. K atributů, se zpravidla přistupuje přímo bez nutnosti používat gettery a settery. Instance řádku se vytváří nejčastěji pomocí asociativního pole, které je předáno konstruktoru. Instance řádku tedy nevytváří přímo uživatel, ale vytváří ji oddělená vyhledávací třída. Samotná vyhledávací třída může implementovat vzor Table Data Gateway a je i vhodné tyto vzory kombinovat

Důvodem, proč je konstruktoru předáno pole hodnot a nikoliv primární klíč, je optimalizace databázového výkonu. Třída, která zajišťuje vyhledávání, provede pouze jeden databázový dotaz, který může vrátit více výsledků. Pro každý vrácený řádek vytvoří instanci bez nutnosti volání dalšího databázového dotazu.



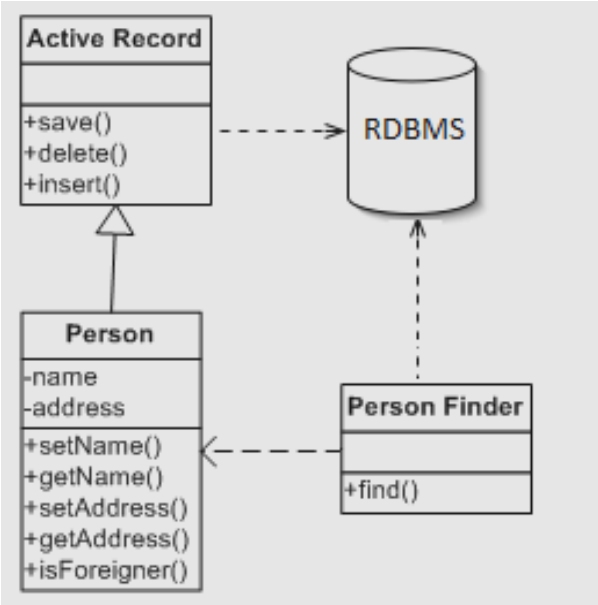
Obrázek č. 2 - Návrhový vzor Row Data

2.3.1.3 Active Record

Návrhový vzor Active Record je mezi vývojáři velmi oblíbený a jedná se o jeden z nejpoužívanějších návrhových vzorů v oblasti ORM. Active Record je podobný návrhovému vrozu Row Data Gateway, ale navíc implementuje vlastní business logiku. Jedná se tedy o doménový objekt, který obsahuje navíc CRUD operace. Vytváření instancí zajišťuje oddělená vyhledávací třída nebo statická metoda, která vyhledávací třídu deleguje.

Existence metod, které zajišťují CRUD operace uvnitř doménového objektu, jako je tomu u vzoru Active Record, není úplně ideální. Návrhový vzor Active Record tím porušuje pravidlo, kdy by měl mít každý objekt nanejvýš jednu zodpovědnost za určitou činnost. Objekt implementující vzor Active Record je ale zodpovědný jak za business logiku, tak za persistentní uchování dat. S doménovým objektem se často pracuje na různých úrovních aplikace a může být proto matoucí, aby doménový objekt obsahoval navíc metody pro uložení do databáze. Doménový objekt má reprezentovat entitu, která neví o existenci databázového systému a persistentním uchování dat. Z hlediska kvalitního objektového návrhu je použití návrhového vzoru Active record nežádoucí.

Další nepříjemností vzoru Active Record je omezení, kdy atributy doménového objektu musí přesně odpovídat sloupcům databázové tabulky. Tedy atributy doménového objektu a sloupce databázové tabulky musí odpovídat 1:1. Podle zvolené vývojové metodiky tedy musí být diagram tříd závislý na datovém modelu a obráceně. Pokud tedy dojde k jakékoliv změně v datovém modelu, je potřeba současně upravit doménový model a obráceně. Toto platí i pro výše zmíněné návrhové vzory (Table Data Gateway, Row Data Gateway).

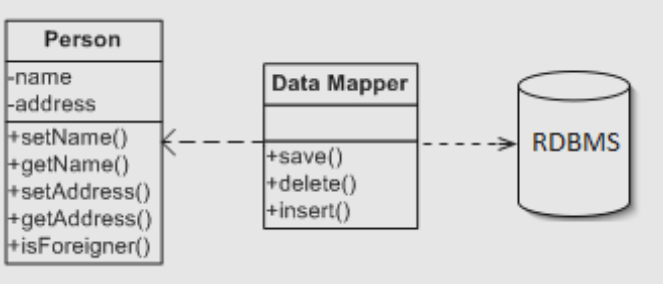


Obrázek č. 3 - Návrhový vzor Active Record

2.3.1.4 Data Mapper

Jedná se o návrhový vzor, kdy doménový objekt neobsahuje žádné CRUD nebo vyhledávací operace. O vytváření, úpravu a mazání doménových objektů z databáze se stará oddělený mapovací objekt. Doménový objekt je tedy zcela nezávislý na databázi. Mapovací objekt má přístup jak k doménovému objektu, tak i k databázovému systému. Tento návrh je vhodné použít pokud nelze zajistit zcela ekvivalentní nastavení objektů vůči databázi, například pro účely dědění, nebo jiných objektově orientovaných vztahů, které relační databáze neumí zachytit. Data Mapper je jednoduše prostředníkem mezi relačním schématem databáze a objektovým schématem. Využívá se hlavně pro účely, kdy tyto pohledy nejsou totožné.

Pokud je ale business logika schovaná v doménových objektech je pro jednodušší případy vhodný vzor Active Record. V případě složitějších objektů, nebo pokud potřebujeme nezávislost objektového a datového modelu, je nutné použít vzor Data Mapper.



Obrázek č. 4 - Návrhový vzor Data Mapper

2.3.2 Návrhové vzory chování

Tyto návrhové vzory se zabývají způsobem, jak efektivně data načítat a ukládat. Jak je jednoznačně identifikovat v paměti, jak zajistit konkurenční přístup. Nejvýznamnější návrhové vzory chování jsou např. Unit of Work, Lazy Load nebo Identity Map.

2.3.2.1 Unit of Work

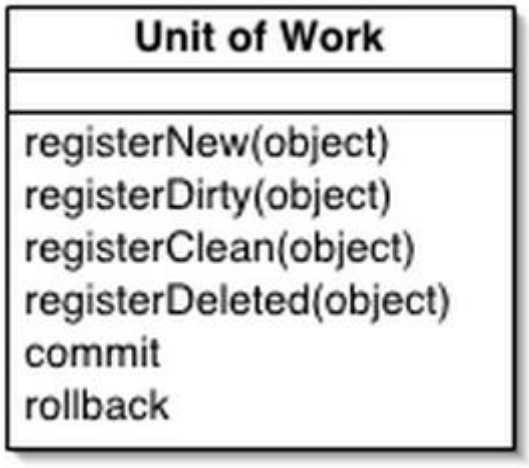
Množina entit v paměti se může během spuštění programu měnit. Některé entity jsou upraveny, jiné zase smazány a mohou vznikat i entity nové. Pokud odpovídající změny nebudou hned provedeny i v trvalém úložišti, budou po odstranění entit z paměti ztraceny.

Někdy je také, aby změny v trvalém úložišti proběhly najednou, v jedné transakci. Toto se špatně zajišťuje především tehdy, když jsou změny roztroušeny v různých částech kódu nebo v čase. Je-li totiž každá malá změna ihned propagována až do trvalého úložiště, snižuje se výkonnost. Všechny tyto problémy řeší návrhový vzor Unit of Work.

Řešením je vytvoření nové třídy, která bude zaznamenávat provedené změny a teprve na požádání tyto změny provede i v trvalém úložišti. Změny nemusí provádět přímo, ale delegovat je na jiné třídy, které mají změny daného typu entity na starosti. Po provedení změn se stav třídy vrátí zpět do výchozího stavu, beze změn. Do takové třídy je někdy příhodné přidat i metodu pro návrat do výchozího stavu bez uložení změn, což je praktické v případě, že se uživatel či program rozhodne vzít změny zpět a v trvalém úložišti je neprovádět.

Tato nová třída v sobě nějakým způsobem bude ukládat informace o stavu každé změněné entity. V typické aplikaci se jedná o tři základní stavy: nová (NEW), změněná (DIRTY) a odstraněná (REMOVED). Změnu je nutné třídě oznámit zvenku, a to jednou z metod *markAsNew* (označí danou entitu jako nově vytvořenou), *markAsDirty* (označí entitu jako změněnou) a *markAsRemoved*. Pokud změny na nějaké entitě nechceme provádět, hodí se i metoda *removeFromChanges,* která odstraní všechny vedené záznamy o změně entity. K potvrzení změn a jejich provedení v trvalém úložišti slouží metoda *commit*, která způsobí postupné provedení všech dílčích změn, zpravidla v jedné transakci. Ke smazání všech záznamů o změnách entit je určena metoda *rollBack.*

Pro jednoduchost se jako změněná označuje celá entity. Ve většině případů užití totiž není nutné zacházet do podrobností a rozlišovat, jakého atributu entity se změna týká. V podobných případech se ani nemusí sledovat, je-li entita následnou změnou opět vrácena do původního stavu, jako změněná zkrátka zůstane označená i nadále. To však nevadí.

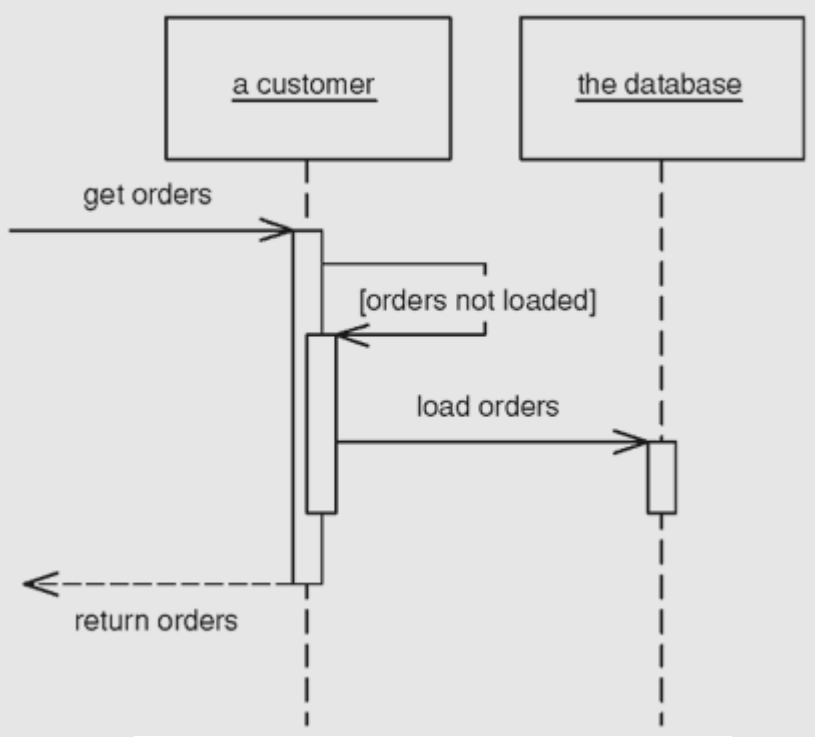


Obrázek č. 5 - Návrhový vzor Unit of Work

2.3.2.2 Lazy loading

Jedná se o návrhový vzor pro strategii načítání objektů z databáze až v momentě, kdy jsou opravdu potřeba. Tato strategie je velice důležitá při optimalizování výkonu aplikací pracujících s databází. Častým problémem je nahrávání entit, které jsou spojeny s větším množstvím jiných entit a tudíž nahrání takové entity vede k nahrání všech těchto spojených entit, čímž výkon aplikace razantně padá. *Lazy loading* tento problém řeší řízením dotazů do databáze, které jsou prováděny až v momentě, kdy jsou data spojená s nahrávanou entitou opravdu zapotřebí. Je však zbytečné zmíněný postup aplikovat, pokud jsou všechna data, která potřebuje uložena v jedná databázové tabulce. *Lazy loading* je užitečný tehdy, když na nahrání celé entity se všemi daty je potřeba více dotazů do několika tabulek databáze najednou.

Například, pokud bude mít třída uživatel vazbu na své objednávky, bude při zachování objektového přístupu obsahovat kolekci objektů objednávek. Při vytvoření instance uživatele, nebudou tedy jeho objednávky nahrávány ihned, ale až v případě kdy budou opravdu potřeba.



Obrázek č. 6 - Diagram aktivit vzoru Lazy

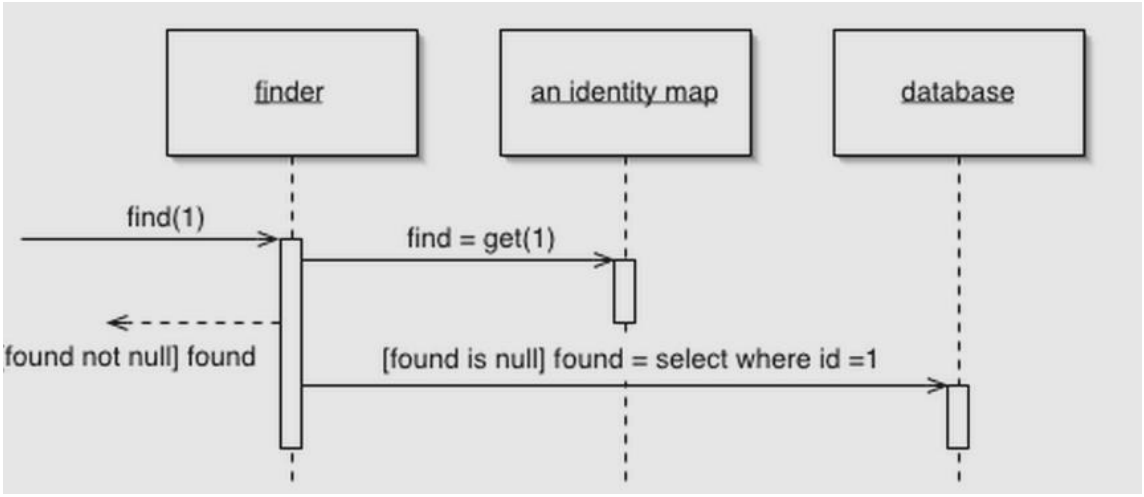
2.3.2.3 Identity map

Trvalé úložiště dat obsahuje entity, jejichž obrazy se vytváří v aplikaci. Tyto entity jsou rozlišeny klíčem.

Nic nebrání tomu, aby aplikace volně vytvářela instance entit a ty rozesílal do různých částí programu. Případná změna jedné takové instance se neprojeví v ostatních instancích, což může být

problém. Také je nutné při každém požadavku na entitu načítat aktuální data z trvalého úložiště, což je zbytečné a neefektivní, pokud se uložená instance mezitím nezměnila.

Řešení tohoto problému spočívá v odstínění programu od vytváření instancí entit. Vznikne nová třída, která bude instance entit spravovat a vytvářet je pouze tehdy, když instance entit s požadovaným klíčem neexistuje.



Obrázek č. 7 - Diagram aktivit vzoru Identity

Popis diagramu aktivit na obrázku 7. *Finder* hledá objekt s identifikačním číslem 1. Nejprve se podívá do své *mapy identit* a ověří, jestli už tento objekt z databáze nezískával dříve. Všechny získané objekty jsou uloženy do *mapy identit*, není tedy problém ověřit existenci tohoto objektu. Pokud se objekt v mapě *identit nenachází*, *Finder* ho získá (např. z databáze), uloží do *mapy identit* a předá ukazatel na něj aplikaci.

3 Doctrine 2

Jedná se o jednu z nejvýznamnějších ORM knihovnu pro PHP. Podporuje databáze jako MySQL, PgSQL, Oracle, Sqlite. Požadavek nezávislosti na databázi tedy splňuje. Oproti jejímu předchůdci knihovně Doctrine je Doctrine 2 už od začátku koncipována jako čisté ORM, bez jakékoliv funkčnosti, kterou lze realizovat ve vyšších vrstvách. A právě tohle čisté ORM je největší plus celé Doctrine 2.

Doctrine 2 pro svůj běh vyžaduje minimálně PHP 5.3. Na starších verzích ji nebude možné rozběhnou, protože využívá nové vlastnosti jazyka, jako jsou například namespaces, anonymní funkce nebo nový vylepšený *garbage collector*.

Doctrine 2 je postavena na návrhovém vzoru *Data Mapper* (byl popsán dříve), reprezentovaném vůči zbytku aplikace zejména fasádou v podobě Entity Manageru. Většina ostatních ORM využívá návrhový vzor *Active Record*.

Odstínění aplikace od databáze je dotaženo téměř k dokonalosti. Pokud nebudete chtít, nemusíte o databázi vůbec zavadit. Struktura databáze i veškeré její změny se generují automaticky z definic entit. Klasické SQL dotazy lze nahradit DQL (Doctrine Query Language) notací, kde se na místo názvů tabulek a sloupců využívají výhradně názvy entit a jejich členských proměných.

Celým systémem je kladen silný důraz na optimalizaci výkonu, což bývá slabou stránkou mnoha ORM systémů. V doctrine 2 je k dispozici *cachování* na několika úrovních. Lazy loading je použit všude, kde to jen jde. Je tu i možnost vynutit si vlastní optimalizovaný dotaz do databáze namísto standardního chování.

3.1 Tři vrstvy Doctrine 2

Knihovna doctrine 2 je rozdělena do tří vrstev. [5]

Common vrstva:

Definuje základní obecná rozhraní, třídy a knihovny. Najdeme zde například nástroje pro práci s kolekcemi, anotacemi, cachováním, událostmi apod. Tyto nástroje jsou dále využívány oběma vyššími vrstvami. Common vrstva je nezávyslá na ostatních vrstvách, může být tedy teoreticky použita i samostatně. Všechno co patří do této vrstvy je definováno v namespace *DoctrineCommon.*

DBAL vrstva (DataBase Abstraction Layer)

Abstrahuje zbytek aplikace od konkrétního typu databáze. Primárně rozšiřuje standardní PDO, ale umí pracovat i s dalšími databázovými drivery. Zavádí již zmíněnou notaci DQL. Vrstva DBAL je závislá na Common vrstvě. Je definována v namespace *DoctrineDBAL*.

ORM vrstva (Object Relation Mapping)

Nejvyšší vrstva, která zajišťuje právě mapování aplikačních objektů na relační databázi, jejich persistování a načítaní. ORM vrstva je závislá na obou předchozích vrstvách. Namespace této vrstvy je *DoctrineORM*.

3.2 Definice entit

Entity jsou základními stavebními kameny celé Doctrine 2. Každá entita reprezentuje mějaká objekt reálného světa. tzv. doménový objekt.

Při práci s Doctrine 2 pracujeme na takové vrstvě, kdy se pro modelování doménových objektů používají entity. Ty nás dokonale odstíní od konkrétní databáze i se všemi jejími podivnostmi, jako jsou např. cizí klíče (ve skutečném světě nic jako cizí klíč není) nebo nutnost rozdělit doménový objekt do více míst (tabulek). Práce s Doctrine 2 se tak stává systémově a logicky čistou.

Ukázka definice entity v Doctrine 2.

Entity v Doctrine 2 jsou běžné PHP objekty, jak je znáte a používáte. Pomocí komentářových anotací Doctrine 2 řekneme, jak s nimi má pracovat.

Jako příklad je uveden článek s titulkem a obsahem, který by jsme v databázi pomocí SLQ vytvořili následovně.

```
CREATE TABLE article (
      id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
      title VARCHAR(50) NOT NULL
      content TEXT NULL
);
```

Příslušnou entitu v Doctrine 2 pak definujeme následovně.

```
/**
* @entity
*/
class Article {

      /**
      * @id
      * @column(type="integer")
      * @generatedValue
      */
      private $id;

      /**
      * @column(type="string", length=50)
      */
      private $title

      /**
      * @column(type="text",
nullable=true)
      */
      private $content
}
```

Pomocí komentářových anotací, jsme Doctrine 2 sdělili, jaké vlastnosti má do databáze ukládat a že *$id* je primární klíč s automaticky generovanou hodnotou.

Vedle komentářových anotací nabízí Doctrine 2 ještě další možnosti definice entit, a to pomocí XML nebo YML formátu či prostřednictvím speciálních PHP volání.

Mapování entity na tabulku:

Doctrine 2 se snaží z názvu třídy sama odvodit jméno příslušné databázové tabulky, do které se budou data ukládat. Entita z příkladu výše tedy bude mapována na tabulku se jménem *article*. Jméno tabulky, na kterou bude daná entita mapována lze určit explicitně, a to pomocí speciální komentářové anotace *@table(name="jméno tabulky")*. Slabým místem Doctrine 2 jsou zde rezervovaná slova z SQL, která nemůžou být použita jako název databázové tabulky, pokud nebudou obalena do zpětných uvozovek. Doctrine takové escapování nezajišťuje, vývojář se musí postarat sám.

Anotace databázových sloupců:

Každý sloupce, který bude Doctrine 2 ukládat do databáze, musí být anotován pomoci *@column*. Vlastnosti takto anotovaného sloupce lze dále upřesnit.

- **type** - datový typ sloupce. Výchozí hodnota je *string*.
- **name** - název atributu v definici databáze. Standardně se použije stejný název, jako má samotná proměnná. Tady ho lze ale změnit a mapovat tak proměnnou na jiný databázový atribut.
- **unique** - udává, jestli má být nad sloupcem kontrolována unikátnost. Nabývá hodnot *true* nebo *false*. Výchozí hodnota je *false*.
- **nullable** - udává. zda může být do sloupce uložena hodnota *NULL*. Nabývá hodnot *true* a *false*. Výchozí hodnota je *false*.
- **length** - udává maximální délku řetězcových sloupců. Výchozí hodnota je 255.
- **precision** a **scale** - udavájí přesnost desetinných čísel. Výchozí hodnota je 0.

Příklad použití:

```
/**
* @column( name="author", type="string", length=100, unique=true )
*/
private $authorName;

/**
* @column(  type ="decimal", precision=2, scale=1, nullable=true)
*/
private $volume;
```

Datové typy sloupců:

Uvnitř anotace *@column(type="datový typ")* je možné použít jakýkoliv z předdefinovaných datových typů. Je potřeba si ale uvědomit, že se zde nejedná o databázové typy, nebo datové typy jazyka PHP. Jsou to speciální mapovací typy Doctrine 2, kterými určuje, co a jak se má z PHP mapovat do databáze. Například speciální mapovací typ *type="text"* říká, že Doctrine 2 má mapovat danou

proměnou, ve které je uložena hodnota PHP typu *string*, na databázový sloupec typu *CBLOB*. Přehled typů a jejich mapování mezi databází a PHP je vidět v tabulce 1.

Doctrine 2 typ	PHP typ	Databázový typ
string	string	VARCHAR
text	string	CLOB
integer	integer	INT
smallint	integer	SMALLINT
bigint	string	BIGINT
decimal	double	DECIMAL
boolean	boolean	BOOLEAN
date	DateTime	DATETIME
time	DateTime	TIME
datetime	DateTime	DATETIME, TIMESTAMP
datetimetz	DateTime	DATETIME, TIMESTAMP
object	object	CLOB
array	array	CLOB

tab. č. 1 - přehled datových typů Doctrine 2

3.3 Základní operace s entitami

Ve většině ostatních ORM frameworků, kde je použit přístup *Active Record*, si základní operace jako jsou ukládání, načítání a mazání, zajišťuje entita sama svými metodami. Doctrine 2 od přístupu Active Record zcela opouští. Nechává entitu samu o sobě nezávislou na jakémkoliv načítání, ukládání či mazání. Nezávislou na způsobu persistování. Entita si tak řeší jen nastavování a vracení svých dat nebo jejich základní konverze a validace. Načítání, ukládání a mazání se pak zajišťuje vně entity za pomoci takzvaného *Entity Manageru*.

Takový přístup je z hlediska celkového návrhu mnohem čistší, protože načítání, persistování nebo mazání dovnitř entity skutečně nepatří. Entita neukládá sama sebe, ale je někým uložena.

Entity Manager je jen fasáda, která umožňuje snadný a rychlý přístup k jednotlivým částem, ze kterých je Doctrine 2 složena uvnitř. Stará se o persistenci jednotlivých objektů. Implementuje

návrhové vzory Data Mapper a Unit of Work. Dále eviduje všechny instance entitních tříd a udržuje informaci o jejich stavu. V okamžiku zavolání metody *flush()* dojde k persistování všech objektů.

Ve všech případech Doctrine 2 zajišťuje, že od jedné entity s jedním ID bude v aplikaci vždy a pouze jen jedna instance, která se všude předává pomocí referencí. Situace, kdy by byly v aplikaci dvě různé instance entity se stejným identifikátorem, nemůže nastat. Doctrine takové chování řídí na pozadí pomocí *Identity Map*. Tu si lze představit jako asociativní pole, kde je každá načtená entita uchována pod kombinací názvu své třídy a ID. Existence *Identity Map* nemusí vývojář vůbec řešit, Doctrine 2 ji používá naprosto transparentně.

Ukázka základních operací s entitami:

```
/* Vytvoření nového článku*/
$article = new Entity\Article;
$article->setTitle( "title_1" );
$article->setContent( "text text" );

/* Uložení článku pomocí entity
manageru*/
$em->persist($article);
$em->flush();

/* Editace článku s ID 1*/
$article = $em->find( "article", 1 );
$article->setTitle( "Nový titulek" );
$em->flush();

/* Smazání článku s ID 1*/
$article = $em->find( "article", 1 );
$em->remove($article);
$em->flush();
```

Jak je patrné z příkladu výše, pro potvrzení všech provedených změn je potřeba nakonec zavolat metodu *$em->flush()*. Typickým případem je, že během jednoho skriptu je provedeno více různých změn. Tyto změny se řadí do fronty. Zavoláním *$em->flush()* jsou všechny změny potvrzeny a odeslány do databáze. V případě, kdy by metoda $em->flush() nebyla zavolána, tak by se žádná ze změn neuložila do databáze a všechny provedené změny u všech persistovaných entit by se ztratily.

3.4 Stavy entit a UnitOfWork

Z pohledu *EntityManageru* můžou být entity v různých stavech Informace o stavu se uchovávají mimo entitu, přesněji si je drží *UnitOfWork*. Návrhový vzor *Unit of work* byl popsán výše.

UnitOfWork se nachází na pozadí *EntityManageru* a zjednodušeně si jej lze představit jako frontu všech změn, které chceme uložit do databáze. Například pokud bude persistována nová entita nebo změněna již persistovaná entita, neprovádí se hned příslušný dotaz do databáze, ale všechny změny se hromadí v *UnitOfWork*. Do databáze se pak odešle vše najednou po zavolání *$em->flush().* Takový přístup přináší hned několik podstatných výhod. Jedna z nich je, že všechny databázové operace jsou prováděny v relativně krátkém časovém úseku, takže případné zamykání je použito jen na nezbytně nutnou dobu. Doctrine 2 si při zavolání *$em->flush()* dokáže lépe rozvrhnou a výkonnostně zoptimalizovat, v jakém pořadí, co a jak do databáze pošle. Další výhodou fronty v *UnitOfWork* je možnost vzít všechny změny snadno zpět. Buď se jednoduše nezavolá *$em->flush()* nebo je tu možnost vyčistit *EntityManager* pomocí volání *$em->clear()*

EntityManager podle stavu každé entity rozhoduje, jak bude s entitou dále nakládat. Doctrine 2 rozlišuje celkem čtyři různé stavy entit. Těmito stavy jsou, nová entita, spravovaná, odpojená a smazaná.

Nová

Jako nová je označena taková entita, která byla právě vytvořena operátorem *new* a doposud na ni nebylo zavoláno *$em->persist($novaEntita)*, takže zatím není pod kontrolou *EntityManageru.*

Spravovaná

Ve stavu Spravovaná se nachází nová persistovaná nebo načtená již existující entita. Jakékoliv změny, které v nich budou provedeny, se v při prvním zavolání *$em->flush()* promítnou do databáze.

Odpojená

Spravovanou entitu lze od *EntityManageru* odpojit zavoláním *$em->detach($entity)*. Daná instance entity sice v aplikaci nadále existuje až do konce požadavku, ale jakékoliv změny v ní provedené se nikam neuloží a po skončení běhu skriptu se ztratí. V databázi ale záznam takové entity nadále existuje a je možné jej kdykoliv znovu načíst. Odpojenou entitu je také možné znovu připojit k *EntityManageru* pomocí zavolání *$em->merge($entity).*

Smazaná

Po zavolání *$em->remove($entity)* nad spravovanou entitou, se entita přepne do stavu Smazaná a při prvním zavolání *$em->flush()* bude vymazána i z databáze. V paměti její instance zůstane až do konce požadavku.

Aktuální stav libovolné entity lze jednoduše zjistit zavoláním metody *$em->getUnitOfWork->getEntityState($entity)*. Vrací se hodnota jedné z následujících symbolických konstant:

- UnitOfWork::STATE_NEW (Nová)
- UnitOfWork::STATE_MANAGED (Spravovaná)
- UnitOfWork::STATE_DETACHED (Odpojená)
- UnitOfWork::STATE_REMOVED (Smazaná)

Ukázka životního cyklu instance pro demonstraci jednotlivých stavů entity:

```
$unit = $em->getUnitOfWork();

$article = new Entity\Article;
echo $unit->getEntityState( $article );    //STATE_NEW

$em->persist( $article );
echo $unit->getEntityState( $article );    //STATE_MANAGE

$em->detach( $article );
echo $unit->getEntityState( $article );    //STATE_DETACHED

$article = $em->merge( $article );
echo $unit->getEntityState( $article );    //STATE_MANAGED

$em->remove( $article );
echo $unit->getEntityState( $article );   //STATE_REMOVED
```

3.5 Implicitní a explicitní transakce

3.5.1 Implicitní transakce

Transakce jsou automaticky využívány ve všech databázích, v kterých jsou podporovány. Ve výchozí situaci se o databázové transakce nemusí vývojář vůbec starat. Doctrine 2 po zavolání *$em->flush()* transakci na začátku zpracování všech změn naplánovaných v *UnitOfWork* sama otevře a po úspěšném provedení commitne.

Pokud během zpracovávání dat dojde k jakékoliv chybě, provede se nad databází *rollback*, *EntityManager* se uzavře a vyhodí se vyjímka.

3.5.2 Explicitní transakce

Doctrine 2 nabízí i možnost, kdy si může vývojář řídit transakce sám. K manuálnímu řízení transakcí jsou k dispozici následující metody.

- *$em->getConnection()->beginTransaction()*
- *$em->getConnection()->commit()*
- *$em->getConnection()->rollback()*

Jakmile je transakce explicitně otevřena zavoláním *$em->getConnection()->beginTransaction()*, tak se už neprovádí implicitní *commit* ani *rollback* uvnitř metody *$em->flush()* a vývojář si vše musí řídit sám.

3.6 Základní asociace mezi entitami

Pokud je mezi dvě entitami jakýkoliv vztah, vždy musí být jedna z nich vlastnící a druhá inverzní strana. Pro práci s asociacemi je velmi důležité tyto dvě strany navzájem rozlišovat, a to hned z několika důvodů:

- Na každé straně se asociace definuje rozdílnými anotacemi.
- Na vlastní straně je definice vždy povinná. Na inverzní straně je definice asociace volitelná v případě, že se jedná o jednosměrnou asociaci.
- Po zavolání *$em->flush()*, se do databáze promítnou jen změny provedené na vlastnící straně, zatímco změny provedené na inverzní straně se ignorují.

Příklad obousměrné 1:N asociace mezi entitami klient a účet. Entita Účet je zde vlastnící stranou.

```
/** entita Ucet **/
class Ucet {
      /**
      * @id @column(type="integer")
      * @generatedValue
      */
      private $id;

      /**
      * @manyToOne(targetEntity="Klient", inversedBy="ucty")
      */
      private $klient;
}

/** entita Klient **/
class Klient {
      // ... definice id

      /**
      * @oneToMany(targetEntity="Ucet", mappedBy="klient")
      */
      private $ucty;
}
```

Díky obousměrné asociaci lze nyní jednoduše zjistit vlastníka účtu *$klient* nebo množinu všech účtů které klient vlastní *$ucty*. Obousměrné asociace jsou zpravidla výkonově náročnější, než asociace jednosměrné. Je tedy dobrým zvykem vytvářet obousměrné asociace jen tam, kde je to opravdu potřeba.

Definice asociací typu 1:1 nebo M:N vypadají velmi podobně a nebudou zde dále probírány. Práce s asociačními proměnnými jako jsou *$klient* nebo *$ucty* z příkladu výše, je poměrně průhledná a není potřeba ji zde jakkoliv rozebírat. Podrobnější informace a další příklady naleznete v dokumentaci Doctrine 2 [4].

3.7 dotazovací jazyk DQL

Jedná se o velice silný nástroj celé Doctrine 2. Dotazovací jazyk DQL v sobě kombinuje přímočarost dotazovacího jazyka SQL a nezávislost objektové entitní vrstvy modelu. Je potřeba zdůraznit, že DQL není žádným dialektem standardního SQL. Jedná se o zcela samostatný jazyk, založený na jiných principech a ležící v úplně jiné vrstvě aplikace.

SQL se dotazuje na jména databázových tabulek a jejich atributů. DQL se místo tabulek používají jména Doctrine 2 entit a jejich členských proměnných. Výsledkem DQL dotazu jsou zpravidla načtené instance entit. Při spojování více tabulek pomocí SQL je výsledkem jedna velká tabulka, která obsahuje všechno. V DQ" se elegantně odkazuje na definované asociace a výsledkem je více vzájemně provázaných entit.

Příklad dotazu na všechny články a jejich kategorie v SQL:

```
SELECT * FROM clanek, kategorie WHERE clanek.kategorie_id =
kategorie.id;
```

Je potřeba používat jména tabulek a atributů, explicitně tabulky spojovat přes cizí klíče a výsledkem bude jedna velká tabulka.

Stejný příklad zapsaný pomocí DQL:

```
SELECT c, k FROM Clanek c JOIN c.kategorie k;
```

Použity jsou názvy tříd s entitami, názvy jejich členských proměnných. Asociace je zde použita naprosto přirozeně. Výsledkem budou instance nalezených článků, vedle nich instance nalezených kategorií. Související články a kategorie budou navzájem referencovány v asociačních sloupcích.

S pomocí DQL je možné provádět pouze dotazy typu SELECT, UPDATE a DELETE. Není možné vytvářet nové záznamy pomocí INSERT dotazu. Jednalo by se o zakládaní nových entit bez povědomí *EntityManageru* a to by vedlo k nepořádku při správě persistování entit uvnitř Doctrine 2. Stejně tak nejsou k dispozici žádné příkazy jako CREATE, ALTER, DROP, GRAND a podobné. Tyto příkazy do aplikace využívající Doctrine 2 vůbec nepatří.

I u zmiňovaných UPDATE a DELETE dotazů může nastat spousta problémů, které souvisejí s prováděním hromadných změn přímo do databáze bez *EntityManageru* a *IdentityMap*. Proto je dobré tyto typy dotazu používat opatrně. Pro běžné používání už zbývá z celého DQL jen dotaz typu SELECT.

DQL dotazy se vytváří pomocí *EntityManageru* voláním metody *$em->createQuery()*. Takový přístup umožňuje vytvoření dotazu ještě předtím, než se provede. Je tedy možné ho dodatečně upravovat.

```
$query = $em->createQuery( 'SELECT c FROM Clanek c' );
$clanky = $query->getResult();
```

3.8 Query Builder

Jedná se o další možnost, jak v Doctrine 2 připravit dotaz do databáze. *Query Builder* je speciální třída, která umožňuje vytvořit databázový dotaz postupným voláním jejích metod.

Nejprve je nutné vytvořit instanci *Query Builderu* voláním *$em->createQueryBuilder()*. Nad touto instancí se pak volají další metody a postupně se tak skládá výsledný dotaz. Jako další je potřeba vybrat typ dotazu, a to pomocí volání jedné z metod *select(), update()* nebo *delete()*. V dalších krocích je možné dále nastavovat různá nastavení a omezení.

Příklad Query Builderu:

```
$query = $em->createQueryBuilder()
       ->select( 'c' )
       ->from( 'Clanek', 'c' )
            ->where( 'c.id' = ?1 )
       ->setParameter( 1, ' id_Clanku' )
       ->getQuery();
```

Zavoláním metody *getQuery()* se vygeneruje výsledná instance třídy *DocrineORMQuery*. Výsledkem je tedy totéž, jako po sestavení DQL dotazu.

3.9 Kdyby/Doctrine

Kdyby/Doctrine je doplněk pro PHP framework Nette [9], který do tohoto frameworku integruje právě Doctrine 2 [7]. Rozšíření Kdyby/Doctrine přidává nové možnosti, co se týkají architektury aplikací. Hlavním cílem je snaha o odstínění od přímého používání *EntityManageru* a rozšíření repozitářů o další operace (ukládání, mazání). Tímto už se nejedná pouze o repozitář, ale takzvaný Data Access Object (DAO).

V konfiguračním souboru Nette frameworku vzniká díky rozšíření Kdyby\Doctrine možnost předávání zmíněných DAO službám. Konfigurační soubor, by tedy mohl vypadat takto:

```
common:
      extensions:
            doctrine: Kdyby\Doctrine\DI\OrmExtension

      services:
            bar: BarModel( @doctrine.dao( "Entities\\FooEntity" ) )
```

Takto by se registrovala služba *bar*, která bude vracet model *BarModel*, kterému bude v konstruktoru předán objekt DAO pro entitu *FooEntity* z namespace *Entities*. Tímto způsobem je možné z velké části odstínit aplikaci od přímého používání *EntityManageru*.

3.10 Integrace Doctrine 2 do Nette

Nejjednodušší cesta, jak integrovat ORM Doctrine 2 do PHP frameworku Nette, je právě již zmiňovaný doplněk Kdyby/Doctrine. Instalace je velice jednoduchá a provádí se za pomoci *Composeru*.

```
$ composer require kdyby/doctrine: ~2.1
```

V konfiguračním souboru Nette frameworku je ještě potřeba přidat daná rozšíření a přístupové údaje k databázi. Mělo by to vypadat následovně:

```
extensions:
      console: Kdyby\Console\DI\ConsoleExtension
      events: Kdyby\Events\DI\EventsExtension
      annotations: Kdyby\Annotations\DI\AnnotationsExtension
      doctrine: Kdyby\Doctrine\DI\OrmExtension

doctrine:
      user: root
      password: pass
      dbname: database
      metadata:
            App: %appDir%
```

V sekci *metadata* se uvádí cesta k entitám a jejich *namespace*.

4 Propel

Jedná se o další oblíbený ORM framework. Propel [6] je na rozdíl od Doctrine 2 založený na návrhovém vzoru Active Record, který byl popsán výše. Řadí se do skupiny komplexních ORM frameworků a svými možnostmi převyšuje většinu ostatních frameworků založených na stejném návrhovém vzoru. Robustnost a rozsáhlost tohoto frameworku klade vysoké nároky na vývojáře. Pro práci s Propelem je nutná dobrá znalost objektově orientovaného programování. Vývojář začínající s tímto frameworkem je nucen nastudovat několik desítek stránek dokumentace a tutoriálů.

4.1 Schéma databáze

Na samotném začátku práce s Proplem je nejprve nutné vytvořit tzv. schéma databáze. Jde o popis struktury datového modelu. Takový popis je možné zapsat buď v XML nebo YAML formátu. Na základě tohoto popisu, dokáže Propel automaticky vygenerovat tzv. modelové třídy, které jsou primárním rozhraním pro manipulaci s daty v databázi. Schéma databáze musí obsahovat jak všechny tabulky a sloupce, tak i vztahy mezi tabulkami. Z tohoto přístupu tedy vyplívá, že vývojář je nucen ze všeho nejdřív navrhnout relační model databáze. Tento model je ale pouze dvourozměrný a neodpovídá úplně reálnému světu. Příkladem může být vztah M:N, který nelze přirozeně vyjádřit a je zde zapotřebí využít další vazební tabulky, která slouží pouze jako pomůcka a neodpovídá realitě. Vývojář je tak na začátku nucen uvažovat omezení relační databáze, místo aby se snažil popsat skutečnost co nejvěrohodněji s využitím objektově orientovaného přístupu.

Ukázka jednoduchého schémata databáze ve formátu XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="knihkupectvi" defaultIdMethod="native">

  <table name="kniha" phpName="Kniha">
    <column name="id" type="integer" required="true" primaryKey="true"
            autoIncrement="true"/>
    <column name="nazev" type="varchar" size="255" required="true" />
    <column name="autor_id" type="integer" required="true"/>

    <foreign-key foreignTable="autor">
      <reference local="autor_id" foreign="id"/>
    </foreign-key>
  </table>

  <table name="autor" phpName="Autor">
    <column name="id" type="integer" required="true" primaryKey="true"
            autoIncrement="true"/>
    <column name="Jmeno" type="varchar" size="128" required="true"/>
  </table>
</database>
```

Schéma popisuje dvě tabulky *kniha, autor* a vztah mezi nimi. Obě tabulky obsahují sloupec *id*, který je jejich primárním klíčem *primaryKey="true"* a jeho hodnota je generována automaticky *autoIncrement="true"*. Vztah typu 1:N je zde vyjádřen pomocí cizího klíče *foreign-key*.

4.2 Základní operace

4.2.1 Vytvoření a uložení objektu

Přidání nových dat (řádků) do databáze je velice jednoduché. Je třeba vytvořit instanci některé z Proplem automaticky vygenerovaných modelových tříd. Pomocí automaticky vygenerovaných metod (setterů) naplnit atributy (sloupce) příslušnými daty a voláním metody *save()* objekt uložit do databáze. Výsledkem metody *save()* je na pozadí vygenerovaný a v zápětí vykonaný příslušný SQL příkaz.

Ukázka vytvoření a persistování objektu:

```
$autor = new Autor();
$autor->setJmeno('Jan');
$autor->setPrimeni('Novák');
$autor->save();
```

Příslušný vygenerovaný SQL příkaz:

```
INSERT INTO autor (jmeno, prijmeni) VALUES (Jan, 'Novák');
```

4.2.2 Získávání dat z databáze

Nejjednodušší způsob jak získat data z databáze, je za použití již Proplem automaticky vygenerovaných dotazovacích metod. Jedná se o jednoduché dotazy typu výběr na základě hodnoty některého z atributů či relace. Jedna z takových metod je například metoda *findPK()*.

Ukázka získání objektu autora s id = 1:

```
$q = new AutorQuery();
$prvniAutor = $q->findPK(1);
```

Pro složitější dotazy Propel nabízí rozhraní zvané Propel Query API. Dotaz se skládá pomocí voláním příslušných metod.

Ukázka rozhraní Propel Query API.

```
$autori = AutorQuery::create()
  ->filterByJmeno('Jan')
  ->find();


$autori = AutorQuery::create()
  ->orderByPrijmeni()
  ->limit(10)
  ->find();
```

V prvním případě je výsledkem kolekce autorů, jejichž křestní jméno je Jan. V případě druhém je to kolekce deseti autorů seřazená podle jejich příjmení.

Propel umožňuje i zápis vlastního čistého SQL dotazu, který se provede tak, jak je napsán. Tento způsob bude výhodný u příliš složitých dotazů, u kterých by byl zápis pomocí Propel Query API zbytečně komplikovaný. Dále je možné tento způsob využívat pokud se vývojář potřebuje využít optimalizace na úrovni databázového jazyka. V takovém případě framework Propel dotaz nijak neparsuje a ani nepřizpůsobuje konkrétnímu databázovému systému. Databázová abstrakční vrstva se tímto zcela obchází a pracuje se na úrovni databáze.

4.2.3 Aktualizace a mazáni dat

Aktualizace databázového řádku v podstatě zahrnuje, získání daného objektu, změnu obsahu a opětovné uložení. V praxi to pro Propel znamená kombinaci dvou předchozích kapitol. Mazání objektu funguje podobně. Jedná se o dosti průhledné operace viz. ukázka.

Ukázka aktualizace a mazání dat.

```
$autor = AutorQuery::create()->findOneByJmeno('Jan');
$autor->setPrijmeni('Neruda');
$autor->save();


AutorQuery::create()
  ->filterByJmeno('Jan')
  ->update(array('Prijmeni' => 'Neruda'));


$autor = AutorQuery::create()->findOneByJmeno('Jan');
$autor->delete();


AutorQuery::create()
  ->filterByJmeno('Jan')
  ->delete();
```

4.3 Základní vztahy

Na základě definovaných cizích klíčů ve schématu databáze, Propel generuje další inteligentní metody, do modelových tříd, které slouží pro práci se vztahy. V praxi to znamená, že vývojář nebude muset nikdy pracovat přímo s cizím klíčem. Takový přístup činí práci s asociacemi velmi jednoduchou.

Propel automaticky vytváří settery na související objekty, které zjednodušují manipulaci s cizím klíčem. Hodnotu cizího klíče není vůbec potřeba definovat. Místo toho stačí nastavit související objekt, následovně:

```
$autor = new Autor();
$autor->setJmeno("Jan");
$autor->setPrijmeni("Novák");
$autor->save();

$kniha = new Kniha();
$kniha->setNazev("Vojna a mír");
// asociace kniha - autor
$kniha->setAutor($autor);
$kniha->save();
```

Metodu *setAutor()* Propel vygeneroval na základě artibutu <foreign-key> ve schématu databáze.

V tomto ukázkovém příkladu se jedná o vztah typu 1:N ze strany autora. Ale ze strany knih to bude vztah N:1, kdy více knih má stejného autora. Propel tedy negeneruje pouze metodu *setAutor()*, ale generuje taky metodu i pro druhou stranu. V tomto případě je to metoda *addKniha()*.

```
$kniha = new Kniha();
$kniha->setNazev("Vojna a mír");
$kniha->save();

$autor = new Autor();
$autor->setJmeno("Jan");
$autor->setPrijmeni("Novák");
//asociace autor - kniha
$autor->addKniha ($kniha);
$autor->save();
```

Metodu *save()* není potřeba volat pro každý objekt. V předchozí ukázce by stačilo zavolat jen *$autor->save()* a framework by automaticky kaskádovitě uložil do databáze i všechny ostatní související objekty.

4.4 Instalace frameworku Propel

Požadavky pro používání frameworku Propel 2.0

- PHP 5.4 nebo novější
- povolený modul DOM (libxml2)
- podporovaná databáze
    - MySQL
    - MS SQL Server
    - PostgreSQL
    - SQLite
    - Oracle

Propel používá některé PDO a SPL součásti, které jsou obsaženy ve výchozím nastavení PHP5

Pro instalaci samotného frameworku je doporučován Composer z důvodu správy závislostí projektu. Soubor *composer.json* by vypadal následovně.

```
{
    "require": {
        "propel/propel": "~2.0@dev"
    }
}
```

Kořenový adresář frameworku Propel obsahuje následující adresářovou strukturu:

Složky	obsah
bin	Obsahuje tři skripty, spravující Propel nástroj příkazového řádku
documentation	Dokumentace
features	Behat testy
resources	Obsahuje některé soubory jako například databáze XSD nebo DTD
src	Zdrojové soubory frameworku Propel
tests	Unit testy

tab. č. 2 - adresářová struktura Propelu

Propel dále očekává konfigurační soubor s názvem *propel.ext*, který je uložený na stejné úrovni jako *schema.xml* nebo v podadresáři *config*. Ukázka konfigurace ve formátu YAML pro MySQL:

```
propel:
  database:
      connections:
          knihkupectvi:
              adapter: mysql
              classname: Propel\Runtime\Connection\ConnectionWrapper
              dsn: "mysql:host=localhost;dbname=my_db_name"
              user: my_db_user
              password:
              attributes:
  runtime:
      defaultConnection: knihkupectvi
      connections:
          - knihkupectvi
  generator:
      defaultConnection: knihkupectvi
      connections:
          - knihkupectvi
```

Další informace a podrobnější postup instalace a konfigurace frameworku Propel zle dohledat přímo v dokumentaci frameworku [6]. Přidávám ještě odkazy, kde jsou popsány možnosti integrac knihovny Propel do aplikačního rámce Zend 2. [10]

5 Vlastní aplikace

Jedná se o webovou aplikaci, informační systém pro evidenci skladu lahvových vín, jednoho menšího rodinného vinařství. Systém uchovává aktuální stav skladových zásob produktů (lahvových vín), které jsou tříděny po jednotlivých odrůdách a ročnících. Na základě aktuálního stavu skladu, je systém schopen zpracovávat zadané objednávky, výdejky nebo dobropisy a k nim generovat příslušené dokumenty, jako jsou například faktury. Aplikace dále poskytuje inteligentní seznam odběratelů, kdy jsou v popředí nabízeni odběratelé s nejvyšší četností objednávek. Užitečnou funkcí je i možnost generování různých sestav. Například přehled prodaných produktů za určité období nebo sestavení inventury k aktuálnímu datu. Sestavení inventury je možné i zpětně ke staršímu datu.

Aplikace je postavena na populárním PHP frameworku Nette [8]. Framework Nette je založený na návrhovém vzoru MVP (model - view - presenter). Tento aplikační rámec jsem zvolil hned z několika důvodů. Jedním z nich byla právě jeho rostoucí popularita, dále pak silná základna českých vývojářů a v neposlední řadě dokumentace k tomuto frameworku, která je napsána pro mě skvělým způsobem, doplněná mnoha ukázkovými příklady a také je z velké části v českém jazyce.

Pro objektově-orientované mapování na datové vrstvě byla zvolena ORM knihovna Doctrine 2 [4]. Tuto knihovnu jsem takové vybral hned z několika důvodů. Doctrina 2 patří k těm oblíbenějším ORM frameworkům. Velkým důvodem bylo, že na rozdíl od Doctrine 1 a většiny ostatních ORM frameworků není Doctrine 2 postavena na návrhovém vzoru Active Record, ale na návrhovém vzoru Data Mapper, což vede k mnohem čistšímu návrhu aplikace. Oba zmíněné návrhové vzory, byly popsány v kapitole 2.3. Doctrine 2 mi byla doporučena i vedoucím mé bakalářské práce.

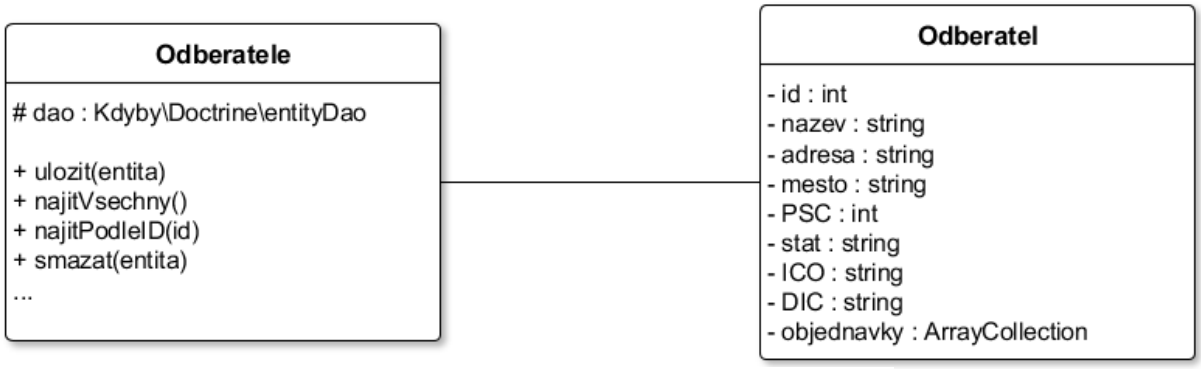
5.1 Vlastní aplikace a Doctrine 2

Pro integraci ORM Doctrine 2 do frameworku Nette existuje doplněk Kdyby\Doctrine [7] (popsán v kapitole 3.9). Tento doplněk jsem použil i ve své aplikaci. Instalace se provádí jednoduše za pomoci Composeru:

```
"require": {
     "kdyby/doctrine": "~2.1"
}
```

Rozšíření Kdyby\Doctrine nám umožňuje se do jisté míry odstínit od přímého používání *EntityManageru*. Ve vlastní aplikaci používám takový způsob, kdy nad entitní třídou vytvářím ještě další třídu tzv. fasádní třídu, která bude obsahovat metody pro manipulaci s danou entitou (uložit,

smazat, změnit, vyhledat). Tato třída v konstruktoru přímá objekt DAO[12] (viz. kapitola 3.9) pro danou entitu, nad kterou operuje.



Obrázek č. 8 - fasádní třída s entityDao

Třída *Odberatele* je zde fasádní třídou, nad entitní třídou *Odberatel*. Třídu *Odberatele* mam dále v Nette frameworku zaregistrovanou jako službu, kterou už si můžu jednoduše injectnout do presenteru a pracovat s ní.

Ukázka registrace služby:

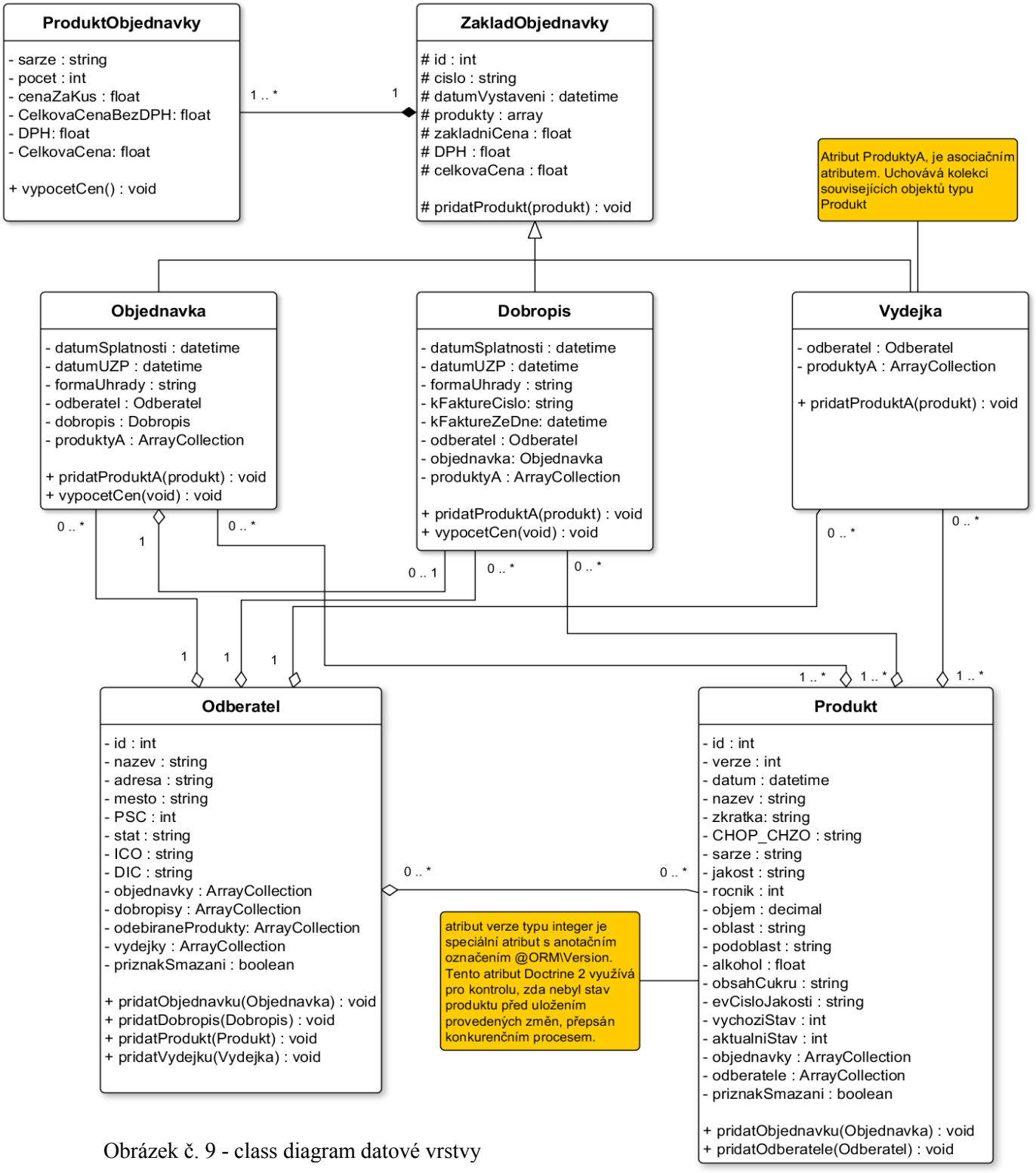
```
services:
	- EntityFacade\Odberatele(@doctrine.dao(Entity\Odberatel))
```

Ukázka injectnutí (předání závislosti) služby do presenteru

```
use EntityFacade\Odberatele;
class OdberatelPresenter extends BasePresenter
{
	/**
	 * @inject
	 *  @var Odberatele
	 */
	public $DBodberatele;
...
}
```

Anotace @inject je způsob získávání závislostí specifický pro DI Container v Nette [9].

5.2 Class diagram datové vrstvy (entity)



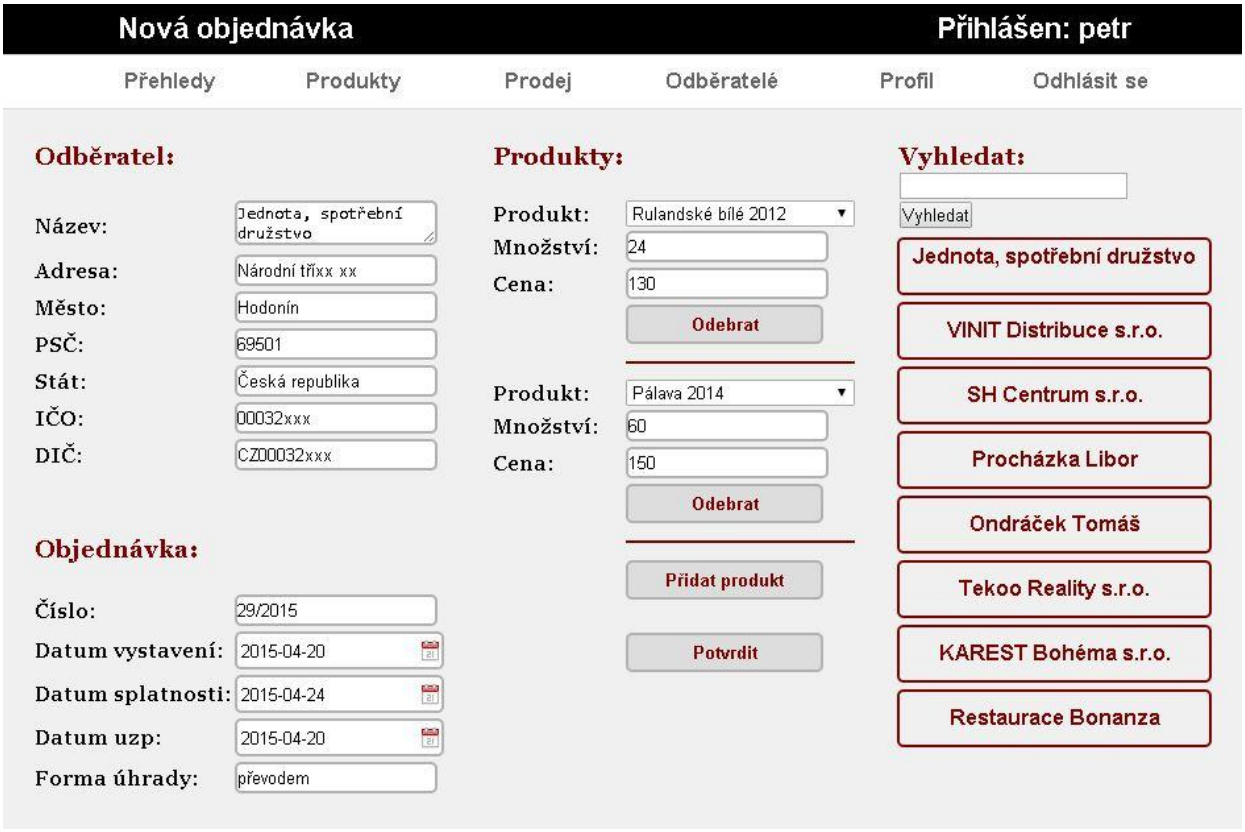
Obrázek č. 9 - class diagram datové vrstvy

Příloha č. 2 ukazuje definici entit Objednávka (OrderE), Produkt (Product) a Odběratel (Subscriber). Definice všech entit lze nalézt na přiloženém CD v adresáři app/model/Entity.

5.3 Popis nejdůležitějších funkcí

5.3.1 Zpracování objednávky (dobropis, výdejka)

Jedná se o jednu z hlavních funkcí systému, která se snaží o co nejjednodušší způsob vyřízení objednávky. Na obrázku jde vidět, že stránka je logicky rozdělena na 4 části. Konkrétně se jedná o části rychlý výběr odběratele, odběratel, objednávka a část produkty.



Obrázek č. 10 - zpracování objednávky

Vpravo se nachází sloupec rychlého výběru odběratele. Sloupec bez zadaného vyhledávání zobrazuje osm nejaktivnějších odběratelů. Aktivita se počítá podle počtu uskutečněných objednávek. Pomocí vyhledávacího pole, lze dohledat jakéhokoliv odběratele uloženého systému. Jednotlivé položky seznamu (sloupce) jsou aktivními odkazy, které po kliknutí vyplní formulář odběratele příslušnými daty.

V případě nového odběratele existují dvě možnosti. Uživatel může vložit nového odběratele v sekci *Odběratelé* pomocí funkce *Nový* a pak si ho zvolit při vytváření jeho první objednávky. Nebo rovnou vyplnit formulář v části *Odběratel* na listu nové objednávky. Systém při zpracovávání objednávky rozpozná nového odběratele a vloží ho do databáze.

Další částí je část vlevo dole nazvaná *Objednávka*. Prvním řádkem je číslo právě zpracovávané objednávky. Toto číslo je generováno automaticky na základě čísla předchozí objednávky a aktuálního roku. Následující řádky jsou nastaveny na nejpravděpodobnější očekávané hodnoty tak, aby se uživatel v běžných případech nemusel zdržovat jejich nastavováním.

Poslední částí je část *Produkty*. Zde uživatel jednoduše volí kolik čeho a za jakou cenu. Na prvním řádku jsou k výběru všechny dostupné produkty. Tedy s aktuálním stavem zásob větším než nula. Nedostupné produkty systém automaticky vyřazuje z nabídky a v sekci *Produkty* jsou označeny jako nedostupné.

Po potvrzení objednávky tlačítkem *Potvrdit*, začne systém zpracovávat zadaná data. Vytvoří se instance třídy *Objednavka* popřípadě i instance třídy *Odberatel*, naplní se příslušnými daty, vytvoří se potřebné asociace mezi entitami, odečtou se skladové zásoby a všechno se uloží do databáze. Problém nastává při odečítání zásob, kdy hrozí nebezpečí konkurenčního přístupu. Může nastat situace, kdy systém načte aktuální stav zásob daného produktu, změní jeho stav, ale než ho stihne znova uložit, je opět změněn nějakým jiným konkurenčním procesem a data jsou poškozena. Pro takové případy systém používá způsob *optimistického zamykání* [11], který poskytuje knihovna Doctrine 2. Jedná se o způsob, kdy datová entitní třída obsahuje speciální atribut *version* typu *integer*, označený anotací *@ORM\Version.*

```
...
/**
 * @ORM\Version
 * @ORM\Column(type="integer")
 */
private $version
...
```

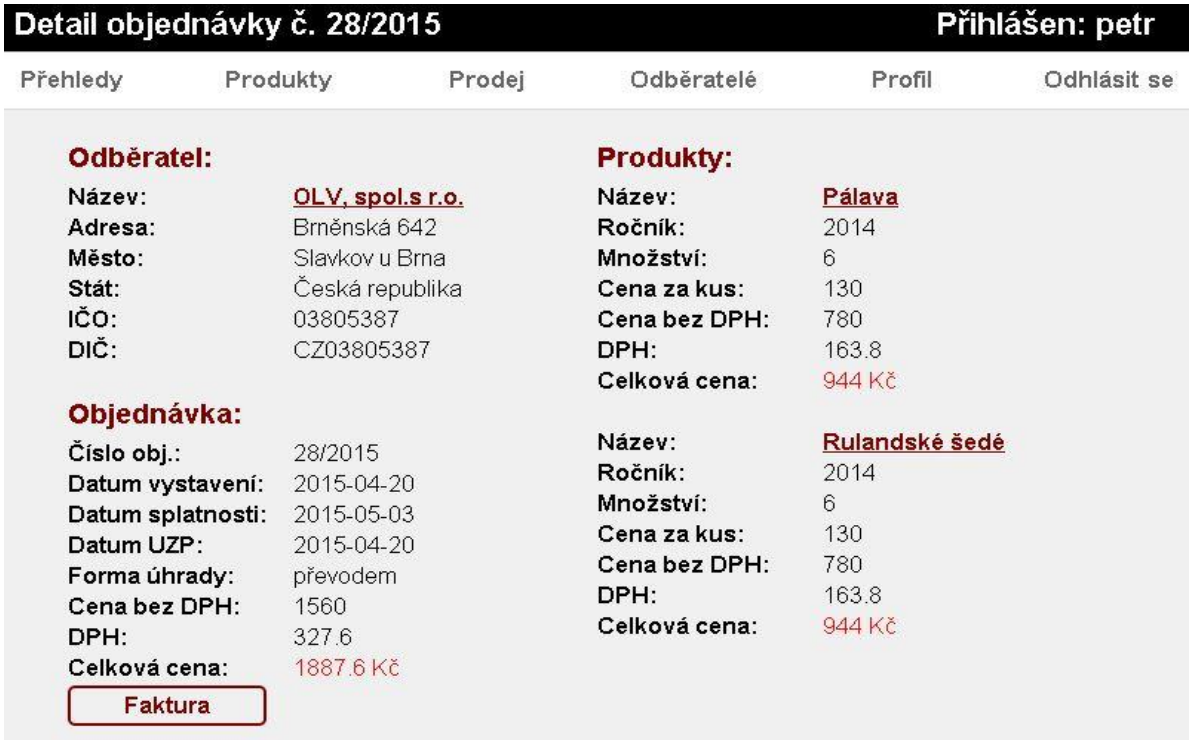
Doctrine 2 si při načtení entity uloží její verzi a při opětovném ukládání kontroluje, zda opravdu přepisuje tu verzi, která byla načtena. V případě přepsání dat konkurenčním procesem, by se hodnoty jednotlivých verzí neshodovaly a byla by vyhozena výjimka *OptimisticLockException* [11].

Následující ukázka *optimistického zamykání* v Doctrine 2 je lehce upravený kus zdrojového kódu informačního systému.

```
$product = $this->DBproducts->findBySarze($sarze);
$version = $product->getVersion();

try {
  //Optimisticke zamykani, kontrola verze
  $this->em->lock($product, LockMode::OPTIMISTIC, $version);

  $product->setAktualStatus($product->getAktualStatus()- $quantity);

  $this->em->flush();
  } catch(OptimisticLockException $e) {
    $form->addError('Byly provedeny změny v produktech, prosím
                    opakujte požadavek.');
    return;
  }
```

Na podobném principu jako nová objednávka jsou zpracovávány i dobropisy a výdejky. Přístupy konkurenčních procesů jsou taktéž kontrolovány pomocí *optimistického zamykání*.

Po zpracování nové objednávky (dobropisu, výdejky) je připravena možnost vygenerování příslušného souboru. U objednávky to bude faktura. Generování probíhá za pomoci knihovny PHPExcel [13]. Generovaný soubor se nijak neuchovává, jen přepíše poslední vygenerovanou fakturu a je nabídnut ke stažení. Fakturu k dané objednávce je možné kdykoliv vytvořit a stáhnout znovu.



Obrázek č. 11 - detail objednávky

5.3.2 Seznamy

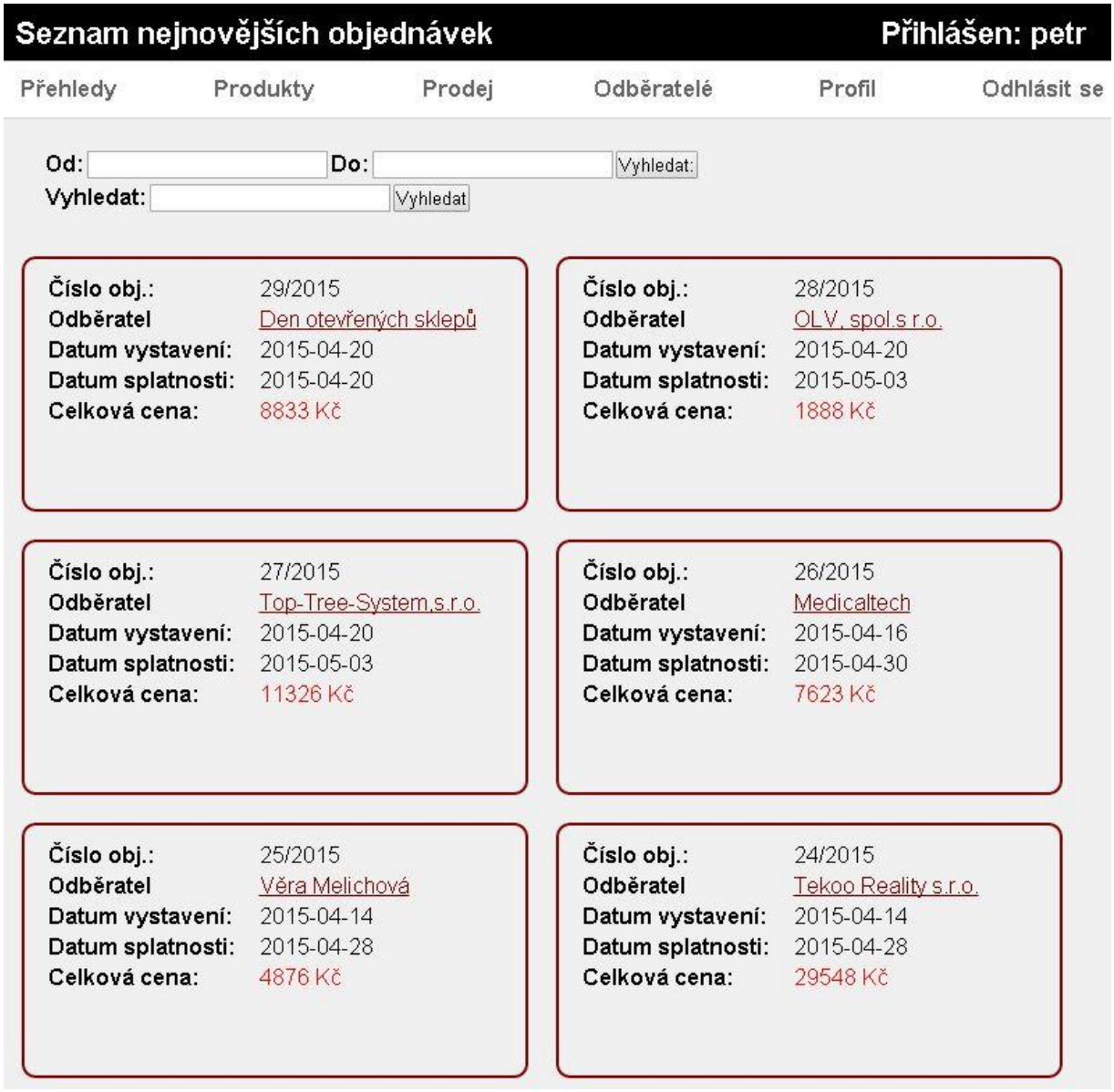
Systém musí uchovávat veškerou možnou historii. Ať už se jedná o historii zpracovaných objednávek, dobropisů, výdejek nebo historii naskladněných produktů a to i těch vyprodaných. K tomuto účelu zde slouží seznamy, pomocí kterých je možné dohledat potřebné informace.

Jedním z takových seznamů je seznam produktů. Na obrázku je vidět, že strana je rozdělena na dvě části. Levý sloupec slouží pro jednoduché filtrování produktů (lahvových vín) podle ročníku. V pravém sloupci se nachází seznam odkazů na detailní informace vybraného ročníku produktů. Červené pozadí znázorňuje nedostupnost produktu.

Obrázek č. 12 - seznam produktů

Dalším takovým seznamem je seznam objednávek, který v základním stavu zobrazuje odkazy na detailní informace deseti nejnovějších objednávek. Je možné vyfiltrovat seznam objednávek za určité období od - do. Dále je možné nechat si zobrazit jen objednávky daného odběratele nebo jednoduše vyhledávat pomocí čísla objednávky.

Další užitečnou funkcí seznamu objednávek je funkce, kdy seznam dokáže graficky odlišit objednávky již zaplacené od nezaplacených. V době psaní této práce není tato funkce ještě plně funkční, otestována a nachází se ve stavu ladění. Hlavní myšlenkou bylo kontrolovat příchozí platby pomocí rozhraní k internetovému bankovnictví a na základě variabilního symbolu je přiřazovat k objednávkám.



Obrázek č. 13 - seznam objednávek

5.3.3 Přehledy a sestavy

5.3.3.1 Inventura k vybranému dni

Jedná se o funkci, která dokáže spočítat inventuru, a to i zpětně k vybranému datu. Ve výchozím stavu zobrazuje inventuru k aktuálnímu datu. Produkty jsou řazeny vzestupně podle ročníku. Je zde možnost vygenerovat soubor ve formátu xls [13] určený k tisku inventury.

Pro výpočet inventury ke staršímu datu, uživatel vybere z kalendáře požadované datum. Systém pak zpět přičte všechny prodané produkty z objednávek, které jsou mladší než je vybrané datum. To stejné platí i pro dobropisy a výdejky.

Objektový přístup k databázovým datům za použití ORM Doctrine 2, zde programátorovi velmi zjednodušuje a zpříjemňuje práci. A to hlavně díky vztahu Objednávka - Produkt, kdy instance entitní třídy Objednávka obsahuje kolekci příslušných objektů třídy Produkt.

Obrázek č. 14 - inventura

5.3.3.2 Přehled prodaných produktů za dané období

Další užitečná funkce, která zobrazuje seznam prodaných produktů, jejich počet a celkovou cenu, za vybrané období. Uživatel jednoduše zvolí hraniční datum a systém pak posbírá zpracované objednávky ve vybraném období, sloučí produkty se stejnou šarží, sečte ceny a všechno přehledně zobrazí.

Obrázek č. 15 - přehled prodaných produktů za dané období

5.4 Instalace aplikace

Požadavky:

- Nette framework vyžaduje PHP verze 5.3.1 nebo vyšší
- Kdyby\Doctrine vyžaduje PHP 5.4 s pdo rozšířením

Instalace:

1. Vytvoření databázových tabulek. V kořenovém adresáři aplikace se nachází soubor *database.sql*.
2. Nastavení přístupu k databázi. V souboru *app/config/config.neon* je potřeba upravit část označenou jako *doctrine:*. Podle příkladu.

```
doctrine:
      driver: pdo_mysql
      host: adresa databáze (localhost)
      user: uživatel
      password: heslo
      dbname: jméno databáze
      charset: utf8
      metadata:
            Entity: %appDir%/model/Entity
```

3. Nahrát aplikaci na webový server.
4. Výchozí přihlašovací údaje: jméno: admin, heslo: 123456. Heslo můžete změnit přímo v aplikaci nebo vytvořit další uživatelský účet

5.5 Zhodnocení použití ORM

Sice mi nějakou dobu trvalo, než jsem prošel důležité části Doctrine 2 dokumentace a naučil se s tím pracovat. Ale později mi to ušetřilo spoustu práce.

V aplikaci na datové vrstvě je hned několik entit, které mají mezi sebou vztah typu N:M. Kdybych pracoval postaru s SQL, musel bych pro uložení takového vztahu, poslat hned několik příkazů do několika různých tabulek. Stejný problém v Doctrine 2 viz. ukázka.

```
$order->addProduct($product);
```

Objednávka - produkt vztah N:M. Pro vývojáře pracujícího s OOP je to mnohem přirozenější přístup, než psát spousty SQL příkazů a myslet při tom ještě na nějakou vazební tabulku. Doctrine 2

poskytuje mnoho dalších věcí, které vývojářům zjednoduší práci a oprostí je od používání SQL příkazů. Osobně vidím ORM jako krok správným směrem.

6 Závěr

Frameworky ORM jsou v dnešní době už poměrně rozšířené do většiny programovacích jazyků. Dál se vyvíjí a vznikají nové verze. Zde byly popsány dva ORM systémy, které patří do skupiny komplexnějších systémů. Doctrine 2 i Propel jsou postaveny na podobném základu, ale každý z nich využívá různých návrhových vzorů.

ORM systémy na správném místě určitě ulehčují a šetří práci. Hlavně při vývoji rozsáhlých a komplexních webových aplikací. Na druhou stranu má takový přístup i nějaké nevýhody. Jako největší nevýhodu bych označil nutnost nastudovat mnohdy velmi rozsáhlou dokumentaci. V případě Doctrine 2 tomu nebylo jinak. Takto komplexní ORM systém, jakým je Doctrine 2 obsahuje velké spousty různých pomocných funkcí a většinou vývojář dokáže využívat jen zlomek z toho všeho. Jakmile však vývojář toto všechno překoná a dokáže se s daným ORM systém trochu sžít, tak bude jeho práce mnohem příjemnější a přirozenější, než dřív, kdy stále dokola vypisoval SQL příkazy. Podle mého názoru, výhody plynoucí z použití kvalitního ORM systému na správném místě, dalece přesahují jeho nevýhody.

Vývoj informačního systému k této bakalářské práci, byla moje první zkušenost s jakýmkoliv ORM systémem. Ze začátku jsem se s tím dost trápil a ne úplně rozuměl základní myšlence ORM. Na konci jsem ale rád, že jsem se naučil něco nového, něco užitečného, něco co funguje a něco co budu určitě ve svých projektech používat dál.

Literatura

[1] GUTMANS, Andi, Stig Saether BAKKEN a Derick RETHANS. Mistrovství v PHP 5. Vyd. 1. Brno: CP Books, 2005, 655 s. ISBN 80-251-0799-x.

[2] FOWLER, M: Patterns of Enterprise Application Architecture. 1. vydání. , Addison Wesley Professional 2002, 560 s. ISBN 0-321-12742-0

[3] FOWLER, Martin. Catalog of Patterns of Enterprise Application Architecture. In: martinfowler.com [online]. 2013-01-01 [cit. 2015-04-25]. Dostupné z: http://martinfowler.com/eaaCatalog/

[4] Doctrine 2. Doctrine 2 ORM's documentation. In: doctrine-orm.readthedocs.org [online] [cit. 2015-04-25]. Dostupné z: http://doctrine-orm.readthedocs.org/en/latest/

[5] TICHÝ, Jan. Seriál: Doctrine 2, In: zdrojak.cz [online]. 2010-07-21 [cit. 2015-04-25]. Dostupné z: http://www.zdrojak.cz/clanky/doctrine-2-uvod-do-systemu/

[6] Propel. Documentation. In: propelorm.org [online]. [cit. 2015-04-25]. Dostupné z: http://propelorm.org/documentation/

[7] Kdyby/Doctrine. Quickstart. In: github.com/Kdyby/Doctrine [online] 2015-2-2 [cit. 2015-04-25]. Dostupné z: https://github.com/Kdyby/Doctrine/blob/master/docs/en/index.md

[8] Framework Nette. Dokumentace. In: doc.nette.org [online] [cit. 2015-04-25]. Dostupné z: http://doc.nette.org/cs/2.3/

[9] Framework Nette. MVC aplikace & presentery. In: doc.nette.org [online] [cit. 2015-04-25]. Dostupné z: http://doc.nette.org/cs/2.3/presenters

[10] ALVAREZ, Peter. Integrate Propel ORM with Zend Framework 2. In: patchworkdev.com [online] 2014-10-15 [cit. 2015-04-25]. Dostupné z: http://patchworkdev.com/blog/integrate-propel-orm-with-zend-framework-2/

[11] Doctrine 2. Transactions and Concurrency. In: doctrine-orm.readthedocs.org [online] [cit. 2015-04-25]. Dostupné z: http://doctrine-orm.readthedocs.org/en/latest/reference/transactions-and-concurrency.html

[12] LEDVINKA, Vít. Kdyby III. - Doctrine - Data Access Object. In: frosty22.cz [online] 2013-02-06 [cit. 2015-04-25]. Dostupné z: http://www.frosty22.cz/kdyby-iii-doctrine-data-access-object

[13] PHPExcel, PHPExcel. In: phpexcel.codeplex.com [online] [cit. 2015-04-25]. Dostupné z: https://phpexcel.codeplex.com/

Seznam příloh

Příloha č. 1 - CD

Příloha č. 2 - fragment zdrojového kódu. Ukázka definice entit, demonstrující použití dědičnosti a asociací s různou kardinalitou.

Obsah CD

- adresář src - zdrojový kód informačního systému
- adresář text - textová práce ve formátu pdf a doc.

Příloha č. 2.

Ukázka dědičnosti a asociace na entitách *OrderE, Product* a *Subscriber*. Entita *OrderE* dědí od *BaseOrderE*. Mezi entitami *OrderE* (objednávka) a *Product* (produkt) je vztah typu M:N. Vztah typu 1:N mají entity *OrderE* a *Subscriber* (odběratel).

```
//Spolecna cast pro entity Objednavka, Dobropis, Vydejka
class BaseOrderE extends \Kdyby\Doctrine\Entities\BaseEntity
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;

    /**
     * @ORM\Column(type="string", unique=true)
     */
    protected $number;

    /**
     * @ORM\Column(type="datetime")
     */
    protected $date;     /* datum vystaveni */

    /**
     * @ORM\Column(type="array")
     */
    protected $products;

    /**
     * @ORM\Column(type="float", precision=2, scale=2)
     */
    protected $basicPrice;
    /**
     * @ORM\Column(type="float", precision=2, scale=2)
     */
    protected $DPH;
```

```
 /**
  * @ORM\Column(type="float", precision=2, scale=2)
  */
 protected $totalPrice;

/**
 * @ORM\Entity
 */
class OrderE extends BaseOrderE
{
    /**
     * @ORM\Column(type="datetime")
     */
    private $dueDate;  /* datum splatnosti */

    /**
     * @ORM\Column(type="datetime")
     */
    private $dateUZP;  /* datum uskutečnění zdanitelného plnění */

    /**
     * @ORM\Column(type="string")
     */
    private $formPayment;

    /**
     * @ORM\ManyToOne(targetEntity="Subscriber",
                      inversedBy="orders")
     * @ORM\JoinColumn(name="subscriber_id",
                       referencedColumnName="id")
     */
    private $subscriber;

    /**
     * @ORM\OneToMany(targetEntity="Creditnote", mappedBy="order")
     */
    private $creditnotes;
```

```
    /**
     * @ORM\ManyToMany(targetEntity="Product",
                       inversedBy="orders")
     */
    private $AProducts;  //A jako asociace


    //Inicializace kolekci v konstruktoru
    public function __construct($values)
    {
        parent::__construct();
        $this->AProducts = new ArrayCollection();
        $this->creditnotes = new ArrayCollection();
    }

    public function addCreditnote(Creditnote $creditnote)
    {
        $this->creditnotes->add($creditnote);
        $creditnote->setOrder($this);
    }

    public function addAProduct(Product $newProduct)
    {
        //Kontrola duplicity produktu
        $add = TRUE;
        foreach ($this->AProducts as $product)
        {
            if ($product->getSarze() == $newProduct->getSarze())
            {
                $add = FALSE;
                break;
            }
        }
        if($add)
        {
            $this->AProducts[] = $newProduct;
        }
    }
```

```
/**
 * @ORM\Entity
 */
class Product extends \Kdyby\Doctrine\Entities\BaseEntity
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    private $id;

    /**
     * @ORM\Version
     * @ORM\Column(type="integer")
     */
    private $version;

    /**
     * @ORM\Column(type="datetime")
     */
    private $date; /* Datum pridani polozky produkt do systemu */

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $shortName;

    /**
     * @ORM\Column(type="string", length=20)
     */
    private $CHOP_CHZO;
```

```
    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $sarze;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $quality;

    /**
     * @ORM\Column(type="integer")
     */
    private $year;

    /**
     * @ORM\Column(type="decimal", precision=2, scale=2)
     */
    private $volume;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $region;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $subRegion;

    /**
     * @ORM\Column(type="float", precision=2, scale=2)
     */
    private $alcohol;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $sugarContent;
```

```
    /**
     * @ORM\Column(type="string", length=80)
     */
    private $evCjakost;

    /**
     * @ORM\Column(type="integer")
     */
    private $defaultStatus;

    /**
     * @ORM\Column(type="integer")
     */
    private $aktualStatus;

    /**
     * @ORM\column(type="boolean")
     */
    private $flagDelete;


    /* Associations*/
    /**
     * @ORM\ManyToMany(targetEntity="OrderE",
                       mappedBy="AProducts")
     */
    private $orders;

    /**
     * @ORM\ManyToMany(targetEntity="Subscriber",
                        mappedBy="products")
     */
    private $subscribers;
```

```
/**
 * Inicializace kolekci v konstruktoru
 */
 public function __construct($values)
 {
     $this->orders = new ArrayCollection();
     $this->subscribers = new ArrayCollection();
 }

 public function addToOrder(OrderE $order)
 {
     $order->addAProduct($this);
     $this->orders[] = $order;
 }

 public function addToSubscriber(Subscriber $subscriber)
 {
     $subscriber->addProduct($this);
     $this->subscribers[] = $subscriber;
 }
```

```
/**
 * @ORM\Entity
 */
class Subscriber extends \Kdyby\Doctrine\Entities\BaseEntity
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=80)
     */
    private $name;

    /**
     * @ORM\column(type="string", length=80)
     */
    private $address;

    /**
     * @ORM\column(type="string", length=50)
     */
    private $city;

    /**
     * @ORM\column(type="integer")
     */
    private $postalCode;

    /**
     * @ORM\column(type="string", length=50)
     */
    private $country;
```

```
    /**
     * @ORM\column(type="string", length=50, nullable=true)
     */
    private $ICO;

    /**
     * @ORM\column(type="string", length=50, nullable=true)
     */
    private $DIC;

    /**
     * @ORM\column(type="boolean")
     */
    private $flagDelete;

    /**
     * @ORM\OneToMany(targetEntity="OrderE",
                      mappedBy="subscriber")
     */
    private $orders;

    /**
     * @ORM\OneToMany(targetEntity="Creditnote",
                      mappedBy="subscriber")
     */
    private $creditnotes;

    /**
     * @ORM\ManyToMany(targetEntity="Product",
                       inversedBy="subscribers")
     */
    private $products;

    /**
     * @ORM\OneToMany(targetEntity="PickList",
                      mappedBy="subscriber")
     */
    private $pickLists;
```

```
    public function __construct($values)
    {
        $this->orders = new ArrayCollection();
        $this->creditnotes = new ArrayCollection();
        $this->products = new ArrayCollection();
    }
    //Settery na inverzni strane asociace
    public function addOrder(OrderE $order)
    {
        $this->orders->add($order);
        $order->setSubscriber($this);
    }

    public function addCreditnote(Creditnote $creditnote)
    {
        $this->creditnotes->add($creditnote);
        $creditnote->setSubscriber($this);
    }

    public function addPickList(PickList $picklist)
    {
        $this->pickList->add($picklist);
        $picklist->setSubscriber($this);
    }

    public function addProduct(Product $newProduct)
    {
        //Kontrola duplicity produktu
        $add = TRUE;
        foreach ($this->products as $product) {
            if ($product->getSarze() == $newProduct->getSarze()){
                $add = FALSE;
                break;
            }
        }
        if($add){
            $this->products[] = $newProduct;
        }
    }
```