Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Automatický import dat do databáze

Bakalářská práce

Autor: Radek Dvořák

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. RNDr. Petra Poulová, Ph. D.

Hradec Králové Srpen 2016

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Pardubicích dne 7.8.2016 Radek Dvořák

Chtěl bych poděkovat vedoucí bakalářské práce, doc. RNDr. Petře Poulové, Ph. D.,

za připomínky při tvorbě bakalářské práce.

Anotace práce

Bakalářská práce stanovuje postup pro automatizovaný import dat do relační databáze. Uvažovaná cílová databáze by měla být relační databází, jejíž schéma je založeno na schématu podle Java Persistence API (JPA). Bakalářská práce analyzuje relevantní části specifikace JPA a problémy, které mohou import komplikovat. Řeší problém cyklických závislostí mezi entitami. Následně je získaný postup ověřen na implementaci prototypu a to s ohledem na paralelní zpracování.

Annotation

Title: Automatic import of data into database

This Bachelor Thesis defines a procedure for automated import of data into a relational database. The target database is supposed to be a relational database with a schema based on Java Persistence API schema (JPA). The Bachelor Thesis analyses relevant parts of JPA specification and difficulties that could complicate the import. Extra effort is spent on cyclic dependencies. This procedure is then verified with a prototype implementation with respect to parallel processing.

Obsah

1. Úvod....1
2. Objektově relační mapování....2
2.1. Java Persistence API....3
2.2. Vztahy mezi entitami....5
2.3. Dědičnost entit....8
2.3.1. Jediná tabulka na celou hierarchii....8
2.3.2. Připojené podtřídy....9
2.3.3. Tabulka pro každou konkrétní třídu....9
3. Analýza....10
3.1. Načtení dat entit....10
3.2. Vztahy mezi entitami....13
3.2.1. Objektový graf....15
3.3. Navrhované řešení....17
3.3.1. Zpracování datových zdrojů....18
3.3.2. Tvorba entit vybrané třídy....18
3.3.3. Správa datového schématu....20
3.3.4. Plánování závislostí....21
4. Implementace....27
4.1. Použitý software....27
4.2. Kompilace entit....29
4.3. Modifikace schématu....29
5. Shrnutí výsledků....33
6. Závěry a doporučení....34
7. Seznam literatury....35
8. Příloha 1: zdrojový kód praktické části....39
9. Příloha 2: vizualizace grafů....40

1. Úvod

V životním cyklu aplikace může dojít k částečné nebo celkové změně schématu databáze. Současně se samotnou změnou je potřeba zajistit přenos informací z původní databáze. Tato bakalářská práce řeší situaci, kdy dojde k celkové změně databázového schématu v relační databázi. V souladu s reálným případem, který zavdal podnět k tomuto tématu, bude cílové databázové schéma odpovídat specifikaci Java Persistence API (Oracle America, Inc. 2013) – dále jen JPA. Použití konkrétního modelu perzistence dat poskytne bakalářské práci pevný omezující rámec a odstraní neurčitost, která by jinak vznikala při snaze vytvořit obecný postup pro libovolné databázové schéma s libovolnou sémantikou.

Na rozsah databázového schématu nejsou v této práci apriori kladeny žádné omezující podmínky. V případě malých změn v praxi přichází v úvahu ruční import dat. Pro uvažovanou neurčitě rozsáhlou databázi však její samotná složitost a rozsah požadovaných změn vytváří riziko chyby. Současně je jedním z hlavních kriterií úspěchu korektnost importu; žádná informace se nesmí ztratit, přibýt nebo změnit význam. V relační databázi tato vlastnost odpovídá referenční integritě, kterou bude proto program pro import dodržovat. V nečekaných situacích, které by ohrozily korektnost dat, se proto navrhovaný program nebude snažit zotavit a ukončí se. Tato vlastnost není na závadu; celková změna databázového schématu spadá do kategorie dlouhodobě plánovaných činností, je možné korektnost postupu předem a opakovaně prověřit při importu nanečisto v testovacím prostředí.

2. Objektově relační mapování

V dnešní době umožňuje objektově orientované programování (dále jen OOP) většina v současnosti populárních programovacích jazyků (TIOBE Software BV 2014). Jednou z běžných architektur je rozdělení zodpovědnosti na logiky aplikační, prezentační a logiku datové vrstvy. Toto uspořádání je též známo jako Model-View-Controller. Vrstvu modelu tvoří dvě části: doménový model a model perzistence dat. Doménový model tvoří objekty, které představují vhodně uspořádané skutečnosti z modelované oblasti. Informace nabývají podobu atributů těchto objektů. Doménový model netvoří jen prostý seznam informací v jednotlivých objektech, nýbrž i chování objektů, vazby mezi objekty a jejich vzájemné uspořádání. Úlohou modelu perzistence dat je mapovat takový doménový model na (relační) databázi. Pro výkon této odpovědnosti musí být tudíž vybaven informacemi jak o doménovém modelu, tak databázi. Přitom toto mapování zůstává relativně[1] transparentní.

Aby bylo možné splnit jeden z hlavních úkolů datové vrstvy, persistenci dat, je nutné objekty, které doménový model tvoří, reprezentovat prostředky vybrané databáze. Tato zodpovědnost připadá modelu perzistence dat. Konkrétně pro persistenci objektového doménového modelu do relační databáze je potřeba mapovat objektový model na relační uspořádání – objektově relační mapování. Toho se dosáhne jednoznačným určením, které strukturální prvky si vzájemně jednoznačně a za jakých podmínek budou odpovídat. Naivní analogii mezi třídami s jejich instancemi a tabulkami se záznamy narušují nekompatibilní vlastnosti obou přístupů. Z teoretického hlediska objektově relačního mapování je hlavní překážkou polymorfismus, kdy atributem může být libovolný potomek třídy deklarované jako

1 Každá netriviální abstrakce „prosakuje“ do vyšších vrstev.

datový typ atributu. Dále jde o problematickou reprezentaci dědičnosti a abstraktnosti tříd, viditelnosti nebo konečnosti atributů a nekompatibilitu datových typů jednotlivých atributů objektů. Ze strany relační databáze objektový model nedostačuje pro pohledy, triggery nebo doménová integritní omezení. Pro složitější SQL dotazy s pod-dotazy, netriviálním připojováním tabulek, agregováním údajů aj. také neexistuje ekvivalent. Mapování tak nutně umožňuje využití pouze podmnožiny možností objektového i relačního modelu. Kde se nabízí více možností mapování, bude nutné zohlednit charakteristiku dat a jejich uvažované použití, aby byla zvolena výkonnostně optimální varianta.

2.1. Java Persistence API

I přes uvedené limity existují ucelené přístupy. Jeden popisuje specifikace Java Persistence API (Oracle America, Inc. 2013). JPA pracuje s návrhovým vzorem „data mapper“. Podle Fowlera (Fowler 2002, s. 165) tak vytváří speciální vrstvu aplikace, která se stará o perzistenci a nutnou objektově relační transformaci. JPA takový objekt označuje jako správce entit, „entity manager“. Ten se řídí konfigurací, která je definována v anotacích tříd doménového modelu a jejich atributů, případně externě v souboru ve formátu XML. Objektům doménového modelu tak není oproti alternativnímu vzoru „Active Mapper“ rozmělňována jejich zodpovědnost a není porušen „Single Responsibility Principle“. JPA takové třídy označuje jako entity.

Entita je jednoduchá třída v Javě (POJO[2]) a v databázi je reprezentována primárně jako samostatná tabulka. Možné je ale entitu rozdělit do více sekundárních tabulek nebo naopak více vložitelných tříd („embeddable class“) persistovat do

2 Zkratka je utvořena z výrazu *Plain Old Java Object*.

hlavní entity. Preferováno je uspořádání jedna ku jedné, neboť sekundární tabulky přinášejí dodatečné výkonnostní nároky i při jednoduchých dotazech.

Entita má sadu atributů, které doplněny o odpovídající anotace:

- identifikují jednoznačně každou instanci třídy,
- obsahují data k uložení do databáze a
- realizují vazby na další entity.

Každá entita musí povinně obsahovat identifikátor, de-facto primární klíč. Vznikne-li hierarchie entit ve smyslu dědičnosti, musí být identifikátor definován právě jednou, a to u kořenové entity. Je přípustný jak jednoduchý, tak i složený identifikátor. Jeho datový typ JPA nepředepisuje, jeho hodnota však musí být pro instanci entity unikátní. Uvedené vlastnosti vyhovují požadavkům na primární klíč tabulky relační databáze.

Atributy entity, nejsou li označeny za přechodny („transient“), jsou automaticky považovány za sloupce. Atributy třídy mají v Javě jako staticky typovém programovacím jazyce povinně deklarovaný datový typ. Není-li výslovně určeno jinak, dokáže implementace JPA automaticky mapovat „*primitivní datové typy, jejich wrapper třídy, java.lang. String, java.math. BigInteger, java.math. BigDecimal, java.util. Date, java.util. Calendar, java.sql. Date, java.sql. Time, java.sql. Timestamp, byte[], Byte[], char[], Character[], enums, a jakýkoliv další typ, který implementuje rozhraní Serializable*“ (Oracle America, Inc. 2013, s. 26) na jejich ekvivalenty z použité relační databáze. Konkrétní mapování na jednotlivé datové typy jazyka SQL však není předepsáno a je tak na implementaci, jaké použije výchozí nastavení. Specifikace JPA pouze říká (Oracle America, Inc. 2013, s. 497), že by mapování v zásadě mělo odpovídat standardnímu mapování použitému pro Java

Database Connectivity (dále jen JDBC). Implementace JDBC API podle jeho specifikace (Oracle America, Inc. 2014, s. 101) nejdříve převede obecný datový typ jazyka Java na typ specifický pro JDBC a ten je následně převeden na typ pro konkrétní relační databázi. Implementace JDBC provádějí i obrácené mapování, jak ukazuje na příkladu databáze MySQL Tabulka 1.

Tabulka 1: Výchozí mapování z databáze do Javy. Upraveno z JPA (Oracle America, Inc. 2013, s. 194–195) *a MySQL Connector/J Developer Guide (Oracle Corporation and/or its affiliates 2014b)*

MySQL	JDBC	Java
CHAR, VARCHAR, TEXT*, ENUM, SET	CHAR, VARCHAR, LONGVARCHAR, NCHAR, NVARCHAR nebo LONGNVARCHAR	String
DECIMAL	NUMERIC	java.math. BigDecimal
BIT (1)	BIT nebo BOOLEAN	boolean
	TINYINT	byte
	SMALLINT	short
SMALLINT, MEDIUMINT (unsigned)	INTEGER	int
BIGINT (unsigned, jinak java.math. BigInteger), MEDIUMINT	BIGINT	long
FLOAT	REAL	float
DOUBLE	DOUBLE	double
BINARY, VARBINARY, BLOB*	BINARY, VARBINARY, nebo LONGVARBINARY	byte[]
DATE	DATE	java.sql. Date
TIME	TIME	java.sql. Time
DATETIME	TIMESTAMP	java.sql. Timestamp

2.2. Vztahy mezi entitami

Atributy entity jsou kromě uchovávání dat použity i k realizaci vazeb mezi ní a dalšími entitami. Z hlediska multiplicity vazeb rozlišuje JPA vazby „OneToOne“,

* Zahrnuje další varianty různé velikosti datového typu.

„OneToMany“, „ManyToOne“ a „ManyToMany“. Multiplicita vazeb mezi entitami je tak obdobná možnostem, které nabízejí relační databáze. Navíc si každá entitní vazba udržuje informaci, která z entit je jejím vlastníkem. „*Vlastnící strana vazby rozhoduje o změnách vazby v databázi*“ (Oracle America, Inc. 2013, s. 43), zejména tedy o odebrání nebo přidání entity na druhé straně vazby. Kromě povinné vlastnící strany může být vazba tzv. oboustranná a mít definovánu i obrácenou (inverzní) stranu. Změny na ní provedené se tak nemusí persistovat. Inverzní strana je v cílové entitě reprezentována atributem s patřičnou významově obrácenou anotací. Z tohoto důvodu jsou vzájemné vazby OneToMany a ManyToOne pouze jiným objektovým pohledem na tentýž cizí klíč[3]. Inverzní vazbou k ManyToMany je opět vazba ManyToMany.

Platí pravidlo, že entita s vazbou ManyToOne je vždy stranou vlastnící a při transformaci do tabulky je na její straně cizí klíč. U vazby OneToOne je cizí klíč součástí tabulky vlastnící strany. Oboustranná vazba OneToOne tak při objektově-relační transformaci ztrácí symetrii a při jejím návrhu je třeba zohlednit strukturu tabulek, které vzniknou. Skutečně symetrickou je vazba ManyToMany, kde je určení vlastnící strany otázkou sémantiky doménového modelu. Speciálním případem je jednostranná vazba OneToMany, která je realizována cizím klíčem v tabulce cílové třídy entit.

Tabulka 2: Vazby mezi entitami. Zdroj: autor.

Název vazby	Vlastnící strana	Realizace vazby
OneToMany - oboustranná	Strana „many“	Cizí klíč na vlastnící straně
OneToMany - jednostranná	Strana „one“	Cizí klíč na inverzní straně
OneToOne	Zvolená strana	Cizí klíč na vlastnící straně
ManyToMany	Zvolená strana	Spojovací tabulka

3 Jednotlivé výrazy jsou používány podle toho, která strana vazby je předmětem vyjádření.

Při transformaci výsledků SQL dotazu na entity implementace JPA potřebuje doplnit objektový graf. Snaží se přitom vyhnout nutnosti vytvořit celý graf – načíst všechny entity z databáze. Neodstraňuje však vrcholy grafu, nahrazuje je „proxy třídami“ (Patricio 2009). Tyto proxy třídy pak teprve na vyžádání („lazy“ metodou) načítá data z databáze. Volba vlastnící strany a oboustrannosti a dalších vlastností vazby tak není otázkou libovůle nebo pouze sémantické vhodnosti v rámci doménového modelu, nýbrž má výkonnostní dopad. Běžná je situace, že schéma stanovuje, že vazba OneToOne nevede povinně na entitu, ale smí nabývat hodnoty NULL. Implementace JPA v této situaci nemůže odkazovanou entitu nahradit proxy třídou. Samotná proxy třída by totiž znamenala, že vlastnost realizující vazbu už nebude mít hodnotu NULL. Jiná situace by nastala, pokud by entita byla povinná. Obdobně entita na opačné straně jednosměrné vazby neobsahuje atribut, jehož hodnota by musela být určena. De-facto ani „neví“, že k ní nějaká vazba vede a vytváření její instance je o to zjednodušeno. Vazby s vícenásobnou multiplicitou využívají podobnou metodu optimalizace, kdy nikoliv jednotlivé entity, nýbrž celá kolekce, je nahrazena proxy třídou.

Vazbám OneToMany a OneToOne specifikuje JPA (Oracle America, Inc. 2013, s. 43) i dvě související vlastnosti: kaskádové mazání a odstranění sirotků. Při smazání vlastnící entity dojde v obou případech ke smazání entity na inverzní straně. Sémantika se liší, když je entita na inverzní straně nahrazena entitou jinou: pouze mazání sirotků ji odstraní. JPA proto tamtéž upozorňuje, že mazání sirotků je tudíž vhodné jen pro případ, kdy je entita na inverzní straně „výlučně vlastněna“, tzn. nevedou k ní další vazby od jiných entit libovolné třídy.

2.3. Dědičnost entit

Mapování entit podle JPA (Oracle America, Inc. 2013, s. 54) umožňuje vytváření hierarchie tříd entit. Třídy entit mohou dědit nejen od dalších entit, ale mohou dědit od tříd, které nejsou entitami. Předek entity, který sám entitou není, může obsahovat stav modelu nebo vazby na další entity. Je označován jako „*mapped superclass*“ a JPA stanovuje, že tato třída tak musí být výslovně označena a její vazby smí být pouze jednostranné (Oracle America, Inc. 2013, s. 55). Může být součástí dotazování, výsledkem je však vždy instance konkrétního potomka.

Relační databáze neobsahují ekvivalent ani optimální reprezentaci konceptu dědičnosti objektů. JPA proto nabízí několik způsobů, jak v nich dědičnost emulovat. Každý má své výhody a jeho použití závisí zvážení konkrétní situace s ohledem na strukturu dat a očekávaný charakter dotazů.

2.3.1. Jediná tabulka na celou hierarchii

Tento způsob emulace dědičnosti spočívá ve využití jedné společné tabulky pro celou hierarchii. Každý řádek obsahuje sloupec, jehož obsah určuje, o jakou entitu konkrétně jde. K získání všech dat instance entity stačí načíst jediný řádek tabulky, čímž nejsou kladeny zvýšené nároky na výkon databáze. Obdobně JPA vyzdvihuje tuto strategii pro dotazy nad celou hierarchií entit (Oracle America, Inc. 2013, s. 58). Záznamy pro atributy, které entita nedědí, jsou v relační databázi nahrazovány hodnotou NULL. Databáze může být pro takové záznamy optimalizovaná, aby nezabíraly žádnou nebo jen minimální paměť[4].

4 Formát řádku v InnoDB engine MySQL databáze obsahuje pro každý sloupec velikost skoku v paměti na sloupec následující. Součástí je podle MySQL Internals Manual (Oracle Corporation and/or its affiliates 2014a) jeden bit, který označuje, že aktuální hodnota je NULL, a tudíž nezabírá žádnou paměť. Formát řádku PostgreSQL obsahuje volitelně bitmapu, která indikuje, který sloupec je NULL (The PostgreSQL Global Development Group 2014, s. 1906). Lze očekávat obdobnou optimalizaci u dalších databází.

2.3.2. Připojené podtřídy

Při mapování hierarchie entit tímto způsobem se vytváří pro každou entitní třídu tabulka se strukturou odpovídající pouze jejím atributům. K získání všech dat instance entity je načten řádek v její tabulce a každé tabulce jejích předků. Práce v hluboké hierarchii entit vyžaduje mnohonásobný JOIN v databázi a zvyšuje nároky na výkonnost (Oracle America, Inc. 2013, s. 58). Pro hluboké hierarchie tento přístup proto není z hlediska výkonu příliš vhodný.

2.3.3. Tabulka pro každou konkrétní třídu

Poslední způsob reprezentace dědičnosti entit je založen na vytvoření samostatné tabulky pro každou konkrétní (tzn. neabstraktní) třídu entit v hierarchii. Tato tabulka obsahuje sloupce pro atributy dané třídy a všech jejích rodičů. JPA uvádí jako nevýhodu potřebu provádět SQL UNION u dotazů prováděných současně nad více entitami hierarchie (Oracle America, Inc. 2013, s. 59). Na druhou stranu jsou využity všechny sloupce tabulky (oproti společné tabulce v kapitole 2.3.1) a není potřeba připojovat žádné další tabulky.

3. Analýza

3.1. Načtení dat entit

Prvním krokem k importu je definice zdrojových dat, ze kterých budou následně podle JPA entity dané třídy vytvářeny. Důležité jsou způsob přístupu k datům, způsob přečtení potřebných informací a jejich vlastnosti. Datové zdroje se pro účely této práce rozdělují na dvě skupiny: soubory a databáze.

Soubory mohou být k dispozici lokálně nebo vzdáleně, prostřednictvím počítačové sítě. V případě počítačové sítě mohou být podle povahy přístupu k nim soubory nejprve zkopírovány, aby s nimi od té doby bylo nakládáno jako se soubory lokálními. Pokud by vzdálené soubory byly získány jednorázově během implementace importu a dále s nimi bylo zacházeno jako s lokálními, nebyla by zajištěna jejich aktuálnost, neboť by byly aktuální jen k okamžiku implementace nebo začátku importu. Vhodnost kopírování závisí na povaze síťového přístupu. Umožňuje-li použitý síťový protokol čtení částí souboru s akceptovatelnou režií[5], jako např. síťový souborový systém NFS (Shepler et al. 2003, sek. 14.2.23) nebo „range request“ v HTTP/1.1 (Lafon et al. 2014), může být s ohledem na systémové zdroje výhodné[6] přenášet po síti pouze části souborů.

I souborové nebo obdobné databáze, jako je například SQLite (ANON. nedatováno), jsou svou podstatou navzdory svému názvu považovány za lokální soubory. Informace z nich jsou získávány pomocí softwarové knihovny, která odpovídá jejich formátu. Ten by proto měl být znám předem při implementaci importu nebo by měl být alespoň určen způsob, jak jej následně určit, a pro všechny pří-

5 V praxi hranice pro rozhodnutí závisí na konkrétním protokolu a vlastnostech síťového přístupu (nejenom propustnost, ale i latence a spolehlivost).

6 Limitujícím faktorem může být mj. kapacita lokálního souborového systému.

pustné varianty by měla být připravena implementace s ekvivalentním výstupem. Všechny formáty však nenabízejí stejné možnosti – např. z rastrového obrázku obecně textové informace běžně načítat možné není, zatímco do CSV (Shafranovich 2005) uložit obrázek možné je (např. kódovaný jako datové schéma URL[7]). Jednotlivé formáty se také liší možnostmi přístupu a dotazování na záznamy. Zatímco formát CSV umožňuje v zásadě pouze čtení řádků, informace z SQLite jsou čteny pomocí dotazů jazyka SQL. Je tak možné v rámci datového zdroje záznamy vybírat podle toho, které záznamy jsou relevantní, a omezovat načtené hodnoty jednoho záznamu.

U databází nemá rozlišení na lokální a vzdálené, jak tomu bylo u souborových datových zdrojů, žádný význam, neboť jsou dostupné prostřednictvím počítačové sítě. Počítačová síť však může být důležitým faktorem pro rychlost importu. Jak uvádí studie IBM (Chao et al. 2005), vyšší síťová odezva a nižší propustnost při práci se záznamy mohou (ve srovnání s přístupem k pevným diskům) představovat úzké hrdlo, zejména pokud by byla databáze přístupná méně kvalitním spojem. S ohledem na náročnost (odvislé od velikosti kolekce dat) a dostupné systémové zdroje by přístup přes loopback nebo jiným lokálním způsobem na témže počítači potenciálně nemusel být výhodný.

> *Aplikace, které vytvářejí značné množství SQL volání, jako například programy běžící v dávkovém režimu, mohou mít prospěch z kolokace s databázovým systémem.* (Chao et al. 2005, s. 8)

U všech datových zdrojů (nejen databáze relační, dokumentová nebo i CSV soubor) je pro implementaci dalšího zpracování potřeba, aby byl k dispozici

7 Viz RFC 2397 (Masinter 1998).

dostatečný dotazovací jazyk nebo API[8], podpůrné knihovny a kandidát na unikátní identifikátor každého záznamu.

Dalším krokem v přípravě informací, ze kterých se vytvoří entita podle JPA, je transformace získaných záznamů. Podle možností datového zdroje mohou být tyto záznamy už předzpracované (např. pomocí SQL), tudíž mohou být ve vyhovující podobě a mít správný obsah, a proto další transformaci nemusí být potřeba provádět. V této fázi dojde také ke sloučení informací z různých datových zdrojů. Sloučení může probíhat nejen podle odpovídajících si identifikátorů, ale i podle obsahu záznamů. I při slučování podle obsahu záznamů musí identifikátor použitých záznamů přečkat transformaci dat nebo být vytvořen nový identifikátor.

Unikátní identifikátory původních záznamů jsou často vhodné k využití jako identifikátory entit. Identifikátor entity by totiž měl být současně primárním klíčem odpovídající tabulky ve vytvářené databázi (Oracle America, Inc. 2013, s. 449). Pro každou třídu entit však není vhodné využívat původní identifikátor zdrojového záznamu. Může tomu tak být proto, že je takový identifikátor nevhodný či nekvalitní nebo by neměl být zveřejňován. Případem nekvalitního identifikátoru může být textová hodnota (je nutné zejména kvůli výkonu zohlednit specifika zvolené relační databáze). Nevhodnými kandidáty na identifikátor entity jsou rovněž nevhodní kandidáti na primární klíč tabulky v relační databázi. Takovým kandidátem je například přirozený primární klíč, který nemá mezi všemi datovými zdroji pro tuto třídu entit zaručenu unikátnost, nebo úředně přidělený identifikátor, který však nebude dostupný u záznamů ze zahraničí. Pokud je kandidát na identifikátor osobním údajem[9] (např. rodné číslo), neměl být použit. Takový

8 Databází přístupnou přes API je například MongoDB (MongoDB, Inc 2015). Typickým jazykem pro přístup k relační databázi je SQL.

9 Viz § 4 písm. a zákona č. 101/2000 Sb., o ochraně osobních údajů, v aktuálním znění (Česko 2000).

údaj totiž může být v rozhraní aplikace veřejně čitelný, např. při adresaci entity v URL. Nabízí se tedy vytvoření syntetického identifikátoru entity, resp. primárního klíče odpovídající tabulky. K tomu se využije vhodná metoda podle zvolené databáze (AUTO_INCREMENT u MySQL (Oracle Corporation and/or its affiliates 2015, s. 238) nebo SERIAL (The PostgreSQL Global Development Group 2014, s. 110), případně SEQUENCE u PostgreSQL atd.). Pro třídu entit může při importu nastat situace, že k již zpracovanému záznamu Z_n má následující záznam Z_{n+1} vztah (chápáno v pojetí významovém, v relační databázi by mohl být realizován cizím klíčem) a současně nové schéma vyžaduje tento vztah korektně přenést – vytvořit ekvivalentní vztah mezi entitou E_{n+1} a již úspěšně importovanou entitou E_n. Taková situace nastane při reprezentaci datových struktur, např. lineárního seznamu nebo stromu. Při prostém vytváření syntetických identifikátorů entit se mapování mezi originálním identifikátorem záznamu a nové entity nezachovává. Řešením této situace je uchování mapování identifikátoru entity na zdrojový záznam v pomocné tabulce v relační databázi. Multiplicita relace mezi pomocnými identifikátory může být 1: 1, avšak bylo-li pro vytvoření jedné entity třeba více záznamů datového zdroje, může pomocná tabulka obsahovat i relaci 1: N nebo M: N.

Výběr datových zdrojů, výběr záznamů v nich a jejich zpracování jsou specifické pro každou třídu entit, která bude do nového schématu importována. Import dat do nového schématu představuje surjektivní zobrazení z množin datových zdrojů na množinu tříd entit.

3.2. Vztahy mezi entitami

Vytvořený systém jedinečných identifikátorů entit a záznamů není vhodný jen k rozlišení jednotlivých entit nebo pro vzájemné odkazování entitami z jedné třídy,

nýbrž i pro identifikaci ve vtazích mezi různými třídami entit. Třídy entit nejsou modelovány v obecných vztazích tříd objektového modelu, nýbrž s vazbami OneToOne, OneToMany a ManyToMany[10]. Proto může být objektový graf reprezentován jako graf *G(V, E)*, ve kterém množinu vrcholů *V* tvoří třídy entit a množinu hran *E* tvoří vazby mezi třídami entit. Na úrovni tříd entit nehraje multiplicita vazby roli, neboť v každém případě hrana spojí právě dvě třídy entit.

Elementární požadavek korektnosti importu vyžaduje, aby byla zajištěna referenční integrita. Ta je v databázi definována cizím klíčem. Z povahy věci musí nejprve existovat klíče vlastní, na které bude ze závislé tabulky odkazováno. Z kapitoly 3.1 Načtení dat entit vyplynulo, že primární klíč odpovídá identifikátoru entity, dokonce se jedná o bijektivní (vzájemně jednoznačné) zobrazení. Vlastnící strana vazby (cizí klíč tabulek) je proto zohledněna v hranách *E*. Hrana je upravena tak, že ji tvoří uspořádaná dvojice, ve které je prvním vrcholem ta třída entit, která je vlastníkem vazby. Graf *G* se tím stává orientovaným grafem.

Objektový graf zachycuje i hraniční stavy vyplývající z vlastností vazeb v JPA nebo samotné grafové reprezentace. Oboustrannost vazby ve smyslu JPA není v grafu tříd entit zohledněna, neboť nevytváří požadavky na korektnost importu. Existuje-li však vazba, která ze závislé entity odkazuje zpět na třídu hlavní entity, tato vazba se zaznamená nezávisle jako samostatná hrana (obousměrné hrany není možné vytvářet).

Zvláštním případem je vazba ManyToMany, která nevytváří v relačním modelu potřebu striktně seřadit importované třídy entit. Cizí klíče ze spojovací tabulky, kterou je na úrovni databáze vazba M:N fakticky realizována, v této úrovni přerušují potenciální cyklickou závislost mezi tabulkami. Porušení objektové abstrakce

10 Viz kapitolu 2.2 Vztahy mezi entitami.

by tak v tomto případě umožnilo vazbu ManyToMany vynechat z počátečního importu a doplnit ji až následně. To by se pozitivně projevilo jako zjednodušení grafu závislostí entit. Na druhou stranu se dá předpokládat, že by takový vícekrokový import zvýšil složitost analýzy vazeb třídy entit i samotného importu. Z hlediska objektového modelu představuje navíc spojovací tabulka implementační detail, který proto do objektového modelu není zahrnut. V této práci je proto pro vazby ManyToMany dodržena stejná konvence jako u ostatních druhů vazeb. Tzn. hrana grafu mezi vrcholy, které reprezentují třídy entit, je orientována od vrcholu, který představuje třídu závislou, k vrcholu, který představuje třídu hlavní.

Dalším hraničním případem je situace, kdy závislá třída entity je na třídě hlavní závislá prostřednictvím více vazeb. Tyto vazby jsou výše uvedeným postupem přeměněny na hrany grafu, které mají stejný počáteční a koncový vrchol. Jsou paralelní neboli násobné. Reprezentace takových hran je v objektovém modelu tříd entit žádoucí, proto se tato práce bude držet obecnější definice grafu a nebude rozlišovat grafy podle přítomnosti paralelních hran na prosté grafy a multigrafy.

Při vytváření objektového grafu je nutné zohledňovat rovněž dědičnost entit. Potomek třídy entity, který nevlastní žádnou vazbu, by jinak mohl být vytvořen jako vrchol bez žádných odchozích hran. Pro reálné zachycení jeho postavení v grafu je však nutné mezi jeho vazby zahrnout i vazby jeho rodičovských tříd. Pro účely zachycení vazeb z těchto rodičovských tříd přichází v úvahu pouze ty třídy, které jsou označeny buď jako entity nebo jako „mapped superclass“.

3.2.1. Objektový graf

Na vytvořeném orientovaném grafu tříd entit je možné definovat řadu vlastností, které mají význam pro porozumění a další zpracování grafu. Od nich se pak bude odvíjet samotné řešení importu entit, resp. ta část importu, která se bude zabývat

zachycením vztahu mezi entitami. Tato kapitola je založena na publikaci Diskrétní matematika (Čada et al. 2004), není-li určeno jinak.

Stupeň vrcholu $d(v)$ je číslo, které je dáno počtem hran, které obsahují vrchol v. U orientovaného grafu je zvlášť rozlišován stupeň vstupní a výstupní. Vstupní stupeň $d^+(v)$ je dán počtem hran, které obsahují vrchol v *a* jsou směřovány do vrcholu v. Výstupní stupeň $d^-(v)$ je určen počtem hran, které obsahují vrchol v a jsou směřovány z vrcholu v.

Sledem je posloupnost vzájemně incidentních vrcholů a hran. Cestou je takový sled, ve kterém se neopakují ani vrcholy ani hrany. Sled i cesta mohou být označeny jako uzavřené. Uzavřený sled nebo cesta je takový sled, resp. cesta, že jeho počáteční vrchol je zároveň vrcholem konečným. Pravidlo pro neopakování vrcholů v cestě se v tomto případě na tento počáteční vrchol a jen za účelem uzavření cesty neaplikuje. Orientovaná cesta je taková cesta, která respektuje orientaci incidentních hran. Uzavřená cesta je kružnicí, uzavřená orientovaná cesta se nazývá cyklus.

Jako souvislý je označován takový graf G, pro jehož každé dva vrcholy u, v existuje alespoň jedna cesta z vrcholu u do v. Orientovaný graf je souvislý, existuje-li mezi každou dvojicí vrcholů u, v neorientovaná cesta.

Acyklický graf je takový orientovaný graf, který neobsahuje žádný cyklus. Nutno poznamenat, že triviální cesta, tj. taková cesta, která obsahuje pouze jeden vrchol a žádnou hranu, není považována za cyklus, byť je její počáteční vrchol zároveň vrcholem konečným. Acyklický graf nemusí být souvislý.

Komponentou grafu je jeho maximální souvislý podgraf. Platí, že graf je souvislý právě tehdy, když obsahuje jednu komponentu.

Slabě souvislý orientovaný graf je silně souvislý, právě když každá jeho hrana je obsažena v nějakém cyklu (Čada et al. 2004, s. 93)

Silně souvislý graf je takový souvislý orientovaný graf, kde pro každý vrchol u, v existuje orientovaná cesta z vrcholu u do v a zároveň existuje orientovaná cesta z vrcholu v do u. Analogickou aplikací této definice na komponentu je definována silně souvislá komponenta jako maximální silně souvislý podgraf. S využitím triviální cesty je vrchol sám o sobě silně souvislou komponentou.

Kondenzací grafu G je vytvoření takového grafu G', jehož vrcholy jsou silně souvislé komponenty. Existuje-li v G hrana mezi silně souvislými komponentami, je tato hrana použita i v G'. Výsledný graf G' je vždy acyklický graf.

Topologické řazení vrcholů acyklického grafu G je lineární uspořádání jeho vrcholů tak, že pro každou hranu *(u, v)* platí, že vrchol u předchází v řazení vrchol v (Cormen 2001, s. 549). Tamtéž uvádí, že topologické řazení může být zobrazeno tak, že jsou všechny vrcholy grafu seřazeny podél horizontální přímky a všechny hrany jsou orientovány jedním směrem.

3.3. Navrhované řešení

Teoretické informace o objektově relačním mapování podle specifikace JPA, analýza vlastností tříd entit, informace o vztazích mezi třídami entit a jejich abstrakce jako objektový graf tvoří základní skutečnosti, které determinují tvorbu postupu pro import dat do databáze. Z jednotlivých dílčích částí předchozí analýzy vychází základní části navrhované aplikace:

- Práce s datovými zdroji:
  - Zpracování datových zdrojů
  - Tvorba entit vybrané třídy
- Správa datového schématu
- Plánování závislostí
- Aplikační část, která spojuje předchozí dílčí části a spouští import

Základní části tvoří jádro aplikace, které přímo zajišťuje hlavní případ užití – import dat do databáze. Kromě částí nezbytných pro hlavní případ užití může být aplikace dále rozšiřována o vedlejší části, např. uživatelské rozhraní, testy algoritmů (unit testy) nebo dokumentaci. Vedlejší části však nebudou s ohledem na rozsah v této práci implementovány ani navrženy.

3.3.1. Zpracování datových zdrojů

Tato část aplikace umožňuje čtení jednotlivých záznamů z datových zdrojů. Datový zdroj je však jen obecné označení pro jednotlivá technologická řešení, která poskytují kolekci záznamů k importu. Kolekce záznamů je objekt, který je možné iterovat pro jednotlivé záznamy seřazené v takovém pořadí, v jakém mají být importovány do databáze. Každý záznam poskytuje všechny hodnoty entity, její unikátní identifikátor i informace o jejích vazbách na jiné entity.

3.3.2. Tvorba entit vybrané třídy

Každá třída entit má jiné vlastnosti a jiné datové zdroje, tudíž vytváření entit nemůže být prováděno jedním postupem. V úvahu přichází dva způsoby vytvoření entity a její naplnění daty: explicitním naprogramováním a konfigurací[11]. Algorit-

11 Konfigurací je myšlena parametrizace obecného postupu. Může k tomu docházet v kódu nebo i v konfiguračním souboru.

mus na míru se může přesně přizpůsobit specifickým požadavkům na vytvoření entity, je však potřeba ho pro každou třídu pracně vytvořit, a proto je málo znovupoužitelný. Obecný konfigurační postup sice zvyšuje znovupoužitelnost kódu, vyžaduje však značné úsilí k vytvoření základního „frameworku“. Záleží zejména na povaze a počtu tříd entit a příslušných zdrojů dat, kterou strategii v konkrétním případě využít. Obecně se dá předpokládat, že výhodnost přístupů bude záviset na množství importovaných tříd entit. S rostoucím množstvím entit se může vyplatit konfigurační přístup, kdy vysokou vstupní investici do vývoje nahradí následně nižší variabilní náklady pro každou třídu entit. Z hlediska složitosti vnitřní logiky třídy entit je však možné upřednostnit jiné kritérium: na jednodušší „číselníkové“ třídy entit využít konfigurační metodu a pro složité třídy entit (např. ty se složitější business logikou) postup explicitně naprogramovat.

V tomto kroku musí být entitě nastaven unikátní identifikátor (viz kapitolu 3.1). Použití přirozeného identifikátoru nebo vygenerování syntetického (včetně mapování původního identifikátoru na nový syntetický) by mělo být součástí obecné konfigurace nebo explicitního algoritmu pro vytvoření entity.

Data, která jsou nastavována vytvářené entitě, obsahují i vazby na jiné třídy entit. Na jinou než aktuálně zpracovávanou třídu je odkazováno jejím identifikátorem. Jeho hodnotu lze vyčíst buď ze zpracovávaných dat záznamu (ať už přímo nebo po transformaci) v případě, že odkazovaná entita je identifikována přirozeným identifikátorem. Je-li však odkazovaný identifikátor vytvářen uměle, může být získán pouze z „mapy“, pokud byla apriori pro odkazovanou třídu entit vytvořena. Taková mapa by měla obsahovat pro každý záznam identifikátor původní entity a identifikátor entity nové. Při implementaci může být použita například tabulka v relační databázi, která by na každém řádku obsahovala oba identifikátory. Každý algoritmus pro tvorbu entit proto musí mít informace o re-

levantních mapách identifikátorů entit (resp. o pomocných databázových tabulkách) a jimi zachycené multiplicitě. Jsou nezbytné pro implementaci importního algoritmu pro entity v každém případě, kdy k sobě mají vztah záznamy s generovanými identifikátory.

3.3.3. Správa datového schématu

Aplikace využívá tuto součást k:

- vytvoření databázové struktury nového datového schématu,
- analýze JPA schématu pro potřeby plánování a spouštění importu a
- úpravám schématu, není-li jinak korektní import možný.

Před samotným importem je nutné, aby cílová relační databáze byla připravena a pomocí DDL dotazů byly vytvořeny všechny tabulky, indexy atd. Tyto DDL dotazy by měla vygenerovat zvolená implementace JPA. Takto vytvořené tabulky by neměly obsahovat žádná data z (případných) předchozích importů (např. testovacích).

Je-li uvažováno o kompatibilitě s více implementacemi JPA, pak by jednotlivé implementace měly být abstrahovány v situaci, kdy jsou vyžadována metadata o struktuře schématu. Vystaveny by měly být informace o názvu třídy entit, jejím umístění v hierarchii (informace dědičnost) a vazbách na jiné třídy entit. Pro pomocné kalkulace by mělo být z datových zdrojů dostupné předpokládané množství entit třídy (např. pro ověření multiplicity vazeb nebo výpočet zbývající doby importu entit dané třídy).

Při analýze závislostí tříd může nastat eventualita, kdy vzájemné závislosti zamezí korektnímu importu dat. Z logického „zákona vyloučení třetího“ vyplývá, že když není možný korektní import dat v konzistentním stavu, bude nutné impor-

tovat data v nekonzistentním stavu. Dá se oprávněně předpokládat, že jednorázový nekonzistentní import neumožní JPA ani relační databáze. Pokud na import nebude pohlíženo jako na atomickou operaci, ale na posloupnost kroků, může být za běhu aplikace měněno datové schéma tak, aby po posledním kroku získalo cílovou podobu, byť jednorázově by takový import nebyl možný. Technicky z hlediska JPA i databáze bude tedy každá část importu korektní. Integrita importovaných dat podle původního schématu bude po prvotním importu zajištěna v dalších navazujících krocích opravou JPA vazeb, databázového schématu i samotných záznamů.

3.3.4. Plánování závislostí

Pro korektní realizaci vazeb mezi entitami je nutné určit pořadí, v jakém budou jednotlivé třídy entit importovány. V triviálním případě vytvářené entity, resp. jejich třídy, nevlastní žádné vazby – závislosti. Takové třídy entit je možné importovat v libovolném pořadí. Existují-li však mezi třídami entit vazby, správné pořadí vznikne seřazením všech tříd entit tak, aby všechny závislosti byly splnitelné, resp. entity odkazované třídy byly již importovány. Požadovaného efektu seřazení je možné dosáhnout topologickým řazením vrcholů objektového grafu.

Vhodný algoritmus pro topologické řazení ukazuje Alg. 1. Vnější cyklus prochází všechny vrcholy grafu, čímž zaručí, že budou nakonec navštíveny všechny vrcholy, byť by nebyly dosažitelné ve směru (orientovaných!) hran grafu. Návštěva vrcholu využívá prohledávání grafu do hloubky pro nalezení vrcholu bez výstupní hrany. Průchod grafem je pro přehlednost zapsán rekurzivně, je však ho možné převést do iterativní formy kvůli potenciálnímu přetečení zásobníku při hluboké rekurzi.

```
Vytvoř seznam pro uspořádané vrcholy.
Pro každý vrchol:
     když nemá příznak návštěvy:
          navštiv vrchol.
Návštěva vrcholu:
     Označ vrchol za navštíveny.
     Pro každou odchozí hranu:
          Když cílový vrchol hrany není navštívený:
               Navštiv cílový vrchol.
     Přidej vrchol na začátek seznamu
```

Alg. 1: Topologické řazení. Zdroj: autor.

Topologické řazení poskytuje správný výsledek pro graf, který neobsahuje žádný cyklus. Tuto podmínku je možné ověřit algoritmem pro detekci silně souvislých komponent grafu. Vhodným je např. algoritmus od Roberta Tarjana (Tarjan 1972), protože k jeho ukončení stačí jeden průchod grafem a je implementačně jednoduchý. Pro obecné zhodnocení pořadí importu postačí takto nalezené silně souvislé komponenty kondenzovat. Výsledné vrcholy grafu by pak už bylo možné topologicky seřadit. Řešení vlastního problému silně souvislých komponent kondenzace silně souvislých komponent však jen odkládá.

Aby mohlo být stanoveno pořadí vrcholů i v silně souvislé komponentě, je nezbytné narušit její silnou souvislost. Narušení silné souvislosti je z definice silné souvislosti taková úprava tohoto podgrafu, po které existuje alespoň jeden vrchol, pro který přestane existovat cesta do každého jiného vrcholu nebo přestane existovat cesta z každého jiného vrcholu do něj. Podgraf bude upraven tak, že k vrcholu v vznikne jeden nový vrchol v' takový, že definiční obor vrcholu v' je shodný s definičním oborem vrcholu v. Za definiční obor je v tomto kontextu považována množina identifikátorů entit příslušné třídy. Hrana, která má být v grafu zrušena a směřovala do vrcholu v, je upravena tak, aby směřovala do vrcholu v'. Vrcholu v' ve smyslu tříd entit odpovídá nová třída, která obsahuje pouze iden-

tifikátor stejného typu jako v třídě vrcholu v. Ukázkou takové operace jsou grafy na Obr. 2 a Obr. 3 na straně 40.

Samotný mechanismus narušení silné souvislosti nestačí, neboť je nutné stanovit kritéria, podle kterých budou porovnány hrany k odstranění. Základním kritériem je bezesporu požadavek na porušení silné souvislosti komponenty. Při zohlednění vícenásobně hranově souvislých komponent tomuto odpovídá snížení hranové souvislosti komponenty o jedna. Při porovnávání dvou a více hran, které obě splní podmínku porušení souvislosti, přichází na řadu ekonomické kritérium, porovnat vzájemně náročnost možností. Ta může být odvozována z náročnosti jednotlivých kroků, které jsou potřeba pro vytvoření pomocného vrcholu:

1. vytvoření třídy entit v objektovém a tabulky v relačním schématu,
2. změna vazeb z originální na pomocnou třídu entit, resp. tabulku,
3. načtení identifikátorů entit z vrcholu v do entit vrcholu v' a
4. po importu oprava vazeb a odstranění pomocné tabulky.

Zevrubné určení náročnosti jednotlivých kroků je nad rámec této práce. Proto se dále uvažuje zjednodušený model, který vychází z analogie k asymptotické složitosti algoritmů. Jako hlavní kritérium je v každém kroku uvažováno množství záznamů, které jsou v jednotlivých alternativách ovlivněny. První dva kroky jsou tvořeny „jen“ operací v rámci JPA a databáze, proto jsou (podle zvoleného kritéria) zanedbatelné. Algoritmy obou zbývajících kroků nejsou apriori určené, neboť závisejí dílem i na databázovém systému a datovém schématu. Přesto srovnání počtu ovlivněných záznamů dává alespoň vodítko. Podle způsobu realizace vazby[12] může být za srovnávací kritérium (každých) dvou hran považováno množství záznamů třídy entit odpovídající cílovému vrcholu nebo naopak množství záznamů

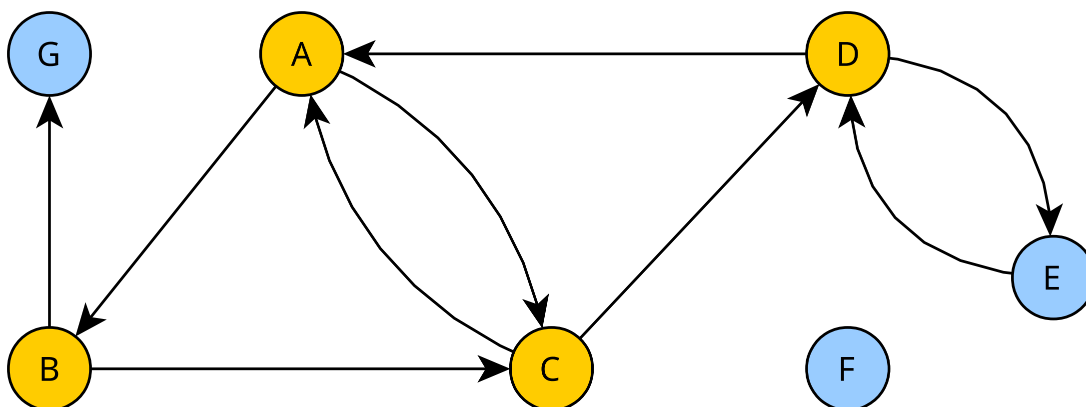
12 Tato variabilita vychází ze strany, která realizuje v databázi vazbu, jak ukazuje Tabulka 2.

třídy entit odpovídající zdrojovému vrcholu. Dalším vhodným řešením volby optimálního kritéria může být praktické změření. Bez ohledu na to, jaké kritérium bude zvoleno, může o srovnávaných hranách být uvažováno jako o hranách ohodnocených. Váha má charakter ceny za narušení silné souvislosti podgrafu v této hraně.

Silně souvislá komponenta může obsahovat další silně souvislé podgrafy, které nejsou shodné s touto komponentou. V komponentě jsou pro další analýzu hledány podgrafy, které jsou silně souvislé a zároveň netriviální. Možný postup je takový, že z každého[13] vrcholu silně souvislé komponenty je zahájeno její prohledání do hloubky. Větev stromu nalezených vrcholů, který je při prohledávání do hloubky vytvářen, je zaznamenána vždy poté, když prohledávání navštíví kořen stromu. Vrchol větve/listu prohledávacího stromu tudíž ukazuje na rodičovský vrchol, od kterého k němu vede orientovaná cesta silně souvislého podgrafu. Tímto jsou všechny vrcholy silně souvislé komponenty zařazeny do některého silně souvislého (pod)grafu.

Popsaný algoritmus je jednoduchý na popis i implementaci, avšak výkonnostně neoptimální. Nijak například nevyužívá případu, kdy algoritmus prohledávání do hloubky narazí na jiný než kořenový vrchol ze současné nebo větve.

13 Z libovolného vrcholu silně souvislé komponenty vede cesta do každého vrcholu této komponenty.



Obr. 1: Silně souvislá komponenta. Zdroj: autor.

Například Obr. 1 znázorňuje graf se silně souvislou komponentou *ABCD*. Vrchol *F* není spojen s komponentou žádnou hranou a tvoří samostatnou komponentu. Vrchol *G* je sice spojen s komponentou *ABCD* hranou *(B, G)*, chybí však hrana z vrcholu *G*, která je podmínkou silné souvislosti grafu. Proto není součástí silně souvislé komponenty. Vrchol *E* vytváří samostatnou silně souvislou komponentu *DE*. Po vyhledání dílčích silně souvislých komponent jsou pro účely této práce v grafu identifikovány silně souvislé komponenty *ABCD*, *DE*. Komponenta *ABCD* obsahuje silně souvislé podkomponenty *ABC*, *ACD* a *AC*.

Ze seznamu silně souvislých (pod)komponent již je možné určit, které hrany odstranit. V každé takové silně souvislé komponentě jsou hranám inkrementovány čítače použití. Rovněž je každá hrana (jednou) ohodnocena hodnotou podle stanoveného srovnávacího kritéria. Protože je optimalizačním cílem nízký počet přerušených hran s minimální celkovou cenou, jsou zvýhodněny ty hrany, které sdílí více silně souvislých (pod)komponent. Uvažované náklady („cena“) přerušení vícenásobně sdílených hran se snižují tak, že ohodnocení hrany se vydělí hodnotou čítače počtu jejího použití. Obdobným způsobem by měly být penalizovány paralelní hrany. Pro nalezení hrany silně souvislého podgrafu, která je vhodná k přerušení, pak stačí hrany seřadit podle výsledných nákladů a nalézt hranu

s nejnižší hodnotou. Další vhodnou hranu k přerušení je možné získat iterací výsledného seznamu ohodnocených hran. Navíc se uplatní podmínka, že musí být u hrany zkoumáno, jestli je součástí alespoň jedné nepřerušené silně souvislé (pod)komponenty. Algoritmus končí v okamžiku, kdy je v grafu narušena silná souvislost všech nalezených (pod)komponent.

Velmi důležitou vlastností při hledání hrany k přerušení je její orientace. Nesmí být zaměněna s případnou hranou opačné orientace. Např. přerušením hrany *(A, C)* v Obr. 1 ztrácí silnou souvislost podgraf *ACD*. Podgraf *ABC* však obsahuje hranu opačné orientace *(C, A)*, proto není dotčen. Silně souvislá komponenta *ABCD* sice hranu *(A, C)* obsahuje, přesto stále zůstává silně souvislá díky cestě *(A, B)*, *(B, C)*, *(C, D)* a *(D, A)*.

4. Implementace

Cílem implementační části je vytvořit „proof of concept", potvrdit implementací realizovatelnost předchozích částí. Vzhledem k rozsahu a hloubce tématu je implementována jen podmnožina možností jak definovat v JPA schéma, jako příklad jsou použity pouze lokální datové zdroje a některé algoritmy jsou zjednodušeny.

Projekt je strukturován obdobně k „standardnímu rozložení adresářů" (The Apache Software Foundation 2002b) projektu Maven. Z důvodu rekompilace a dynamického načítání jsou samostatně umístěny třídy entit, jejich továrny a datové zdroje a související třídy.

Další podkapitoly popisují vybrané otázky řešené při implementaci.

4.1. Použitý software

Zadání práce vyžaduje použití standardu JPA. Nejjednodušeji použitelným jazykem implementace v praktické části této práce je Java, ačkoliv by bylo teoreticky možné použít např. jazyk Scala.

> *Scala běží na JVM, proto mohou být knihovny jazyků Java a Scala volně míchány a úplně bezproblémově integrovány.*(École Polytechnique Fédérale de Lausanne 2002)

Kvůli předchozí autorově znalosti byla implementace provedena v jazyce Java. Byla použita aktuálně nejnovější, osmá, verze.

Moderní správu použitých knihoven třetích stran zajišťuje Apache Maven (The Apache Software Foundation 2002a). Používá se k tomu soubor pom.xml umístěný v kořenovém adresáři projektu. Pomocí jednoduchého zápisu ve formátu XML je možné v tomto souboru deklarovat závislosti na externích knihovnách a specifi-

kovat jejich požadovanou verzi. Maven dokáže ověřit existenci těchto závislostí, rekurzivně vyhledá jejich další závislosti a všechny je nainstaluje do adresáře *$ {user.home}/.m2/repository*. Odpadá tak potřeba distribuovat s projektem další knihovny v podobě archivů .jar.

Jako implementace JPA byl zvolen Hibernate ORM (ANON. 2016). Důvodem volby je kvalitní dokumentace, aktivní vývoj a alespoň základní předchozí autorova zkušenost. Početná komunita uživatelů rovněž vytváří prostředí pro snadné řešení problémů.

Kapitola 3.3.4 Plánování závislostí požaduje k řešení cyklů v grafu závislostí generování a úpravy zdrojového .java souboru pro třídy entit. To je prováděno pomocí knihovny Roaster (Gastaldi et al. nedatováno). Byla vybrána kvůli úplnosti implementace s důrazem na úpravy anotací, které jsou používány jako metadata definující strukturu objektového modelu. Důležitými faktory byly také aktivní vývoj, kvalitní dokumentace a subjektivně intuitivní API. Načtení vytvořených nebo upravených tříd entit zajišťuje JVM agent Spring Loaded (Pivotal Software nedatováno). Důvody tohoto řešení jsou podrobněji zpracovány v kapitole 4.3 Modifikace schématu.

Organizaci a konfiguraci jednotlivých tříd aplikace napomáhá ve smyslu programovacího principu „dependency inversion“ spring-context ze Spring Framework (Pivotal Software 2016). Vybrané služby, označované jako „Bean“, jsou definovány v XML souboru context.xml. Zde jsou nejen službám určeny použité třídy, ale nastaveny i parametry jejich konstruktoru – ať už jimi jsou další služby nebo data. Konfiguraci Hibernate ORM navíc usnadňuje nadstavba jménem spring-orm (která je rovněž součástí výše uvedeného projektu Spring). Konfigu-

rační údaje jsou načítány z .yaml souboru pomocí knihovny SnakeYAML (Somov 2015).

Projekt k reprezentaci vztahů tříd entit využívá grafů. Ani pro toto nebyla autorem vytvářena vlastní implementace, a to jak kvůli složitosti, tak i rozsáhlosti a z toho vyplývající časové náročnosti. K reprezentaci grafů a pro některé grafové algoritmy je proto využita knihovna JGraphT (Naveh et al. 2003). K dalším běžným činnostem aplikace patří alespoň jednoduché logování. V tomto projektu je realizováno v minimální variantě pomocí Simple Logging Facade for Java (SLF4J) (QOS.ch Sàrl nedatováno). Získávání datových zdrojů z CSV souboru usnadňuje parser z knihovny uniVocity-parsers (uniVocity Software Pty Ltd nedatováno). Pro vizualizaci grafu tříd entit byl použit jazyk DOT a program dot z projektu Graphviz (AT&T Labs Research nedatováno) pro jeho zobrazení[14].

Aplikace využívá jako relační databázi MySQL. Tato databáze byla vybrána díky předchozím zkušenostem autora bez dalšího srovnávání.

4.2. Kompilace entit

Zdrojové kódy entit jsou kvůli rekompilaci udržovány stranou hlavní aplikace. Umístění v samostatném jmenném prostoru zjednodušuje hlídání závislostí kompilace. Kompilace totiž vyžaduje, aby byly současně kompilovány všechny třídy, které jsou v kódu uvedeny. Není tak možné rekompilovat jednu entitu bez všech entit, na které odkazuje. Implementace získá ze spring contextu cestu k entitám, prohledá ji, zaeviduje nalezené třídy, zkompiluje je a následně doplní class loader.

4.3. Modifikace schématu

Teoretické úvahy kapitoly 3.3.4 implicitně uvažují o úpravách grafu, jehož vrcholy tvoří třídy entit. Zatímco v grafu je vytváření vrcholů, rušení a vytváření hran

14 Viz grafy na Obr. 2 a po úpravě na Obr. 3 v příloze č. 1.

typickou operací, je pouze předpokládáno, že ekvivalentní operace jsou možné na všech úrovních abstrakce. Zatímco na nejnižší úrovni, v relační databázi, jsou úpravy schématu přímo součástí SQL, mezistupeň abstrakce v podobě JPA dynamickou úpravu schématu neobsahuje[15]. Při implementaci byly uvažovány následující varianty řešení:

a) úprava reprezentace v Hibernate ORM a

b) úprava zdrojového kódu tříd entit.

Dočasná úprava interní reprezentace entit přímo v Hibernate ORM s následnou synchronizací do relační databáze by představovala elegantní řešení dočasného problému závislostí při importu. Taková implementace by mohla být odvozena ze série článků o tom, jak „programově sestavit konfigurační metadata pro Hibernate“ (Stalla 2012). Při experimentálním ověřování se však ukázalo, že kromě vysoké složitosti mapování atributů tříd entit na sloupce tabulek je problémem zejména ta skutečnost, že použitá API jsou obsažena pouze ve starším Hibernate ORM verze 4.x. Protože se nepodařilo získat žádné informace o podobném řešení pro aktuální verzi Hibernate ORM bylo od této varianty upuštěno.

Druhou možností je úprava zdrojového kódu. To znamená, že soubory .java s třídami entit by měly být přepsány a znovu použity. Po přepsání jsou však .java soubory zkompilovány do bytecode – class souborů, se kterým JVM dále pracuje. Ačkoliv je možné programově pomocí `javax.tools. JavaCompiler` zkompilovat .java soubory do bytecode, spuštěné procesy JVM tuto změnu automaticky nezohledňují a pracují proto nadále s původní podobou tříd, v tomto případě entit. Řešení tohoto problému poskytuje JVM agent Spring Loaded, který se v samostatném vlákně stará o periodickou kontrolu bytecode a upravené části za běhu vymění. Protože

15 Výjimkou je generování počátečního schématu (Oracle America, Inc. 2013, kap. 9.4 Schema Generation).

Spring Loaded používá vlastní vlákno, probíhá kompilace v jiném vlákně než výměna bytecode. Kvůli zajištění jednorázovosti změny tříd entit musí být činnosti těchto vláken synchronizovány. K tomu je využívána nadstavba nad `java.util.concurrent.CountDownLatch`, která čeká na hot-swap konkrétních jmenovaných tříd. Údaje o nahrazených třídách poskytuje agent Spring Loaded registrovaným implementacím `org.springsource.loaded.ReloadEventProcessorPlugin`. Konzistence datového modelu je zajišťována tak, že používaný entity manager je při změně entit uzavřen a při vytvoření nového dojde automaticky k aktualizaci databázového schématu. Konkrétně nastavení `hibernate.hbm2ddl.auto` na hodnotu *auto* způsobí, že se spustí `org.hibernate.tool.hbm2ddl.SchemaUpdate`. Kvůli chybě (nebo nedostatečné implementaci) ve volané metodě applyForeignKeys třídy `org.hibernate.tool.schema.internal.SchemaMigratorImpl` však nedojde k úplné úpravě cizích klíčů. Nové neexistující cizí klíče jsou přidávány, avšak existující cizí klíče odstraňovány nejsou, zůstávají v databázi. To je způsobeno tím, že výchozí název klíče je odvozován od vlastností vazby. A proto změní-li se vazba, výchozí název cizího klíče se rovněž změní. Nepomůže však ani explicitní nastavení anotací `@JoinColumn` s `@ForeignKey`. V tom případě sice nedojde k duplikaci cizích klíčů, avšak klíč zůstane v původním stavu a změna schématu není nijak reflektována. Jméno je dané explicitně, a proto Hibernate neobjeví podle atributů změněné jméno pro cizí klíč. Vložení dat do takové tabulky vyvolá v relační databázi chybu, neboť je kvůli nesmazanému cizímu klíči porušena referenční integrita. Oprava tohoto chování vyžaduje úplnou výměnu implementace pro aktualizaci schématu nebo doplnění kódu projektu Hibernate. Za změnu schématu zodpovědná instance třídy není konfiguračně dostupná, aby mohla být nahrazena např. svou ad-hoc upravenou kopií. Z těchto důvodů v prototypovém projektu není možné bezchybně při změně symé-

matu dokončit import dat, pokud je hodnota asociace vyžadována (tj. nesmí mít hodnotu `null`).

Problematické, byť řešitelné, je samotné odkazování na kopírované třídy entit při jejich plnění daty. Typová kontrola v jazyce Java zamezuje vložení instance libovolné třídy. Protože je cílová třída změnou zdrojových kódů změněna, musí se importér při plnění asociací dynamicky dotazovat na název cílové třídy, zjistit typ atributu a vkládané hodnotě dynamicky změnit typ („type cast“). Uložena pak může být pomocí reflexe, jak ukazuje třída `importer.ThemeImporter` (byť implementace nakonec není z výše uvedených problémů s cizími klíči použita).

5. Shrnutí výsledků

Tato bakalářská práce analyzovala hlavní vlastnosti datového schématu definovaného podle JPA a předložila postup jak pomocí topologického řazení řídit import dat do takové databáze na základě vazeb mezi třídami entit. Věnovala se problému cyklických závislostí v objektovém grafu tříd entit a navrhla způsob jak identifikovat silně souvislé komponenty, s ohledem na efektivitu nalézt hrany k přerušení a po importu dat obnovit požadované schéma.

V praktické části byl vypracován způsob jak analyzovat třídy entit a získané schéma zachytit do orientovaného grafu. Byla provedena základní vizualizace popisem grafu do jazyka DOT. Byl implementován způsob nalezení silně souvislých komponent, způsob jejich analýzy a dále bylo implementováno popsané vytváření náhradních tříd entit pro rozbití silné souvislosti. Vzniklo vícevláknové řešení pro import dat do databáze pro skupiny vzájemně nezávislých vrcholů získaných na základě topologického řazení objektového grafu tříd entit.

6. Závěry a doporučení

Bakalářská práce úspěšně splnila cíle zadání. Použité algoritmy v teoretické části práce by mohly být optimalizovány a následně formálně dokázána jejich správnost. Z hlediska šíře byly popsány pouze standardní způsoby, jak jsou realizovány vazby tříd entit.

Rozsah a složitost projektu se projevily na praktické části, která nebyla implementována v produkční kvalitě, ale jako prototyp. Problematika importu byla zúžena na typické případy, a byly proto vynechány otázky dědičnost entit, vložených tříd nebo problematika složených identifikátorů a mapování původních identifikátorů na nové syntetické identifikátory. Zjednodušená objektová reprezentace nebere v úvahu paralelní hrany a zjednodušuje výpočet „ceny“ přerušení hrany silně souvislé komponenty. Změnu SQL schématu podle anotací tříd entit by bylo vhodné nahradit úplnější implementací.

Nezbytné použití JVM agenta Spring Loaded pro hot-swap bytecode zkomplikovalo packaging zdrojových kódů, a proto není dostupný java archive (soubor .jar). Spouštění z příkazové řádky nebo vývojového prostředí však vzhledem k prototypové povaze projektu není považováno za nedostatek.

7. Seznam literatury

1. ANON., 2016. *Hibernate ORM - Hibernate ORM* [online] [vid. 2016-07-30]. Dostupné z: http://hibernate.org/orm/

2. ANON., nedatováno. About SQLite. *SQLite* [online]. Dostupné z: https://www.sqlite.org/about.html

3. AT&T LABS RESEARCH, nedatováno. *Graphviz | Graphviz - Graph Visualization Software* [online] [vid. 2016-07-17]. Dostupné z: http://www.graphviz.org/

4. CORMEN, Thomas H., 2001. *Introduction to algorithms.* 2nd ed. Cambridge, Mass: MIT Press. ISBN 978-0-262-03293-3.

5. ČADA, Roman et al., 2004. *Diskrétní matematika.* Plzeň: Západočeská univerzita. ISBN 978-80-7082-939-4.

6. ČESKO, 2000. *Zákon č. 101/2000 Sb., o ochraně osobních údajů a o změně některých zákonů.* 2000.

7. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2002. *The Scala Programming Language* [online] [vid. 2016-07-17]. Dostupné z: http://www.scala-lang.org/

8. FOWLER, Martin, 2002. *Patterns of Enterprise Application Architecture.* 1st edition. Boston: Addison-Wesley Professional. ISBN 978-0-321-12742-6.

9. GASTALDI, George et al., nedatováno. *Roaster* [online] [vid. 2015-12-27]. Dostupné z: https://github.com/forge/roaster

10. CHAO, Victor, Leticia CRUZ a Nin LEI, 2005. Local versus Remote Database Access: A Performance Test. *IBM Redbooks* [online]. Dostupné z: http://www.redbooks.ibm.com/redpapers/pdfs/redp4113.pdf

11. LAFON, Yves, Roy FIELDING a Julian RESCHKE, 2014. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests* [online] [vid. 2015-09-02]. Dostupné z: https://tools.ietf.org/html/rfc7233

12. MASINTER, Larry, 1998. *RFC 2397 - The „data" URL scheme* [online]. srpen 1998. B.m.: Internet Engineering Task Force. [vid. 2015-07-13]. Dostupné z: https://tools.ietf.org/html/rfc2397

13. MONGODB, INC, 2015. *MongoDB CRUD Concepts* [online]. Dostupné z: http://docs.mongodb.org/manual/core/crud/

14. NAVEH, Barak et al., 2003. *Welcome to JGraphT - a free Java Graph Library* [online] [vid. 2016-07-30]. Dostupné z: http://jgrapht.org/

15. ORACLE AMERICA, INC., 2013. JSR 338: Java Persistence API, Version 2.1. *JSR 338: Java Persistence API, Version 2.1* [online]. Dostupné z: https://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-eval-spec/index.html

16. ORACLE AMERICA, INC., 2014. *JDBC 4.2 Specification* [online]. Dostupné z: https://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-eval-spec/index.html

17. ORACLE CORPORATION AND/OR ITS AFFILIATES, 2014a. Field Contents. *MySQL Internals Manual* [online]. Dostupné z: https://dev.mysql.com/doc/internals/en/innodb-field-contents.html

18. ORACLE CORPORATION AND/OR ITS AFFILIATES, 2014b. Java, JDBC and MySQL Types. *MySQL Connector/J Developer Guide* [online]. Dostupné z: https://dev.mysql.com/doc/connector-j/en/connector-j-reference-type-conversions.html

19. ORACLE CORPORATION AND/OR ITS AFFILIATES, 2015. *MySQL 5.6 Reference Manual* [online]. Dostupné z: http://downloads.mysql.com/docs/refman-5.6-en.a4.pdf

20. PATRICIO, Anthony, 2009. Some explanations on lazy loading (one-to-one). *JBossDeveloper* [online]. Dostupné z: https://developer.jboss.org/wiki/SomeExplanationsOnLazyLoadingone-to-one

21. PIVOTAL SOFTWARE, 2016. *Spring Framework* [online] [vid. 2016-07-30]. Dostupné z: http://projects.spring.io/spring-framework/

22. PIVOTAL SOFTWARE, nedatováno. Spring Loaded. *GitHub* [online] [vid. 2016-07-30]. Dostupné z: https://github.com/spring-projects/spring-loaded

23. QOS.CH SÀRL, nedatováno. *Simple Logging Facade for Java* [online] [vid. 2016-07-30]. Dostupné z: http://www.slf4j.org/

24. SHAFRANOVICH, Yakov, 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files* [online] [vid. 2015-09-02]. Dostupné z: https://tools.ietf.org/html/rfc4180

25. SHEPLER, Spencer et al., 2003. *Network File System (NFS) version 4 Protocol* [online] [vid. 2015-09-02]. Dostupné z: https://tools.ietf.org/html/rfc3530

26. SOMOV, Andrey, 2015. *SnakeYAML* [online] [vid. 2016-08-07]. Dostupné z: https://bitbucket.org/asomov/snakeyaml

27. STALLA, Alessio, 2012. Portofino 4 by ManyDesigns. *Configuring Hibernate programmatically, for real* [online]. Dostupné z: http://portofino.manydesigns.com/en/blog/configuring-hibernate-programmatically

28. TARJAN, Robert, 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* [online]. 6., roč. 1, č. 2, s. 146–160. ISSN 0097-5397, 1095-7111. Dostupné z: doi:10.1137/0201010

29. THE APACHE SOFTWARE FOUNDATION, 2002a. Maven – Guide to Configuring Maven. *Apache Maven Project* [online] [vid. 2016-07-17]. Dostupné z: https://maven.apache.org/guides/mini/guide-configuring-maven.html

30. THE APACHE SOFTWARE FOUNDATION, 2002b. Maven – Introduction to the Standard Directory Layout. *Apache Maven Project* [online] [vid. 2016-07-17]. Dostupné z: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

31. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2014. *PostgreSQL 9.3.5 Documentation* [online]. Dostupné

z: http://www.postgresql.org/files/documentation/pdf/9.3/postgresql-9.3-A4.pdf

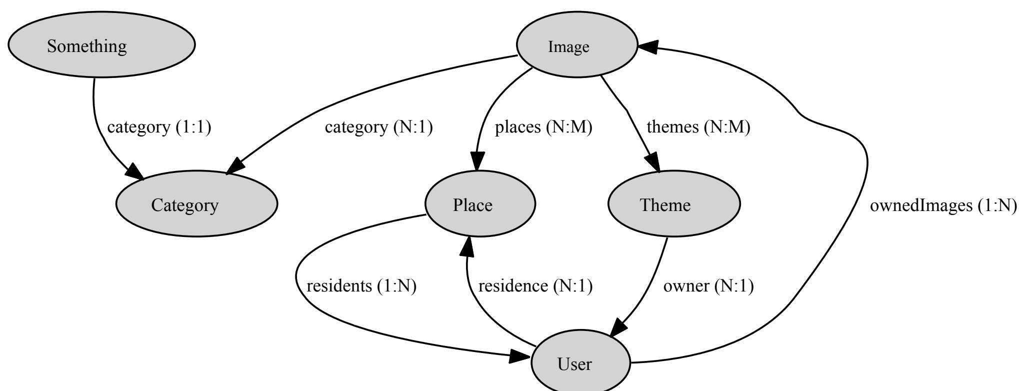
32. TIOBE SOFTWARE BV, 2014. *TIOBE Index for August 2014* [online]. Dostupné
z: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

33. UNIVOCITY SOFTWARE PTY LTD, nedatováno. Welcome to uniVocity-parsers. *GitHub* [online] [vid. 2016-07-30]. Dostupné
z: https://github.com/uniVocity/univocity-parsers

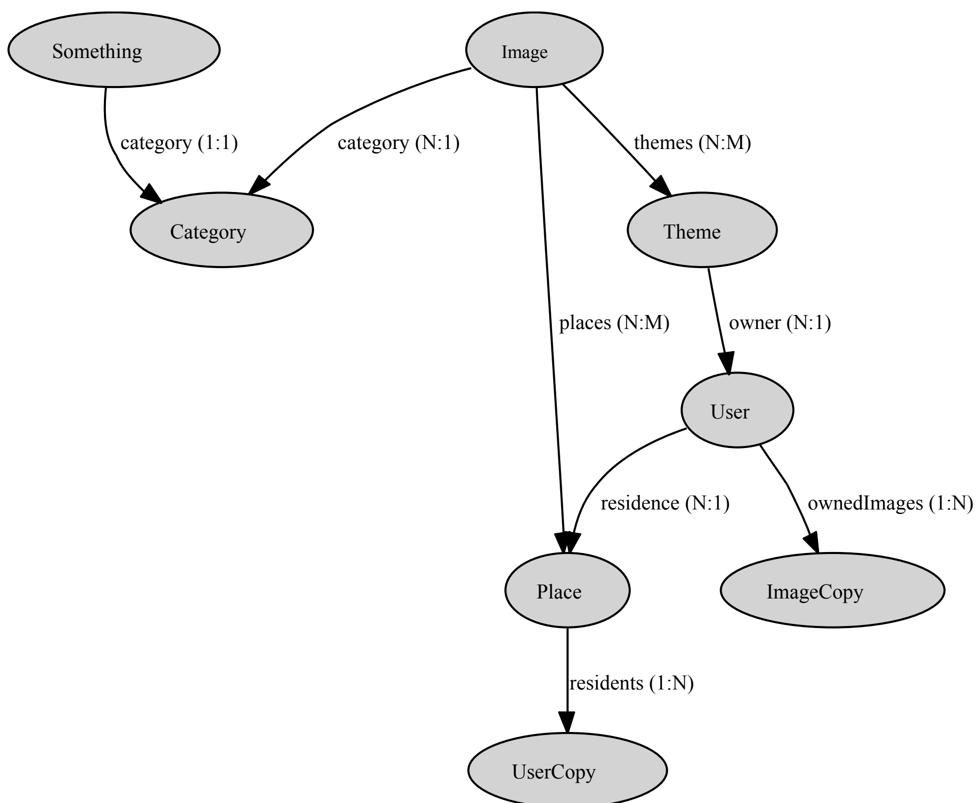
8. Příloha 1: zdrojový kód praktické části

Zdrojový kód bakalářské práce je na CD, které je vloženo do obálky vnitřní straně zadních desek práce.

9. Příloha 2: vizualizace grafů



Obr. 2: Graf obsahující cyklus Zdroj: autor



Obr. 3: Upravený acyklický graf Zdroj: autor

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Radek Dvořák**

Obor studia: Aplikovaná informatika
Jméno a příjmení vedoucího práce: **Petra Poulová**

Název práce:
Automatický import dat do databáze

Název práce v AJ:
Automatic import of data into database

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Student analyzuje specifikaci JPA2 a problémy importu dat do datového modelu. Zaměří se na vzájemné datové závislosti s ohledem na paralelizaci a cyklické závislosti. Navrhne vhodné řešení těchto problémů.

Osnova práce:

1. Úvod
2. Teoretická část
    1. ORM podle JPA2
    2. Algoritmus importu
3. Praktická část
    1. Modelový příklad
    2. Použitý software
    3. Implementace importu
4. Závěr
5. Seznam použité literatury
6. Přílohy

Projednáno dne: 14. 10. 2014

Podpis studenta

Podpis vedoucího práce