VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

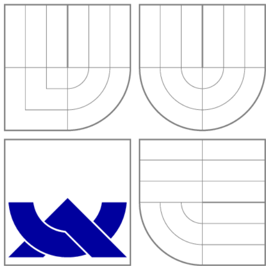
TEMPORÁLNÍ ROZŠÍŘENÍ PRO JAVA DATA OBJECTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

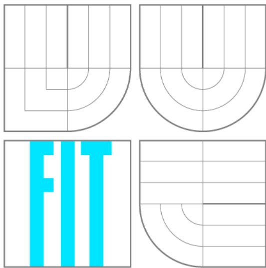
AUTOR PRÁCE
AUTHOR

Bc. JAKUB HORČIČKA

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TEMPORÁLNÍ ROZŠÍŘENÍ PRO JAVA DATA OBJECTS

A TEMPORAL EXTENSION FOR JAVA DATA OBJECTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB HORČIČKA

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2012

Abstrakt
Obsah této práce je rozdělen do pěti částí. Nejprve jsou přiblíženy principy, datové modely a některé dotazovací jazyky temporálních databází. Následuje kapitola popisující možnosti perzistence datových objektů v jazyce Java. Dále jsou uvedeny hlavní myšlenky standardu Java Data Objects, návrh temporálního rozšíření tohoto standardu a poslední kapitola uvádí detaily realizované implementace.

Abstract
The content of this thesis is devided into five parts. Firstly basic principles, data models and some languages of temporal databases are introduced. Next chapter describes ways and techniques for persistent storing of data objects in the programming language Java. Following chapters contain main ideas and key concepts of the Java Data Objects standard, draft for temporal extension of this standard and in the final chapter there are details of the actual implementation.

Klíčová slova
Temporální databáze, JDO, Java Data Objects, perzistence v jazyce Java, objektově-relační mapování.

Keywords
Temporal database, JDO, Java Data Objects, Java persistence, object-relational mapping.

Citace
Jakub Horčička: Temporální rozšíření pro Java Data Objects, diplomová práce, Brno, FIT VUT v Brně, 2012

Temporální rozšíření pro Java Data Objects

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literální prameny, ze kterých jsem čerpal.

....
Jakub Horčička
6. května 2012

Poděkování

Na tomto místě bych velice rád poděkoval svému vedoucímu za odbornou pomoc, užitečné rady, které mi během práce poskytl, a v neposlední řadě za rychlou odezvu při komunikaci.

© Jakub Horčička, 2012.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **6**

2 Temporální databáze **7**

2.1 Temporální databáze . . . 7

2.1.1 Historie . . . 7

2.2 Datové modely s podporou času . . . 8

2.2.1 Vyjádření času . . . 8

2.2.2 Druhy časů . . . 9

2.3 Referenční integrita temporálních dat . . . 10

2.4 Temporální logika . . . 11

2.5 Jazyky pro temporální dotazování . . . 12

2.5.1 TQUEL . . . 12

2.5.2 TSQL2 . . . 14

2.5.3 ATSQL . . . 18

2.6 Existující implementace temporálních databází . . . 18

3 Perzistence v Javě **19**

3.1 Podpůrné technologie . . . 19

3.1.1 Manuální práce se soubory v Javě . . . 19

3.1.2 Ukládání dat do XML souborů . . . 20

3.1.3 JDBC . . . 20

3.2 Serializace a objektově-relační mapování . . . 20

3.2.1 Serializace v Javě . . . 20

3.2.2 Objektově-relační mapování . . . 21

3.2.3 JPA . . . 22

3.2.4 Hibernate . . . 23

3.2.5 JDO . . . 27

4 Java Data Objects **28**

4.1 Specifikace JDO . . . 28

4.2 Stručná historie . . . 28

4.3 Základní pojmy . . . 29

4.4 Hlavní třídy a rozhraní . . . 29

4.4.1 JDOHelper . . . 30

4.4.2 PersistenceManagerFactory . . . 30

4.4.3 PersistenceManager . . . 30

4.4.4 Extent . . . 32

4.4.5 PersistenceCapable . . . 33

4.5 JDO architektura . . . 33
4.5.1 Použití ve dvouvrstvé architektuře . . . 33
4.5.2 Použití v rámci aplikačního serveru . . . 34
4.6 Životní cyklus JDO instancí . . . 34
4.7 Transakce . . . 35
4.8 Mapování do relační databáze . . . 36
4.8.1 Mapování vazeb . . . 39
4.8.2 Verzování a dědičnost . . . 41
4.9 JDOQL . . . 42
4.10 Dotazování . . . 42

5 Návrh temporálního rozšíření **45**
5.1 Vlastnosti a návrh jejich realizace . . . 45
5.1.1 Ukládání temporálních dat . . . 45
5.1.2 Bitemporální model dat . . . 46
5.1.3 Dotazování temporálních dat . . . 46
5.1.4 Temporální mazání . . . 46
5.1.5 Fyzické mazání . . . 46
5.1.6 Temporální referenční integrita . . . 47
5.2 UML diagramy . . . 47
5.2.1 Architektura . . . 47
5.2.2 Diagramy tříd . . . 48
5.2.3 Ukládání temporální instance . . . 50
5.2.4 Provedení dotazu . . . 52

6 Implementace **53**
6.1 Základní vlastnosti . . . 53
6.1.1 Bázová třída TemporalBase . . . 53
6.1.2 Ukládání temporálních verzí . . . 54
6.1.3 Kontrola temporální referenční integrity . . . 55
6.1.4 Shlukování . . . 56
6.1.5 Temporální dotazování . . . 57
6.1.6 Odsávání . . . 57
6.1.7 Historie dat . . . 57
6.2 Způsoby práce s entitami . . . 58
6.2.1 Temporální s ručním verzováním . . . 58
6.2.2 Temporální s automatickým verzováním . . . 58
6.2.3 Netemporální bez verzování . . . 58
6.2.4 Netemporální s verzováním . . . 59
6.3 Ukázka použití TJDO . . . 59
6.4 Omezení implementovaného systému . . . 62
6.5 Temporální vlastnosti systému . . . 63
6.6 Možná rozšíření systému . . . 63

7 Závěr **65**
7.1 Testování . . . 66

A Ukázky způsobů perzistence objektů v jazyce Java **69**

B Kompletní diagram tříd **74**

C Obsah DVD **75**

Seznam obrázků

2.1 Rozdíl mezi operátory *overlap* a *extend* 15
2.2 Základní časová linie a synchronizační body. Převzato z [17]. 15
2.3 Význam predikátů pro práci s časovými intervaly v TSQL2 [22]. 16

3.1 Jednoduchá ukázka dotazu v *Java Persistence Query Language* 23
3.2 Konfigurace Hibernate, soubor *hibernate.cfg.xml* 25
3.3 Definice objektově-relačního mapování, soubor *Person.hbm.xml* 25
3.4 Uložení a načtení objektů v Hibernate, soubor *Main.java* 26

4.1 Metody třídy JDOHelper 30
4.2 Metody rozhraní PersistenceManager 31
4.3 Metody rozhraní PersistenceManager – pokračování 32
4.4 Metody rozhraní Extent. 32
4.5 Schéma architektur *non-managed environment* (vlevo) a *managed environment* (vpravo). Převzato z [13]. 33
4.6 Diagram přechodů mezi jednotlivými stavy instancí. Převzato z [13] 35
4.7 Metody rozhraní Transaction 36
4.8 Definice Java třídy a její namapování do MySQL databáze. 37
4.9 Ukázka ORM, kdy je použito více tabulek v databázi. 38
4.10 Struktura databáze při mapování třídy do více tabulek. 38
4.11 Mapování vazby N-1. 40
4.12 Mapování vazby 1-N. 40
4.13 Mapování dědičnosti (nová tabulka) s využitím verzování. 41
4.14 Metody rozhraní Query 43
4.15 Ukázka jednoduchého dotazu. 44

5.1 Přehled jádra architektury JDO. 47
5.2 Jádro navrhovaného temporálního rozšíření JDO. 48
5.3 Část diagramu tříd týkající se datových objektů. 48
5.4 Část diagramu tříd týkající se intervalů u datových objektů. 49
5.5 Část diagramu tříd týkající se temporálních dotazů. 49
5.6 Diagram aktivity znázorňující uložení temporální instance. 51
5.7 Sekvenční diagram znázorňující proces vytvoření a provedení dotazu. . . . 52

6.1 Porušení temporální referenční integrity 56
6.2 Porušení temporální referenční integrity 57
6.3 Jednoduchá ukázka použití TJDO. 60
6.4 Ukázka editace objektu. 61
6.5 Ukázka temporálního smazání objektu. 61

A.1 Třída *Person* bude použita v některých z následujících ukázek. 69
A.2 Ukázka manuálního ukládání a načítání dat ze souboru v Javě 70
A.3 Uložení a načtení objektu pomocí JAXB. 71
A.4 Uložení a načtení stavu objektu pomocí JDBC. 72
A.5 Ukázka využití serializace objektu. 73

B.1 Diagram hlavních tříd návrhu. 74

C.1 Obsah DVD 75

Kapitola 1

Úvod

Cílem první, teoreticky zaměřené, části této práce je přiblížit čtenáři oblast *temporálních databází*, seznámit jej s možnostmi trvalého ukládání dat v jazyce *Java* a v neposlední řadě vysvětlit základní principy standardu *Java Data Objects (JDO)*. Druhá, prakticky zaměřená, část se věnuje technikám a postupům použitým při implementaci temporálního rozšíření již uvedené specifikace JDO.

První kapitola rozebírá teorii temporálních databázových systémů a uvádí jejich charakteristické rysy. V první sekci, která se věnuje historickému vývoji, bude čtenář uveden do problematiky. Dále se dozví, jak je možné čas vyjádřit nebo jaké druhy časů lze v počítačových aplikacích používat. Následovat budou sekce rozebírající temporální referenční integritu a temporální logiku, která představuje důležitý teoretický aparát, na kterém reálné systémy staví. Ke konci této kapitoly bude zmíněno několik existujících implementací temporálních systémů a počítačových jazyků, které se používají pro dotazování nad takovými systémy.

Druhá kapitola obsahuje přehled nejčastějších způsobů trvalého ukládání datových objektů v jazyce Java. Jsou zde uvedeny podpůrné technologie i existující rámce, které *perzistenci* usnadňují a částečně automatizují. Nejprve jsou uvedeny základní techniky jako například ukládání do textových souborů nebo použití relační databáze klasickým způsobem. Další sekce se však věnují pokročilejším způsobům, které využívají objektově relační mapování. Každá sekce obsahuje jednoduchou ukázku zdrojového kódu, která by měla probíranou teorii pomoci lépe a rychleji pochopit.

Třetí kapitola se zabývá standardem *JDO*. V prvních sekcích je probrána specifikace, stručná historie a základní pojmy. Následující sekce jsou detailně zaměřeny na popis hlavních tříd a rozhraní. V dalších sekcích získá čtenář přehled o celkové architektuře, o životním cyklu instancí či transakčním zpracování. V poslední sekci jsou potom uvedeny ukázky *objektově-relačního mapování*.

Čtvrtá kapitola otevírá praktickou část práce. Jsou zde diskutovány jednotlivé vlastnosti a navržen způsob implementace temporálního rozšíření standardu JDO pro účely uchovávání a dotazování temporálních objektových dat. Složitější části jsou doplněny UML diagramy pro snadnější pochopení.

Pátá kapitola začíná popisem a vysvětlením výsledných vlastností implementovaného systému včetně technik a algoritmů, pomocí kterých jich bylo dosaženo. Dále je čtenáři na jednoduchých ukázkách naznačen způsob použití. V následujících sekcích jsou uvedena omezení vytvořeného systému a diskutovány vlastnosti z temporálního hlediska. Celou kapitolu zakončují návrhy, které by mohly systém rozšířit nebo vylepšit.

Kapitola 2

Temporální databáze

2.1 Temporální databáze

Databázové systémy označované přívlastkem *temporální* patří společně s objektovými, XML, prostorovými či deduktivními do skupiny *postrelačních* databází. Tyto systémy zpracovávají specializovaná data a pro efektivní činnost nevystačí se základním relačním schématem [9]. Poskytují tedy nové datové typy a optimalizované operace. V dalším textu budou označovány jako *TDBMS (Temporal Database Management System)*, tedy databázový systém s podporou času. Alternativou může být kombinace relační databáze a složité aplikační logiky, která nahradí funkcionalitu specializované databáze. Tento přístup však bývá náročnější a často méně efektivní z hlediska rychlosti nebo času potřebného pro vývoj aplikace.

Temporální databáze je charakteristická tím, že zohledňuje časové vlastnosti ukládaných dat. Znamená to, že kromě vlastních hodnot ukládá u každého záznamu například časové razítko okamžiku vložení nebo interval platnosti (toto bude přesněji vysvětleno v sekci zabývající se časovými modely). V aplikaci je pak možné pracovat s historií a rozlišovat nová data od starých. Využití těchto vlastností je vhodné v rezervačních aplikacích nebo informačích systémech bank či pojišťoven. Další uplatnění můžeme nalézt v systémech řídících výrobní procesy, kde je nutné znát vývoj nebo historii, dále pak v účetních, skladových nebo docházkových evidencí.

S využitím temporální databáze se však pojí i jedna negativní vlastnost. Vzhledem k tomu, že je zapotřebí uchovávat údaje o historii, není možné staré záznamy jednoduše mazat. Tím však vzniká situace, kdy do databáze neustále data pouze přibývají, čímž do nekonečna rostou požadavky na diskovou paměť. Tento problém se často řeší tak, že se po určitém období stará a neplatná data fyzicky odstraní. To však narušuje podstatu temporální databáze, neboť již není možné sledovat všechna data, která kdy byla vložena.

2.1.1 Historie

Vývoj temporálních databází probíhal a stále probíhá paralelně s vývojem databází jako takových. V roce 1992 navrhl Richard Snodgrass temporální rozšíření jazyka SQL (Structured Query Language), který se běžně používá v relačních databázích. Na základě toho byla vytvořena rozšíření standardu SQL-92 (konkrétně se jedná o ANSI X3.135.-1992 a ISO/IEC 9075:1992) známá jako TSQL2. K jejich implementaci došlo o rok později. Specifikace jazyka se pak objevila v dubnu 1994 a v září téhož roku došlo k oficiálnímu vydání. Rok 1999 byl ve znamení snah o vytvoření nového SQL standardu (SQL3), ale tyto pokusy byly neúspěšné. [25]

2.2 Datové modely s podporou času

Každý databázový systém je založený na *datovém modelu*, který definuje formalismy, s jejichž pomocí lze spravovaná data popsat, modifikovat a získávat jednotným způsobem. Dále poskytuje prvky pro popis datových struktur, integritních omezení a operací manipulujících se strukturami. Formálně jde o trojici $MODEL = (DS, OP, IO)$, kde DS značí datové strukruty, OP operace a IO integritní omezení.

Datový model s podporou času přidává do všech položek podporu práce s časovými údaji. To znamená, že datové strukury jsou schopny uchovávat časově proměnlivá data, operace jsou přizpůsobeny pro podporu časových aspektů a integritní omezení zahrnují časové údaje spojené s vlastními záznamy. Formálně se datový model s podporou času zapisuje opět jako trojice, ale tentokrát s horními indexy T. $MODEL^T = (DS^T, OP^T, IO^T)$ [19].

2.2.1 Vyjádření času

V reálném světě představuje čas fyzikální veličinu, kterou lze měřit a určovat, ale z filozofického pohledu existuje řada teorií a názorů, které se mohou výrazně lišit. V počítačových programech a systémech je však potřeba tento pojem přesně vymezit. Základní dělení rozlišuje následující tři časové modely [19].

1. Spojitý (continuous) časový model.
2. Hustý (dense) časový model.
3. Diskrétní (discrete) časový model.

Spojitý časový model zachycuje všechny časové body vyjádřitelné reálnými čísly. To znamená, že mezi libovolnými dvěma časovými body existuje další. Jedná se o nejpřesnější model ve vztahu s realitou. Z technického hlediska však není možné a často ani žádoucí tento model implementovat. Proto je k dispozici *hustý časový model*, který zachycuje body vyjádřitelné racionálními čísly. Takovýto model je již technicky realizovatelný a také se používá. Posledním z výše uvedených je *diskrétní časový model*, ve kterém lze jednotlivé body na ose zapisovat celými čísly. Zde platí, že mezi libovolným bodem a jeho následníkem již neexistuje žádný jiný.

Časovou osu si lze představit jako množinu bodů, což mimo jiné znamená, že je oboustranně omezená (má svůj počátek a konec). Nejmenší časovou dále nedělitelnou jednotkou je *Chronon*. V případě diskrétního modelu jsou všechny body časové osy zároveň chronony, neboť menší jednotka než celé číslo zde není k dispozici. U ostatních modelů však tento fakt neplatí. Při návrhu temporální databáze se určí velikost chrononu, který takto omezí množinu přípustných hodnot časových bodů. Jako příklad uveďme systém založený na hustém modelu. Teoreticky je možné vyjádřit hodnoty z množiny $\{0,25; -8,75; 1010,50; 0,0; 4,2\}$, ale pokud určíme chronon jako $0,1$, budeme schopni vyjádřit pouze $\{0,3; -8,8; 1010,5; 0,0; 4,2\}$.

Při vyjadřování konkrétních časových údajů lze použít *časové okamžiky (instants)*, *intervaly (intervals)* nebo *periody (periods)* [16]. Časový okamžik je jeden konkrétní chronon na časové ose. Vyskytuje se právě jednou a nadále se s ním pracuje jako s údajem v minulosti. Ve standardu SQL-92 je zastoupen datovým typem *Datetime*. Příkladem může být začátek počítačového času (resp. Unixu): 1.1.1970 0:00:00. Interval reprezentuje úsek na časové ose. Definuje jeho délku, ale nespecifikuje počáteční ani koncový okamžik. Lze s ním tedy vyjadřovat doby trvání, které není potřeba přesně zasazovat do historického kontextu.

SQL datový typ *Interval* je realizován jako struktura uchovávající počet let, měsíců, dnů, hodin, minut a sekund. Příkladem může být stáří člověka: 42 let, 6 měsíců, 8 dní, 5 hodin, 53 minut a 10 sekund. Třetí možností, jak vyjádřit časový údaj, je perioda. Jde o konkrétní úsek časové osy jednoznačně určený svým počátečním a koncovým okamžikem. Jako příklad uveďme vymezení letního semestru v roce 2011: 7.2.2011 – 6.5.2011.

2.2.2 Druhy časů

V rámci TDBMS můžeme rozlišit tři druhy časů v závislosti na jejich významu a způsobu použití. V následujícím textu budou blíže přiblíženy a vysvětleny.

1. Uživatelský, popř. uživatelsky definovaný čas (user-defined time).
2. Čas platnosti (valid time).
3. Transakční čas (transaction time).

Uživatelský čas

V případě uživatelského času se jedná o běžný datový typ jako je třeba číslo nebo řetězec. Uživatel databáze sám přiřazuje význam hodnotám uloženým pod tímto typem. Je dostupný během všech dotazů a příkazů, kde se používá tradičním způsobem (jde o sloupec tabulky). SQL pro tyto účely běžně poskytuje datové typy DATE, TIME, DATETIME nebo INTERVAL. Hodnoty tohoto typu nemají žádný temporální vliv na TDBMS a pro časové aspekty databáze se nepoužívají. Příkladem použití může být atribut tabulky uchovávající informaci o datu narození nějaké osoby nebo časové razítko posledního přihlášení do informačního systému.

Čas platnosti

Společně s transakčním časem představují to, čím se temporální databáze odlišuje od relační. Nejedná se o volně dostupný atribut tabulky, který by bylo možné používat v dotazech klasickým způsobem. TDBMS oba tyto časy používá vnitřně pro temporální sémantiku.

Čas platnosti, jak už název napovídá, slouží k uchování informace o tom, kdy je daný záznam platný v reálném světě. Tedy ve světě, který je databázovým systémem modelován. Ukládaný interval platnosti nemusí být pouze od přítomnosti do budoucnosti. Zcela běžně se ukládají intervaly, které jsou celé v minulosti nebo celé v budoucnosti.

Představme si aplikaci, která uchovává kurzy měn. Pro jednoduchost budeme uvažovat pouze české koruny vůči americkému dolaru. Jak jistě všichni víme, jedná se o velmi proměnlivou hodnotu (pokud chceme být přesní). Budeme-li údaje o vývoji kurzu ukládat do databáze, bude velmi důležité společně s vkládaným počtem korun ukládat i časové razítko začátku (popřípadě konce) platnosti. Díky tomu bude později vůbec možné jednotlivé záznamy chronologicky uspořádat a použít pro užitečné statistiky.

Transakční čas

Jak již bylo uvedeno výše, transakční čas je důležitý pro vnitřní potřeby TDBMS. Slouží k zaznamenání informace o času provedení změn v tabulkách. Kdykoli se provede nějaká aktulizace nebo vložení nového záznamu, dojde také k uložení časového razítka provedení tohoto příkazu. Poté je tedy možné dohledat, kdy a k jakým změnám došlo. Transakční čas

se na rozdíl od času platnosti neuvádí explicitně, ale je ukládán systémem automaticky. Další rozdíl představuje nemožnost aktualizace takovýchto údajů. Pokud tedy uložíme záznam a následně jej budeme chtít upravit (např. kvůli chybě), přepíšeme čas platnosti, ale přidáme nový transakční čas. Při dotazech pak budeme mít k dispozici jeden čas platnosti, ale dva časy transakční, čímž budeme mít uloženu informaci o provedené změně.

Pokračujme v příkladu aplikace pro správu kurzů měn. Představme si situaci, kdy je aplikace nově nainstalována a databáze neobsahuje žádná data. Nyní budeme chtít vložit údaje za uplynulé období. Budeme tedy ukládat záznamy s časem platnosti, který bude celý v minulosti, ale transakční čas bude vždy v přítomnosti.

2.3 Referenční integrita temporálních dat

Referenční integrita je způsob, kterým databázový systém vyjadřuje vztahy v tabulkách. Základem jsou pojmy *primární klíč* a *cizí klíč*. Primární klíč je jednoznačný identifikátor záznamu v tabulce, v rámci které je unikátní (a ideálně i neměnný). Cizí klíč představuje odkaz na primární klíč jiné, odkazované, tabulky. Příklad v tabulce 2.1 tedy uvádí, že Alois je zaměstnán jako automechanik, Bořek jako truhlář a Cyril jako zedník s příslušnými platy.

PK	**Jmeno**	**CK_Zamestnani**
1	Alois	2
2	Bořek	3
3	Cyril	1

PK	**Zamestnani**	**Plat**
1	Zedník	10 000
2	Automechanik	15 000
3	Truhlář	20 000

Tabulka 2.1: Primární a cizí klíč v relační databázi.

V případě temporální databáze však takto uvedený princip nebude fungovat. Abychom mohli pracovat s historií, bude potřeba pro identifikaci vazeb mezi tabulkami začít uvažovat čas platnosti nebo transakční čas (popř. oba) [16]. V obou tabulkách 2.2 jsme proto přidali sloupec *Platnost*. Primární klíč je nyní složen ze dvojice *ID* a *Platnost*.

Uvažujme pouze historii pana Aloise, který v rámci své kariéry vystřídal dvě zaměstnání (automechanik a truhlář) a tři různé platy (15 000, 20 000 a 25 000). V první tabulce je jeden záznam s platností $1980 - \infty$. Jde o tzv. *shluk*. Více logických záznamů lišících se pouze svojí platností, které na sebe navazují, je spojeno v jeden celek. Tento přístup je vhodný kvůli nižším paměťovým nárokům a rychlejšímu zpracování (např. během dotazů). Ve druhé tabulce jsou ale záznamy s hodnotu ID rovnou jedné celkem tři. Nejde o celý primární klíč, a proto se nejedná o chybu. Jednoznačná identifkace vyžaduje ID a interval definující platnost záznamu. Můžeme si všimnout, že v letech 1980 až 1990 pracoval pan Alois jako automechanik s platem 15 000. Poté změnil zaměstnání a do roku 2000 pracoval jako truhlář s platem 20 000. Nakonec dostal přidáno na 25 000, což je i jeho aktuální mzda.

Při manipulaci s platností údajů je důležité pamatovat na to, aby v celém období platnosti odkazujícího záznamu existovaly i záznamy odkazované. Kdyby v uvedeném příkladu řekněme chyběl řádek, kdy byl pan Alois automechanikem, nebyla by databáze v konzistentním stavu. Při dotazu na pana Aloise v období 1980 – 1989 by neexistovala informace o jeho zaměstnání, což představuje problém.

ID	Platnost	Jmeno	CK_Zamestnani
1	1980 – ∞	Alois	1
2	1990 – 1999	Bořek	2
3	2010 – ∞	Cyril	3

ID	Platnost	Zamestnani	Plat
1	1980 – 1989	Automechanik	15 000
1	1990 – 1999	Truhlář	20 000
1	2000 – ∞	Truhlář	25 000
2	1990 – 1999	Truhlář	20 000
3	2010 – ∞	Zedník	10 000

Tabulka 2.2: Primární a cizí klíč v temporální databázi.

2.4 Temporální logika

Temporální logika spadá do oblasti formální logiky. Na rozdíl od predikátové logiky poskytuje předložky a predikáty, jejichž pravdivost záleží na čase. Umožňuje tedy vyjádřit fakta jako například „Někdy v budoucnu si pořídím nový počítač“ nebo „Dokud nebudu mít dostatek peněz, žádný počítač si nekoupím“. Jde tedy o fakta, která v průběhu času mohou změnit svou pravdivost.

Temporální logika prvního řádu je temporální logika, která obsahuje predikátové symboly, proměnné, kvantifikátory a časové spojky. V souvislosti s temporálními databázemi tvoří obecný dotazovací jazyk. Její vznik se datuje do 50. let 20. století, kdy byla autorem Arthur Prior navržena pro reprezentaci a dedukci nad přirozenými jazyky. Později byla použita jako nástroj pro formální verifikaci softwarových systémů a nakonec se stala základem pro dotazovací jazyky v temporálních databázových systémech. Konkrétně se využívá *Linear Time First Order Temporal Logic*, což zjednodušeně znamená, že se uvažuje čas platnosti [3].

Čas, který je v rámci logiky uvažován, může být buď lineární (LTL – Linear Temporal / Time Logic) nebo větvený (CTL – Computation Tree Logic). LTL představuje lineární posloupnost událostí, kde každý stav má přesně jednu možnou budoucnost. Naopak CTL umožňuje reprezentovat více potenciálních budoucností každého stavu. Syntaxe temporální logiky prvního řádu je následující.

$$Q \ ::= r(x_{i_1}, \ldots, x_{i_k}) \mid x_i = x_j \mid Q \ \wedge Q \mid \neg Q \mid \exists x.Q \mid Q \ since \ Q \mid Q \ until \ Q \tag{2.1}$$

K syntaxi predikátové logiky tedy přidává spojky *since* a *until*, které se používají v českém významu „od určité doby (since)“ a „do určité doby (until)“. Tyto spojky je možné vyjádřit následujícím způsobem [3].

$$X_1 \ since \ X_2 := \exists t_2.t_0 > t_2 \wedge X_2 \wedge \forall t_1(t_0 > t_1 > t_2 \rightarrow X_1) \tag{2.2}$$

$$X_1 \ until \ X_2 := \exists t_2.t_0 < t_2 \wedge X_2 \wedge \forall t_1(t_0 < t_1 < t_2 \rightarrow X_1) \tag{2.3}$$

Všechny formule mají relativní význam, vztahují se vždy k určitému časovému okamžiku (evaluation point). Nejčastěji se jedná o aktuální čas (právě teď), ale je možné jej posunout kamkoli do minulosti nebo budoucnosti. Společně se spojkami since a until se často používá

několik dalších. Teoreticky nejsou potřeba, protože je lze vyjádřit pomocí již definovaných spojek, ale pro praktické účely jsou vhodné.

$\blacklozenge X \ = true\ since\ X$ – někdy v minulosti

$\blacksquare X \ = \neg\blacklozenge\neg X$ – vždy v minulosti

$\bullet X \ = false\ since\ X$ – předchozí

$\Diamond X \ = true\ until\ X$ – někdy v budoucnosti

$\Box X \ = \neg\Diamond\neg X$ – vždy v budoucnosti

$\circ X \ = false\ until\ X$ – následující

2.5 Jazyky pro temporální dotazování

V relačních databázích se dnes nejčastěji používá dotazovací jazyk *SQL (Structured Query Language)*, který je vyvíjen od 70. let 20. století. Od té doby vzniklo několik standardů, které přidávaly nové prvky a funkcionalitu. Jde o deklarativní počítačový jazyk, který se snaží co nejvíce přiblížit přirozené angličtině. Dle [26] můžeme rozlišovat čtyři základní části tohoto jazyka.

1. Příkazy pro manipulaci dat (select, insert, update, ...).
2. Příkazy pro definici dat (create, alter, drop, ...).
3. Příkazy pro řízení přístupových práv (grant, revoke, ...).
4. Příkazy pro řízení transakcí (commit, rollback, ...).

Z pohledu temporální podpory je důležitý standard z roku 1992 (SQL-92 nebo SQL2), který zavádí operace s podporou času a datové typy DATE, TIME, TIMESTAMP a INTERVAL [6].

2.5.1 TQUEL

TQUEL (Temporal Query Language) je temporální nadstavba dotazovacího jazyka QUEL, který byl vytvořen pro účely relačního databázového systému Ingress [23]. Dnes je QUEL již dlouhou dobu nahrazen jazykem SQL, ale ve své době byl vybrán pro svou jednoduchost. TQUEL byl syntakticky i sémanticky velmi podobný, takže dotazy a příkazy byly QUEL kompatibilní.

Dotazy se skládají ze dvou hlavních částí: cílový seznam (datová struktura obsahující výsledek dotazu) a části *where*, která specifikuje požadovaná data. V tabulce 2.3 jsou záznamy osob, nad kterými provedeme následující dotaz.

```
range of o is Osoby
retrieve into Deti (Rok = o.Rok_narozeni)
  where o.Rok_narozeni > 1996
```

PK	Jmeno	Rok_narozeni
1	Karel	1972
2	Karolína	1984
3	Petr	2001
4	Jana	2005

Tabulka 2.3: Databázová tabulka *Osoby* – příklad pro QUEL dotaz.

Klíčové slovo *range* určuje tabulky, kterých se bude dotaz týkat. Literál *o* slouží jako zástupce tabulky *Osoby* pro zbytek dotazu. Výsledek bude uložen do asociativního pole *Deti*, kde na indexu *Rok* bude hodnota sloupce *Rok_narozeni*. Záznamy, které budou vybrány, určuje podmínka za klíčovým slovem *where*. V ukázce jde tedy o osoby, které se narodily po roce 1996, čili *Petr a Jana*.

V tomto případě jde o klasický relační dotaz (*tzv. snapshot*), který pro naše účely není příliš zajímavý. To se však změní, začneme-li v části where používat operátory *begin of*, *end of*, *overlap*, *extend*, nebo *precede*. Další možností je konstrukce *valid from . . . to . . .* [15].

Následuje několik ukázek temporálních dotazů v jazyce TQUEL, které používají databázi uvedenou v tabulce 2.4. Tato data jsme již částečně použili v podsekci vysvětlující druhy časů v temporální databázi. V tomto případě jsme ale rozdělili sloupec platnost na začátek a konec platnosti, cizí klíče do tabulky zaměstnání jsme pro jednoduchost nahradili přímo názvy povolání a přidali jsme několik dalších záznamů.

ID	Začátek_platnosti	Konec_platnosti	Jmeno	Zamestnani	Plat
1	1980	1989	Alois	Automechanik	15 000
1	1990	1999	Alois	Truhlář	20 000
1	2000	∞	Alois	Truhlář	25 000
2	1985	1999	Bořek	Truhlář	20 000
2	2000	∞	Bořek	Truhlář	25 000
3	1985	1994	Cyril	Zedník	10 000
3	1995	∞	Cyril	Truhlář	25 000

Tabulka 2.4: Tabulka *Osoby* – testovací data pro ukázky dotazů v jazyce TQUEL.

- Ukázka použití operátorů *begin of* a *overlap*. Cílem bude zjistit plat pana Aloise v momentě, kdy pan Bořek nastoupil do zaměstnání. Dotaz tedy budeme specifikovat tak, že platnost požadovaného záznamu (o panu Aloisovi) bude přesahovat začátek platnosti záznamu o panu Bořkovi.

```
range of a is Osoby
range of b is Osoby
retrieve into Alois (Plat = a.Plat)
  where a.Jmeno = "Alois" and b.Jmeno = "Bořek" and
    a overlap begin of b
```

- Dále využijeme operátor *precede* a *end of*. Budeme chtít získat platovou historii pana

Aloise do roku 2000. Konce platnosti požadovaných záznamů tedy budou předcházet roku 2000.

```
range of o is Osoby
retrieve into Alois (Plat = o.Plat)
  where (end of a) precede "2000"
```

- V této ukázce použijeme konstrukci *valid from ... to ...*. Výsledek dotazu bude obsahovat historii zaměstnání pana Aloise za období, kdy je pan Bořek zaměstnán. Požadované záznamy pana Aloise tedy budou platné v období začínajícím nástupem pana Bořka do prvního zaměstnání a končícím v okamžiku, kdy pan Bořek ukončí kariéru (dle aktuálních dat v nekonečnu).

```
range of a is Osoby
range of b is Osoby
retrieve into Alois (Zamestnani = a.Zamestnani)
  valid from begin of b to end of b
  where a.Jmeno = "Alois" and b.Jmeno = "Bořek"
```

- Nakonec nám zbývá operátor *extend*. Budeme chtít zjistit platovou historii pana Bořka od okamžiku, kdy pan Alois i pan Cyril změnili své povolání na truhláře. Sjednotíme tedy intervaly, kdy pan Alois pracoval jako automechanik a pan Cyril jako zedník. Toto období bude předcházet všem požadovaným záznamům o panu Bořkovi.

```
range of a is Osoby
range of b is Osoby
range of c is Osoby
retrieve into Bořek (Plat = B.Plat)
  where a.Jmeno = "Alois" and b.Jmeno = "Bořek" and
    c.Jmeno = "Cyril" and a.Zamestnani = "Automechanik" and
    c.Zamestnani = "Zedník" and (a extend c) precede b
```

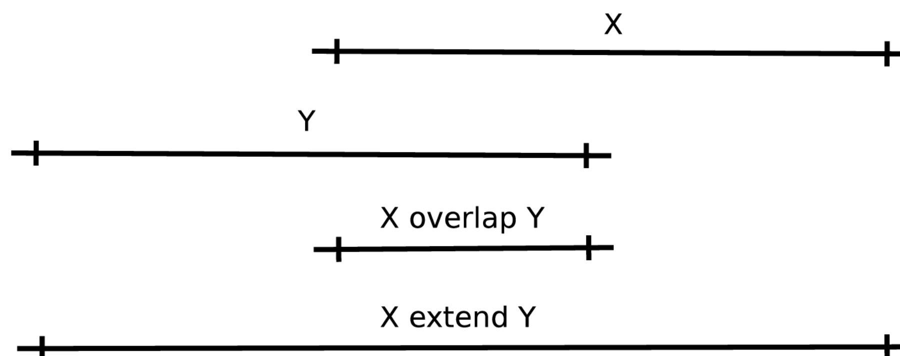
Na obrázku 2.1 je znázorněn rozdíl mezi operátory overlap (průnik) a extend (sjednocení) [15].

2.5.2 TSQL2

TSQL2 (Temporal Structured Query Language) je temporální rozšíření standardu SQL-92, se kterým je zpětně kompatibilní (TSQL2 rozumí SQL-92 příkazům, ale ne opačně). První návrhy specifikace vznikly v roce 1993 a první kompletní specifikace o rok později. Za vývojem stojí skupina vedená Richardem Snodgrassem.

Ontologie času

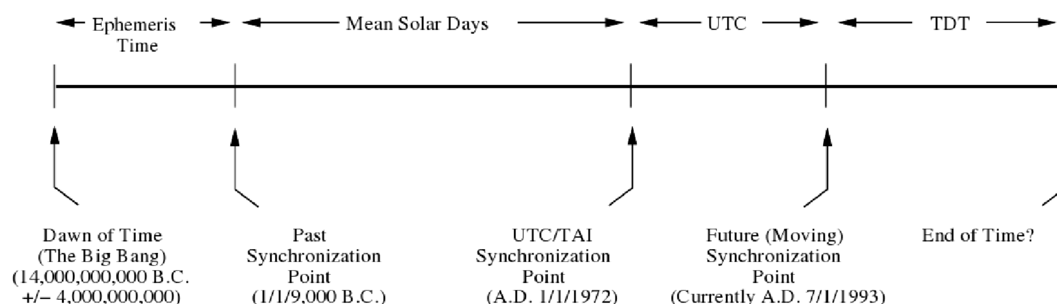
Reprezentace časové osy je v TSQL2 oboustranně ohraničená. Má tedy absolutní počátek i konec. Model zastupuje všechny tři způsoby reprezentace času (spojitý, hustý, diskrétní),



Obrázek 2.1: Rozdíl mezi operátory *overlap* a *extend*

čehož dosahuje pomocí časových okamžiků menších než jsou chronony. Z uživatelského hlediska jde však o transparentní vlastnost, neboť nelze explicitně zvolit konkrétní způsob [17].

SQL-92 definuje časové údaje ve formátu *UTC (Coordinated Universal Time)* sekund. To ovšem znamená, že nedokáže definovat (pre)historické údaje. TSQL2 však používá koncept základní časové linie (*baseline clock*) založený na časových razítkách (*timestamps*), který je uveden na obrázku 2.2. Čas je rozdělen na jednotlivé úseky pomocí synchronizačních bodů. Reprezentace a formát času jsou v každém úseku různé.



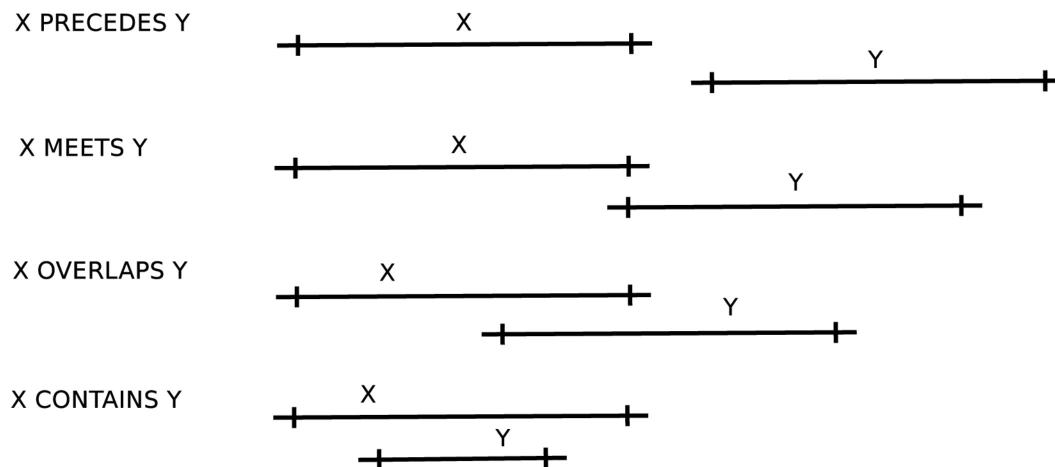
Obrázek 2.2: Základní časová linie a synchronizační body. Převzato z [17].

Datové typy

TSQL2 přebírá datové typy z SQL-92 a rozšiřuje tuto množinu o nový datový typ – *PERIOD*, u kterého je možné specifikovat rozsah a přesnost. Dále zavádí tzv. *surrogate data (zásupce)*, což jsou unikátní, jednoznačné a pro uživatele skryté identifikátory porovnatelné na shodnost. Slouží pro vnitřní potřeby databáze při práci s temporálními atributy a nejde o náhradu cizích nebo primárních klíčů [17].

Predikáty

Pro práci s časovými intervaly jsou k dipozici následující predikáty: *PRECEDES*, *MEETS*, *OVERLAPS* a *CONTAINS* [?, 22]. Jejich význam je znázorněn na obrázku 2.3.



Obrázek 2.3: Význam predikátů pro práci s časovými intervaly v TSQL2 [22].

Bitemporální model dat

Databázové tabulky bez podpory času (netemporální) jsou označovány jako *snímkové (snapshot)*. Tento typ zůstává i v TSQL2, ale navíc jsou přidány i tabulky *stavové (state)*. Ve stavové tabulce je každému záznamu přiřazena datová struktura obsahující časová razítka. Nejde však o další běžně dostupný sloupec. Pomocí této struktury je možné definovat různé datové modely, k dispozici je celkem šest druhů tabulek [17], které jsou uvedeny níže.

- *Snapshot* (snímkové tabulky) – pouze uživatelsky definovaný čas, žádná temporální podpora ze strany databáze.
- *Valid-time state* (tabulky s časem platnosti) – uchovávají období platnosti jednotlivých záznamů.
- *Valid-time event* (tabulky s časem platnosti) – uchovávají okamžik platnosti jednotlivých záznamů. Od valid-time state se liší tím, že ukládají záznamy platící v konkrétní okamžik a ne po konkrétní období nebo dobu.
- *Transaction-time* (tabulky s transakčním časem) – uchovávají transakční čas jednotlivých záznamů.
- *Bitemporal state* (tabulky s bitemporálním modelem) – jde o kombinaci valid-time state a transaction-time.
- *Bitemporal event* (tabulky s bitemporálním modelem) – jde o kombinaci valid-time event a transaction-time.

Další vlastnosti

- *Odsávání (vacuuming)* – proces, při kterém jsou z temporální databáze fyzicky odstraněny požadované záznamy.
- *Restrukturalizace a shlukování (restructuring, coalescing)* – při temporálních dotazech mohou vzniknout takové výsledky, kdy se záznamy např. z hlediska času platnosti

překrývají. TSQL2 v takovém případě umožní sloučení těchto záznamů dohromady. Restrukturalizace znamaná, že výsledná tabulka bude odpovídat provedeným sjednocením. Bude tedy obsahovat požadované sloupce bez vztahů k ostatním v dotaze nespecifikovaným sloupcům, které jsou jinak obsaženy ve zdrojových tabulkách.

Ukázky dotazů v TSQL2

Následuje několik ukázek dotazů a příkazů v jazyce TSQL2. Další příklady je možné nalézt v [18].

- Definice schématu tabulky – příkaz `CREATE TABLE`. Části `AS VALID STATE` ve spojení s `AND TRANSACTION` značí, že jde o bitemporální tabulku. `YEAR (2)` definuje rozsah platnosti na 100 let ($10^2 = 100$) a `TO DAY` určuje přesnost záznamu na jeden den.

```
CREATE TABLE Osoby (
  Jmeno CHARACTER (30) NOT NULL,
  Zamestnani CHARACTER (20) NOT NULL,
  Plat DECIMAL (10, 2) NOT NULL
) AS VALID STATE YEAR (2) TO DAY AND TRANSACTION;
```

- Vložení záznamu – příkaz `INSERT INTO`.

```
INSERT INTO Osoby (Jmeno, Zamestnani, Plat)
  VALUES ('Alois', 'Automechanik', 15000.00)
  VALID PERIOD '[1980-01-01 - 1989-12-31]';
```

- Výběr záznamu – příkaz `SELECT`. Klíčové slovo `SNAPSHOT` značí, že výsledkem bude snímková tabulka. Konstukce `VALID(O) OVERLAPS PERIOD '[1982-01-01 - 1985-01-01]'` požaduje záznam, který platí v zadaném období.

```
SELECT SNAPSHOT O.Plat FROM Osoby O
  WHERE O.Jmeno = 'Alois' AND
  VALID(O) OVERLAPS PERIOD '[1982-01-01 - 1985-01-01]';
```

- Zneplatnění záznamu – příkaz `DELETE`. Nejde o fyzické odstranění dat z databáze.

```
DELETE FROM Osoby WHERE Jmeno = 'Alois';
```

- Fyzické odstranění záznamů – příkaz `VACUUM`. Uvedený příklad odstraní všechny záznamy z tabulky `Osoby`, které byly vloženy do roku 2010.

```
VACUUM Osoby WHERE TRANSACTION (Osoby) PRECEDES DATE '2010-01-01';
```

2.5.3 ATSQL

ATSQL je temporální rozšíření standardu SQL-92. Vychází z koncepce TSQL2 a ChronoLogu (temporální rozšíření jazyka Prolog). Hlavní myšlenkou v ATSQL jsou *stavové modifikátory (statement modifiers)*, které se používají v podobě klíčových slov *SEQUENCED* a *NONSEQUENCED* před SQL-92 dotazy [4].

- *Klasické SQL-92 dotazy* – nepoužívají stavový modifikátor a vztahují se k aktuálnímu času. Nelze využívat temporálních atributů (čas platnosti nebo transakční čas) u záznamů.
- *SEQUENCED VALID dotazy* – výsledkem je chronologicky seřazená (využití temporální relace) množina výsledků.
- *NONSEQUENCED VALID dotazy* – výsledek je specifikován pouze dotazem. Databázový systém neprovádí žádnou činnost spojenou s temporálními atributy, ale tyto je možné v dotazu používat například pomocí predikátu *VTIME()*.

2.6 Existující implementace temporálních databází

- *Oracle Workspace Manager:* PL/SQL balíček v Oracle databázích umožňující verzování tabulek pomocí automaticky přidávaných metadat. Podporuje čas platnosti i transakční čas.

  Příklad nastavení verzování tabulky (zapnutí podpory času platnosti):

  ```
  EXECUTE DBMS_WM.EnableVersioning
    ('nazevTabulky', 'VIEW_WO_OVERWRITE', FALSE, TRUE);
  ```

  U jednotlivých záznamů se čas platnosti definuje pomocí datového typu `WM_PERIOD`. K dispozici je řada funkcí a predikátů pro práci s temporálními atributy záznamů [10, 11].

- *PostgreSQL:* Pro databázový systém PostgreSQL je k dispozici open-source balíček *pgsql-temporal*, který přidává nové datové typy, operátory a funkce pro jednodušší práci s časem v PostgreSQL databázích [1].
- *TimeDB:* Bitemporální relační databázový systém využívající ATSQL2 (mix TSQL2, ChronoLogu a Bitemporal ChronoSQL). První verze byla napsána v jazyce Prolog. Druhá verze je však založená na jazyce Java a využívá *JDBC (Java Database Connectivity)* – jednotné API pro přístup k relačním databázím. TimeDB funguje jako *frontend* ke komerčním databázovým systémům od společnosti *Oracle*. Veškeré ATSQL2 dotazy jsou vnitřně převedeny na SQL-92 formát a takto provedeny v relační databázi (*backend*) [21].
- *Teradata R13.10:* Softwarová společnost, která vyvíjí relační databázový systém *Teradata*. Verze R13.10 zavádí podporu temporálních objektů (tabulky, pohledy, makra) a dotazů [20].

Kapitola 3

Perzistence v Javě

Java je objektově orientovaný interpretovaný programovací jazyk. Jeho hlavní výhodou je možnost širokého uplatnění díky přenositelnosti na různé platformy. Toho se dosahuje implementacemi *JVM (Java Virtual Machine)*, což je běhové prostředí, které interpretuje předaný Java kód. Základní dělení rozlišuje verze pro čipové karty (JavaCard), mobilní telefony (JavaME), desktopové aplikace (JavaSE) a distribuované systémy (JavaEE). Tato kapitola se bude zabývat možnostmi *perzistence* java objektů. Tedy způsoby, jak ukládat, permanentně uchovávat a zpětně načítat stavy instancí tříd napsaných v jazyce Java.

Libovolná data lze ukládat do textových souborů, ať už ve vlastním nebo specializovaném (např. *XML – Extensible Markup Language*) formátu, nebo do databáze. Nejčastěji jsou k dispozici databáze *relační*, kdy je potřeba řešit tzv. *relační mapování* objektů (*ORM – Object Relation Mapping*). Další možností jsou *objektové* nebo *XML* databáze. V následujících sekcích budou rozebrány techniky perzistence včetně *rámců (framework)*, které využívají ORM. Smyslem celé kapitoly bude nastínění různých možností tak, aby si čtenář mohl lépe zařadit technologii *JDO*, která je tématem této práce, a které se bude detailněji věnovat celá následující kapitola.

V některých ukázkách zdrojových kódů bude použita třída `Person`, která je na obrázku A.1.

3.1 Podpůrné technologie

3.1.1 Manuální práce se soubory v Javě

Ukládání dat do datových souborů (ať už textových nebo binárních) představuje jednoduché a rychlé řešení. Pokud jsou však ukládaná data strukturovaná a předpokládá se jejich častá editace, pak při větším množství není tento způsob efektivní. Další problémy souvisí s nemožností vícenásobného přístupu k jednomu souboru (zejména z hlediska současného zápisu) [12].

Představme si situaci, kdy potřebujeme v souboru s daty o velikosti jednotek MB doplnit řádek uprostřed záznamu. Nejdříve vyhledáme pozici, od které budeme chtít zapisovat. Dále bude pravděpodobně nutné načíst zbytek souboru od nalezené pozice a tato data uložit na jiné místo. Po zápisu nových údajů se dočasně přemístěná data opět zapíší do původního souboru. Podobná situace může nastávat velmi často a výkon aplikace, která by s daty pracovala uvedeným způsobem, by byl velmi omezen.

Na obrázku A.2 je ukázka manuálního uložení a opětovného načtení dat z textového souboru.

3.1.2 Ukládání dat do XML souborů

XML je značkovací metajazyk, který lze používat pro popis strukturovaných dat. Je relativně dobře čitelný pro lidi a existuje mnoho nástrojů, jak s ním efektivně pracovat programově. XML dokumenty jsou obyčejné textové soubory, což představuje jednoduché použití a šíření.

V porovnání s metodou serializace eliminuje XML formát část jejich nedostatků. Data jsou popsána strukturovaně a lze je jednoduše měnit i rozšiřovat. V případě rozšíření rozhraní třídy je stále možné z uložených údajů sestavit funkční objekt. Na druhou stranu ale zůstává problém s nedostatečnou podporou vyhledávání, nutností načítat všechny objekty pro nalezení konkrétního či absence transakcí.

Na obrázku A.3 je ukázáno použití *JAXB (Java Architecture for Xml Binding)*. Pomocí tříd z balíčku `javax.xml.bind` uložíme a opětovně načteme vytvořený objekt.

3.1.3 JDBC

Java Database Connectivity (JDBC) je rozhraní sjednocující přístup k relačním databázím v Javě. Při konkrétním nasazení je zapotřebí speciální *ovladač (driver)*, který řeší implementační detaily zvoleného databázového systému. Aplikaci je tedy možné napsat jednou a pak použít ve spojení s libovolnou databází, která poskytuje JDBC ovladač. Pro práci s JDBC je potřeba třída `java.sql.DriverManager` a rozhraní `java.sql.Connection`, `java.sql.Statement` a `java.sql.ResultSet` [12].

- `java.sql.DriverManager` – třída zprostředkovávající komunikaci s databází. Poskytuje instanci připojení a umožňuje komunikovat s JDBC ovladačem.
- `java.sql.Connection` – uchovává instanci připojení získanou od DriverManager. Poskytuje instanci dotazu.
- `java.sql.Statement` – uchovává instanci dotazu získanou od Connection. Přebírá SQL dotazy a příkazy jako své parametry.
- `java.sql.ResultSet` – jde o *databázový kurzor*, tedy prostředek pro procházení výsledků dotazu.

Na obrázku A.4 je uveden způsob vložení záznamu do předem připravené tabulky (SQL kód je uveden níže) a jeho opětovné načtení při použití *MySQL* databáze.

```
-- Vytvoreni struktury tabulky 'persons' v MySQL databazi
CREATE TABLE persons (
  id int auto_increment primary key,
  name varchar(60) not null default ''
);
```

3.2 Serializace a objektově-relační mapování

3.2.1 Serializace v Javě

Serializace je obecný proces transformace stavu objektu (hodnot jeho atributů) do posloupnosti dat, kterou lze sériově zpracovávat.

V jazyce Java je k dispozici rozhraní `java.io.Serializable`, které umožňuje jednotný a jednoduchý způsob, jak perzistentně ukládat objekty do datových souborů. Principiálně tedy jde o metodu popsanou v předchozí kapitole, ale v tomto případě je poskytnuto *API (Application Programming Interface)*, které tuto činnosti částečně automatizuje, usnadňuje a zpřehledňuje.

Na obrázku A.5 je uvedena ukázka použití serializace pro uložení a načtení stavu objektu. Nejprve je potřeba definovat třídu implementující rozhraní `Serializable`. V našem případě jde o dříve uvedenou třídu `Person` na obrázku A.1. Dále je potřeba vstupní bod programu, tedy statická metoda `main`, a vlastní kód serializace.

Kromě obtíží spojených s principem ukládání dat do datových souborů přináší serializace i své vlastní problémy. Pokud dojde ke změně atributů či metod serializované třídy, nebude již možné dříve uložená data do objektu aktualizované třídy načíst, a to i v případě, že došlo pouze o rozšíření původní třídy, kdy všechny uložené atributy jsou nadále podporovány.

Další úskalí představuje granularita načítání. Není možné získat pouze určité atributy, ale pouze celé objekty [12].

3.2.2 Objektově-relační mapování

ORM (Object-Relational Mapping) je automatizovaný převod objektů v programovacím jazyce do tabulek v relační databázi. Pro tyto účely se využívá speciálních metadat, která popisují způsob provedení těchto operací. Dle [2] je možné v každé implementaci ORM rozlišovat následující části.

- Rozhraní umožňující operace vytvoření, načtení, modifikaci a odstranění objektu určených pro perzistenci.
- Jazyk nebo rozhraní pro vytváření dotazů.
- Prostředí nebo způsob, jak určit mapovací metadata.
- Technika umožňující transakce a optimalizaci.

Zřejmě jako každá technologie, i ORM má své výhody a nevýhody. Vzhledem k tomu, že se nejedná o jeden nástoj, který by bylo možné použít (jde o techniku), nabízí se otázky k zamyšlení [2].

- Jak mají vypadat perzistentní třídy nebo v jakém měřítku je proces perzistence automatizován?
- Jakým způsobem se budou definovat metadata?
- Jakým způsobem budou provázány instance tříd s konkrétními záznamy v tabulkách?
- Jak se bude mapovat hierarchie dědičnosti?
- Jak se bude vzájemně ovlivňovat perzistenční a aplikační logika?
- Jaký je životní cyklus perzistentního objektu?
- Jakým způsobem je řešeno řazení, vyhledávání nebo agregace?
- Jak efektivně získávat data?

Nyní uvedeme výhody spojené s používáním konkrétního ORM nástroje.

- Programátor se nemusí zabývat SQL dotazy, které jsou často komplikované a nepřehledné.
- Kód spojený s perzistencí je částečně automatizován, zjednodušen a odstíněn od aplikační logiky. Výsledkem by měl být kratší čas pro vývoj požadované aplikace.
- Kratší a přehlednější kód je srozumitelnější a vhodnější ke změnám či úpravám.
- Nezávislost na použitém databázovém systému, ORM rozhraní je jednotné a rozdíly na nižší úrovni řeší nástroj sám.

3.2.3 JPA

Java Persistence API (JPA) je specifikace firmy *Oracle* (JSR-317) definující způsob správy a perzistence objektů v JavaEE nebo JavaSE. Pro definici objektově-relačního mapování používá tzv. *anotace*. Jde o způsob, jak třídě, atributu, balíčku nebo metodě přiřadit metadata ve zdrojovém kódu určená pro JVM. K tomuto účelu slouží rozsáhlá množina klíčových slov, které jsou uvozeny znakem zavináč (@).

Vzhledem k tomu, že je JPA pouze specifikace, je v praxi nutné použít konkrétní implementaci. K dispozici jsou například *Oracle Toplink, Toplink Essentials, EclipseLink, OpenJPA, JPOX nebo Hibernate*. Poslední uvedený, Hibernate, bude přiblížen v následující sekci.

Základním stavebním kamenem při používání JPA je definice entitní třídy pomocí anotace `@Entity`. Jde o běžnou třídu definující objekty, které se budou ukládat. Navíc však musí splňovat následující požadavky [5].

- Použití anotace `@Entity` nebo označení v XML deskriptoru (speciální XML soubor definující ORM).
- Definice bezargumentového konstruktoru jako `public` nebo `protected`.
- Nejde o výčtový typ (`enum`) nebo rozhraní (`interface`).
- Třída, metody ani atributy určené pro ukládání nesmí být definovány jako `final`.
- Pokud se bude instance třídy předávat hodnotou, je potřeba implementovat rozhraní `Serializable`.
- Třída podporuje dědičnost a polymorfisum.
- Může jít o abstraktní třídu (`abstract class`).
- Všechny ukládané atributy jsou dostupné pomocí veřejných metod (tzv. *getter* a *setter* metody).
- Kromě základních datových typů je možné pro atributy použít `java.util.Collection`, `java.util.Set`, `java.util.List` a `java.util.Map`.

Pokud je mezi dvěma ukládanými entitami nějaký vztah, vyjadřuje se pomocí vazeb definovaných opět pomocí anotací. Vztahy mohou být jednosměrné (*unidirectional*), kdy referenci obsahuje pouze jedna strana, nebo obousměrné (*bidirectional*), kdy se obě strany vidí navzájem.

- `@OneToOne` – vztah mezi jednou odkazující a jednou odkazovanou entitou (příklad: osoba má jeden občanský průkaz).
- `@OneToMany` – vztah mezi jednou odkazující a více odkazovanými entitami (příklad: sportovní tým má několik hráčů).
- `@ManyToOne` – vztah mezi více odkazjícími a jednou odkazovanou entitou (příklad: hráči hrající za stejný tým).
- `@ManyToMany` – vztah mezi více odkazujícími a více odkazovanými entitami (příklad: zaměstnanci a jejich nadřízení).

Všechny ukládané entity (kontext perzistence) jsou spravovány instancí třídy `EntityManager`, která ovlivňuje jejich životní cyklus. Každý objekt v kontextu perzistence se může nacházet v některém z následujících stavů.

- *Nově vytvořená instance* – počáteční stav.
- *Aktuálně spravovaná entita* – entita, na kterou byla zavolána metoda `persist`.
- *Odpojená entita* – stav po dokončení transakce nebo zavolání metody `close`.
- *Odstraněná entita* – entita, na kterou byla zavolána metoda `remove`.

Java Persistence Query Language

Pro účely dotazování definuje JPA vlastní dialekt SQL jazyka. Jeho výhody plynou z nezávislosti na použité relační databázi. JPA zajišťuje překlad na konkrétní SQL kód, který je potřeba na nižší úrovni (v databázi). Další výhodou je podpora objektové notace, kdy se v dotazech neobjevují názvy relačních tabulek a jejich sloupců, ale názvy tříd a jejich atributů. Tím se sytaxe přibližuje více k Javě. Je možné využívat vztahů a vnořených objektů pomocí speciálních operátorů. Na obrázku 3.1 je jednoduchá ukázka získání objektu z databáze.

```
SELECT DISTINCT p
FROM Person p
WHERE p.id = 1 AND p.name = 'Alois Andrle'
```

Obrázek 3.1: Jednoduchá ukázka dotazu v *Java Persistence Query Language*

3.2.4 Hibernate

Hibernate je implementace JPA specifikace od firmy *JBoss (dnes Red Hat)*. Lze ji použít na platformách Java a *.NET*. Objektově relační mapování je možné popsat pomocí anotací nebo XML souborů. Základní koncepce a teorie byly popsány v předchozí sekci, a proto se nyní zaměříme převážně na praktickou ukázku.

Moduly

Implementace nástoje Hibernate je rozdělena do modulů, které lze používat samostatně nebo je podle potřeby kombinovat [2].

- *Hibernate Core* – základní API pro perzistenci s popisem mapování v XML souborech. Poskytuje vlastní dialekt dotazovacího jazyka známého jako *HQL (Hibernate Query Language)*. Všechny ostatní moduly jsou rozšířením tohoto jádra, lze ho ale použít i samostatně.
- *Hibernate Annotations* – umožňuje popsat mapování pomocí anotací ve zdrojovém kódu namísto v XML souborech.
- *Hibernate EntityManager* – rozhraní pro správu perzistentních objektů a jejich životních cyklů. Používá se společně Hibernate Annotations.
- *Java EE 5.0 application servers* – podpora pro servery *JBoss AS* a *Tomcat*.

Dále je k dispozici řada rozšiřujících nástrojů [7].

- *Hibernate Shards* – umožňuje použití více relačních databází pro Hibernate Core.
- *Hibernate Search* – poskytuje textové vyhledávání a stará se o indexování v databázi.
- *Hibernate Tools* – zásuvný modul pro vývojové prostředí *Eclipse* usnadňující práci s Hibernate.
- *Hibernate Validator* – referenční implementace *JSR 303 – Bean Validation*. Porovnává zadané anotace s kódem, ke kterému se vztahují, a upozorňuje na možné chyby. V principu jde o jednoduchou sémantickou kontrolu kódu.
- *Hibernate Metamodel Generator* – nabízí možnost automatického vygenerování metamodelu z uvedených anotací. Metamodel je způsob, jak sjednodit přístup k datům. Ať jsou data fyzicky uložena v databázi libovolného typu nebo v datových souborech, metamodel poskytuje rozhraní, jak k nim přistupovat stejným způsobem.
- *Hibernate OGM (Object/Grid Mapper)* – zajišťuje použití JPA pro různá datová úložiště, která nejsou založená na jazyce SQL.

Praktická ukázka

Projekt, jehož zdrojové soubory budou přiblíženy v této podsekci, je vytvořen ve vývojovém prostředí *NetBeans verze 6.9.1*. Jde o Java SE konzolovou aplikaci, která ukládá a načítá instance třídy `Person` (obrázek A.1) s využitím relační databáze *MySQL*. Do projektu byla přidána knihovna *Hibernate*, která je součástí *IDE (Integrated Development Environment)*.

Nejprve byl vytvořen konfigurační soubor Hibernate – *hibernate.cfg.xml* (obrázek 3.2). V něm jsou důležité údaje pro připojení k databázi (ovladač, url, login a heslo) a uvedení souboru, ve kterém bude uvedeno objektově-relační mapování.

Dalším krokem je ona definice mapování objektů do databáze. Potřebné informace jsou v souboru *Person.hbm.xml* na obrázku 3.3. Je zde uvedeno, že se primární klíč (atribut `id`) bude generovat automaticky, tudíž ho při vytváření objektů nebudeme zadávat.

```
1 <!DOCTYPE hibernate−configuration PUBLIC
2 "−//Hibernate/Hibernate Configuration DTD 3.0//EN"
3 "http://hibernate.sourceforge.net/hibernate−configuration−3.0.dtd">
4
5 <hibernate−configuration>
6   <session−factory>
7     <property name="dialect">
8       org.hibernate.dialect.MySQLDialect
9     </property>
10    <property name="hibernate.connection.driver_class">
11      com.mysql.jdbc.Driver
12    </property>
13    <property name="hibernate.connection.url">
14      jdbc:mysql://localhost:3306/java
15    </property>
16    <property name="hibernate.connection.username">username</property>
17    <property name="hibernate.connection.password">password</property>
18    <property name="connection.pool_size">1</property>
19    <property name="show_sql">true</property>
20    <mapping resource="Person.hbm.xml" />
21  </session−factory>
22 </hibernate−configuration>
```

Obrázek 3.2: Konfigurace Hibernate, soubor *hibernate.cfg.xml*

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate−mapping PUBLIC
3 "−//Hibernate/Hibernate Mapping DTD 3.0//EN"
4 "http://hibernate.sourceforge.net/hibernate−mapping−3.0.dtd">
5
6 <hibernate−mapping>
7   <class name="Person" table="persons">
8     <id name="id" column="id">
9       <generator class="native"/>
10    </id>
11    <property name="name" column="name" type="java.lang.String" />
12  </class>
13 </hibernate−mapping>
```

Obrázek 3.3: Definice objektově-relačního mapování, soubor *Person.hbm.xml*

Nakonec definujeme soubor *Main.java* se vstupním bodem aplikace. Zde provedeme transakci, během které uložíme nově vytvořený objekt a následně načteme a vypíšeme všechny uložené (obrázek 3.4).

```
1  import java.util.Iterator;
2  import java.util.List;
3  import org.hibernate.Criteria;
4  import org.hibernate.Session;
5  import org.hibernate.SessionFactory;
6  import org.hibernate.Transaction;
7  import org.hibernate.cfg.Configuration;
8
9  public class Main {
10   public static void main(String[] args) {
11     SessionFactory sessionFactory =
12       new Configuration().configure().buildSessionFactory();
13     Session session = sessionFactory.openSession();
14
15     // a) ulozeni objektu
16     Person firstPerson = new Person();
17     firstPerson.setName("Alois Andrle");
18     session.saveOrUpdate(firstPerson);
19
20     Transaction tx = session.beginTransaction();
21     tx.commit();
22
23     // b) nacteni objektu
24     Criteria criteria = session.createCriteria(Person.class);
25     List persons = criteria.list();
26
27     Iterator it = persons.iterator();
28
29     while (it.hasNext()) {
30       Person person = (Person) it.next();
31       System.out.println("Nacteno: id = " + person.getId() +
32         ", name = " + person.getName());
33     }
34
35     session.close();
36     sessionFactory.close();
37   } // main
38 } // class
```

Obrázek 3.4: Uložení a načtení objektů v Hibernate, soubor *Main.java*

3.2.5 JDO

Java Data Objects (JDO) je, stejně jako například JPA, specifikace perzistence v jazyce Java. Umožňuje ukládat běžné Java třídy (tzv. *POJO (Plain Old Java Object*) a nevyžaduje implementaci žádného speciálního rozhraní. Jedná se o technologii, jejíž nasazení začalo v roce 2006 a zatím poslední verze (3.0) vyšla v roce 2010. Metadata pro ORM jsou definována v externích XML souborech podobně jako v případě Hibernate. Data mohou být fyzicky uložena v relačních či objektových databázích nebo v datových souborech [24].

Konkrétní implementace JDO poskytují specializované nástroje (tzv. *enhancers*), které umožňují modifikovat zkompilované Java soubory (s příponou *.class*) tak, aby instance tříd v těchto souborech definované bylo možné ukládat. A to dokonce napříč různými JDO implementacemi.

Tato sekce uvádí JDO specifikaci jen pro úplnost v přehledu možností Java perzistence. Bližšímu popisu, vybraným detailům specifikace a konkrétním ukázkám se bude věnovat celá následující kapitola.

Kapitola 4

Java Data Objects

4.1 Specifikace JDO

Java Data Objects (JDO) je vedle JPA další specifikace perzistence objektů v jazyce Java. Historie JPA a JDO byla a je provázána. Obě tyto technologie mají mnoho společných rysů, ale i prvky, ve kterých se liší. Přehledná tabulka s rozdíly je k dispozici na [8]. Obecně lze říci, že JDO v aktuální verzi (3.0) je nadmnožinou JPA.

Cílem JDO je stejně jako v případě JPA umožnit programátorům transparentní způsob perzistence objektů, který jim umožní se více zaměřit na aplikační logiku místo toho, aby opakovaně a pracně řešili způsob uložení a získávání dat. Další výhodu přináši vlastní dotazovací jazyk, který se svou syntaxí blíží Javě a je nezávislý na konkrétní použité databázi na nižší vrstvě.

Tato kapitola se bude detailněji zabývat právě specifikací JDO. Nejdříve přiblížíme historii vývoje a poté definujeme základní pojmy, které budou použity v následujících sekcích. Dále popíšeme životní cyklus instancí a hlavní třídy v JDO rozhraní. Pokračovat budeme s charakteristikou mapování do relační databáze, strukturou používaných XML souborů, nejdůležitějšími výjimkami a vlastním dotazovacím jazykem, který JDO definuje. Důležité části budou doplněny ukázkami za použití konkrétní JDO implementace.

4.2 Stručná historie

Počátky JDO můžeme hledat v *ODMG (Object Data Management Group)*, což byl první pokus o standardizaci transparentního přístupu k databázi z objektově orientovaných jazyků. V té době (počátek 90. let) se rozhodovalo mezi jazyky *C++* a *Smalltalk*. Java byla vyvinuta až později v rámci *JCP (Java Community Process)*.

Vývoj JDO probíhal paralelně s *JCA (Java Connector Architecture)* a *CMP (Container Managed Persistence)*. JCA slouží pro připojení aplikačních serverů a komerčních informačních systémů k databázi. CMP je koncept perzistence používaný v *EJB (Entity Java Beans)*, což jsou serverové komponenty v Java EE.

První verze JDO specifikace vyšla v roce 1999 jako *JSR 12*. Následovala verze 2.0 z roku 2006 pod označením *JSR 243*, která byla rozšiřována až do 2.2. V roce 2010 pak vyšla zatím poslední verze 3.0 (stále JSR 243), kterou se zabývá tato kapitola.

4.3 Základní pojmy

V následujícím seznamu jsou přiblíženy základní pojmy související s JDO, které jsou uvedeny ve specifikaci [14].

- *JDO Instance* – základem je běžná instance Java třídy, jejíž data (stav) pochází ze systémového souboru nebo databáze. Navíc však musí umožňovat uložení stavu objektu, tedy uložení všech atributů. Toto omezení nesplňují systémové třídy jako jsou například `System, Thread, Socket` nebo `File`.
- *JDO Implementation* – množina tříd, které implementují JDO specifikaci.
- *JDO Vendor* – autor poskytující JDO implementaci. Často přidává vlastní optimalizace nebo rozšíření specifikace, které se hodí pro konkrétní účel.
- *JDO Enhancer* – k tomu, aby byly perzistentní třídy binárně kompatibilní napříč JDO implementacemi, je potřeba, aby implementovaly jistá JDO rozhraní. *Enhancer* nebo také *byte code enhancer* je program modifikující zkompilované Java třídy tak, aby byl tento požadavek splněn.
- *Enterprise Information System (EIS)* – systém poskytující podnikovou informační infrastrukturu a služby pro klienty.
- *EIS Resource* – poskytuje specifickou funkcionalitu pro klienty.
- *Resource Manager (RM)* – spravuje množinu EIS zdrojů. Poskytuje rozhraní pro klienty, kteří přes něj využívají zdroje.
- *Connection* – umožňuje spojení klienta s RM a provádí transakce.
- *Application Component* – část aplikace. Na serverové straně může jít o *EJB (Enterprise Java Beans), JSP (Java Server Pages)* nebo *servlet*. Na straně klienta jde například o *Java applet*.
- *Session Beans* – objekty v EJB, které mají na starosti relaci jednoho klienta a s tím spojená data (každý klient má jeden objekt typu Session Bean).
- *Message-driven Beans* – objekty v EJB starající se o odpovědi na požadavky jednoho klienta.
- *Entity Beans* – objekty v EJB poskytující přímou objektovou reprezentaci dat v databázi.
- *Helper objects* – komponenty poskytující objektový pohled na data. Jsou používány v session beans nebo entity beans.
- *Container* – součást serveru, která zajišťuje nasazení aplikace a její běhovou podporu.

4.4 Hlavní třídy a rozhraní

V této sekci budou přiblíženy nejdůležitější (existuje jich celá řada) třídy a rozhraní v JDO specifikaci, které se využívají pro perzistenci [13].

4.4.1 JDOHelper

`JDOHelper` je třída starající se o zavedení JDO a dokáže zjistit stav požadované instance z hlediska životního cyklu, který bude vysvětlen později. Všechny její metody, které jsou uvedeny na obrázku 4.1, jsou statické. Není tedy potřeba vytvářet konkrétní objekt, i když tato možnost existuje.

- `getPersistenceManager` – vrací objekt správce instancí.
- `makeDirty` – nastaví konkrétní atribut zadané instance do stavu *dirty*, což je příznak určující, že došlo ke změně v rámci aktuální transakce a bude potřeba uložit změny do databáze.
- `getObjectId` – vrací JDO identifikátor zadané instance.
- `getTransactionalObjectId` – obdoba `getObjectId` s tím rozdílem, že pokud byla identita (primární klíč) nějak změněna v rámci transakce, vrací tuto změněnou identitu.
- `isDirty` – zjistí, zda byla zadaná instance změněna v rámci aktuální transakce.
- `isTransactional` – zjistí, zda je zadaná instance součástí aktuální transakce.
- `isPersistent` – zjistí, zda je zadaná instance v perzistentním stavu.
- `isNew` – zjistí, zda byl stav zadané instance v aktuální transakci nově nastaven na perzistentní.
- `isDeleted` – zjistí, zda byla zadaná instance v aktuální transakci smazána.
- `getPersistenceManagerFactory` – obě dvě varianty této metody slouží k získání objektu, který načte zadané konfigurační údaje a poskytne objekt `PersistenceManager`.

Obrázek 4.1: Metody třídy JDOHelper

4.4.2 PersistenceManagerFactory

`PersistenceManagerFactory` je rozhraní použité v každé JDO implementaci. Konkrétní třída s tímto rozhraním pak slouží jako zdroj objektu `PersistenceManager`. Ten lze vytvořit i ručně bez použití „továrny“, ale jde o výjimečný případ, protože továrna poskytuje metody pro nastavení a získání všech vlastností konfigurace. Změna nastavení je však možná pouze do okamžiku, kdy je vytvořena první instance, což je první zavolání metody `getPersistenceManager` nebo `getPersistenceManagerFactory`.

4.4.3 PersistenceManager

`PersistenceManager` je opět rozhraní vyskytující se v každé JDO implementaci. Konkrétní instance funguje jako správce perzistentních instancí. Ovlivňuje jejich životní cyklus a stará se o obsah vyrovnávací paměti (*cache*). Přehled poskytovaných metod je na obrázcích 4.2 a 4.3.

- `evict` – odstranění zadaného objektu z cache.
- `evictAll` – 3 varianty pro odstranění většího počtu (nebo všech) objektů z cache.
- `refresh` – obnova zadaného objektu v cache.
- `refreshAll` – 3 varianty pro obnovu většího počtu (nebo všech) objektů v cache.
- `retrieve` – 3 varianty načtení objektů z cache.
- `makePersistent` – požadavek na vytvoření nové entity v databázi.
- `makePersistentAll` – 2 varianty pro vytvoření většího počtu nových entit.
- `deletePersistent` – odstranění zadaného perzistentního objektu z databáze.
- `deletePersistentAll` – 2 varianty pro odstranění většího počtu objektů z databáze.
- `makeTransient` – nastaví stav zadaného objektu na *transient* (bude vysvětleno v sekci o životním cyklu instance).
- `makeTransientAll` – 2 varianty pro hromadné nastavení většího počtu objektů do stavu *transient*.
- `makeTransactional` – nastaví stav zadaného objektu na *transactional* (bude vysvětleno v sekci o životním cyklu instance).
- `makeTransactionalAll` – 2 varianty pro hromadné nastavení většího počtu objektů do stavu *transactional*.
- `makeNontransactional` – nastaví stav zadaného objektu na *non-transactional* (bude vysvětleno v sekci o životním cyklu instance).
- `makeNontransactionalAll` – 2 varianty pro hromadné nastavení většího počtu objektů do stavu *non-transactional*.
- `getObjectIdClass` – vrací *class descriptor* třídy, jejíž objekt je zadán.
- `getObjectId` – vrací JDO identitu zadané instance.
- `getTransactionalObjectId` – vrací nejnovější JDO identitu v rámci aktuální transakce.
- `newObjectIdinstance` – vytvoří nový objekt na základě zadaného *class descriptor*u a textové reprezentace identifikátoru objektu.
- `close` – uzavření instance `PersistenceManager`. Po této operaci již všechny ostatní metody (až na `isClose`) vyhazují výjimku.
- `isClose` – vrací `true` pokud již byla zavolána metoda `close`.

Obrázek 4.2: Metody rozhraní PersistenceManager

- `setUserObject / getUserObject` – přiřazení (získání) objektu pod správu `PersistenceManager`a.
- `getPersistenceManagerFactory` – získání instance „rodičovské továrny“.
- `currentTransaction` – získání objektu spravující transakci. Každý `PersistenceManager` má právě jeden.
- `setMultithreaded / getMultithreaded` – nastavuje (získává) příznak vícevláknové podpory.
- `setIgnoreCache / getIgnoreCache` – nastavuje (získává) příznak ovlivňující provádění dotazů z hlediska využití cache.
- `getExtent` – vrací *extent* zadané třídy a jejích podtříd. Extent je množina všech instancí konkrétní třídy.
- `getObjectById` – získání objektu podle jeho identifikace.
- `newQuery` – 9 různých variant sloužících pro vytvoření objektu spravující dotaz.

Obrázek 4.3: Metody rozhraní PersistenceManager – pokračování

4.4.4 Extent

Objekt třídy implementující rozhraní `Extent` představuje množinu všech perzistentních instancí stejné třídy. V JDO je však možné pomocí metod jednotlivé instance přidávat nebo odebírat. K dispozici je `Iterator`, pomocí kterého je možné všemi uchovávanými instancemi procházet. Přehled metod, které rozhraní poskytuje, je na obrázku 4.4.

- `iterator` – vrací objekt, pomocí kterého lze procházet jednotlivé instance.
- `hasSubclasses` – vrací příznak zahrnutí podtříd do extentu. Nemusí to však nutně znament, že daná třída nějaké podtřídy skutečně obsahuje.
- `getCandidateClass` – vrací třídu, jejíž extent je spravován.
- `getPersistenceManager` – vrací rodičovský objekt `PersistenceManager`.
- `close` – uzavře zadaný objekt typu `Iterator`.
- `closeAll` – uzavře všechny interní objekty typu `Iterator`.

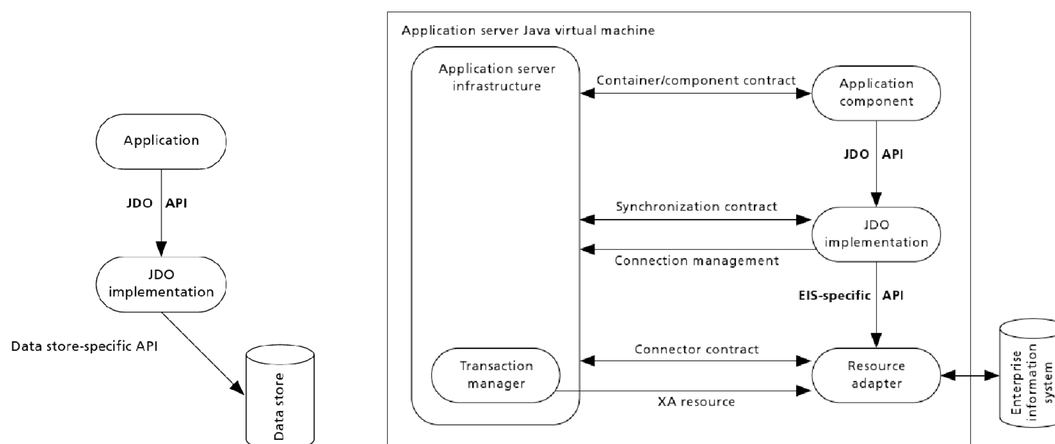
Obrázek 4.4: Metody rozhraní Extent.

4.4.5 PersistenceCapable

`PersistenceCapable` patří mezi interní JDO rozhraní, které nejsou určeny pro aplikační programátory. Každá třída, kterou spravuje `PersistenceManager` musí toto rozhraní implementovat. Možnosti, jak toho dosáhnout jsou ale dvě. První z nich je ruční definice všech potřebných metod, které jsou charakteristické svým perfixem *jdo*. Tento přístup se v praxi však příliš nepoužívá, neboť JDO umožňuje automatizovanou variantu. Jde o tzv. *enhancement (povýšení)* tříd. K dispozici je nástoj zvaný *Enhancer*, který dokáže modifikovat zkompilované (.class) soubory a vložit do nich vyžadované metody.

4.5 JDO architektura

Z hlediska prostředí je možné JDO využívat dvěma způsoby. Prvním z nich je tzv. *non-managed environment* (obrázek 4.5 vlevo), kdy jsou služby implementace využívány přímo v aplikační logice. To znamená ruční konfiguraci `PersistenceManagerFactory`, získání instance `PersistenceManager`, správu transakcí (*begin, commit, rollback*) a provádění perzistentních operací. Typické použití zahrnuje získání jedné instance `PersistenceManager` hned na začátku. Ta je potom dostupná až do ukončení aplikace.



Obrázek 4.5: Schéma architektur *non-managed environment* (vlevo) a *managed environment* (vpravo). Převzato z [13].

Druhým prostředím je *managed environment* (obrázek 4.5 vpravo). V tomto případě je JDO integrováno do aplikačního serveru. Konfigurace a získání instance třídy implementující rozhraní `PersistenceManager` je přenecháno na *JNDI (Java Naming Directory Interface)*, což je jednotné Java rozhraní pro práci s adresářovými službami. Přístup k datům je zprostředkován přes *JCA (Java Connector Architecture)* a instance PersistenceManager, kterých může být více, jsou z důvodu jednotné správy seskupeny ve vyrovnávací paměti zvané *pool*. Díky tomu jsou zdroje efektivněji využívány a uzavírány v momentě, kdy již nejsou potřeba [13].

4.5.1 Použití ve dvouvrstvé architektuře

Dvouvrstvá architektura znamená přímou komunikaci klienta s databází bez mezilehlé vrstvy, která by se starala o aplikační logiku na straně serveru. Jedná se o nejjednodušší způsob pou-

žití JDO, kdy jsou zapotřebí dvě hlavní rozhraní: `javax.jdo.PersistenceManager` (spravuje dotazy, transakce a životní cyklus instancí perzistentních tříd) a `javax.jdo.JDOHelper` (poskytuje další služby jako např. získání instance PersistenceManagerFactory).

4.5.2 Použití v rámci aplikačního serveru

Při použití JDO v rámci aplikačního serveru je využito JCA, které definuje rozhraní mezi aplikačním serverem a EIS. JDO implementace poskytuje rozhraní `ManagedConnectionFactory`, `XAResource` a `LocalTransaction`. Instance třídy `PersistenceManagerFactory` slouží pro konfiguraci tzv. *resource adapter*, což je ovladač sloužící k připojení ke správci zdrojů (JDBC nebo nativní: produkty pro připojení k objektově-relační databázi či objektová databáze). [14].

4.6 Životní cyklus JDO instancí

JDO instance může v průběhu své existence z hlediska perzistence nabývat různých stavů. V této sekci budou přiblíženy jednotlivé stavy a přechody mezi nimi. Aktuální stav je možné získat přes `PersistenceManager`, který poskytuje metody jako např. `isDeleted`, `isNew` nebo `isPersistent`. V praxi je možné rozlišovat dvě skupiny stavů. Jednu definuje samotná JDO specifikace a druhou konkrétní JDO implementace. Výrobci si tedy mohou přidávat a používat vlastní stavy. V následujícím seznamu budou popsány stavy z té první skupiny, tedy stavy dle JDO specifikace [13].

- *Transient* – ve stavu *transient (dočasný)* se nachází každý objekt ihned po jeho vytvoření. Nemá žádnou souvislost s perzistentním prostředím, nepodporuje transakce a nikdy nevyvolá žádnou JDO výjimku (JDO výjimky budou vysvětleny v samostatné sekci).

- *Persistent-New* – v tomto stavu se nachází instance, které se staly perzistentními v aktuální transakci a získaly tak JDO identitu (identifikátor).

- *Persistent-New-Deleted* – pokud se instance v rámci jedné, aktuální, transakce dostane do stavu Persistent-New a současně je poté odstraněna, je ve stavu *Persistent-New-Deleted*. Přístup k atributům vyvolá výjimku `JDOUserException`.

- *Hollow* – instance v tomto stavu reprezentují data v databázi, ale konkrétní hodnoty ještě nebyly načteny. Slouží převážně pro vnitřní potřeby JDO, kdy se tímto způsobem zajišťuje jednoznačnost v transakcích.

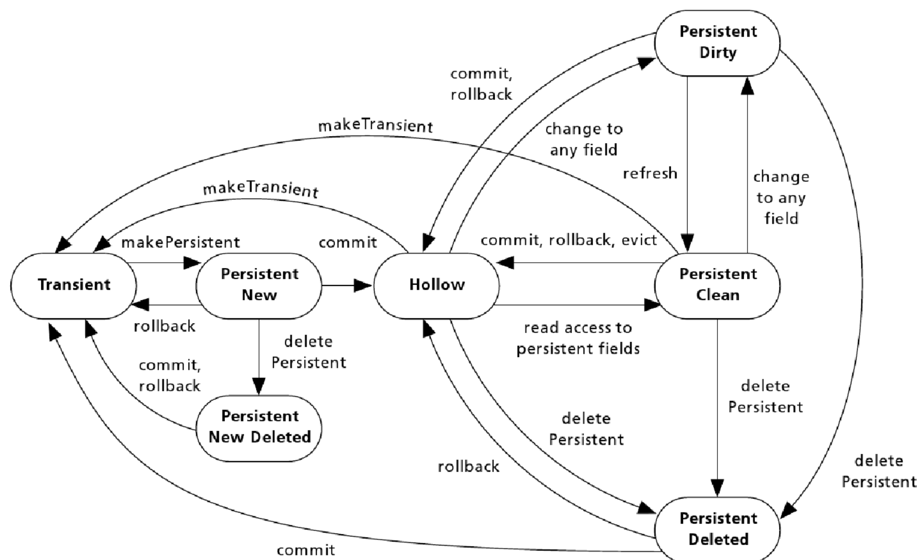
- *Persistent-Clean* – pokud instance reprezentuje data v databázi, má JDO identitu i načtené hodnoty, ale nedošlo k jejich změně, pak je ve stavu *Persistent-Clean*.

- *Persistent-Dirty* – ve stavu *Persistent-Dirty* se nachází instance, které v rámci transakce načetly a změnily své hodnoty. Další možností, jak objekt do tohoto stavu dostat, je zavolání metody `makeDirty`, kterou poskytuje třída `JDOHelper`. Tento postup je vhodný v případě, že došlo ke změnám některých položek v poli, protože JDO monitoruje pole pouze jako celek.

- *Persistent-Deleted* – do stavu *Persistent-Deleted* se dostanou instance, které načetly hodnoty svých atributů z databáze a byly v rámci aktuální transakce odstraněny.

Pokud však byly zároveň ve stejné transakci vytvořeny, nebude jejich stav Persistent-Deleted, ale Persistent-New-Deleted.

Přechody mezi jednotlivými stavy jsou znázorněny na obrázku 4.6.



Obrázek 4.6: Diagram přechodů mezi jednotlivými stavy instancí. Převzato z [13]

4.7 Transakce

Uživatelské akce týkající se perzistence objektů mohou být v JDO prostředí prováděny pomocí transakcí, které mají standardní vlastnosti: atomičnost, konzistenci, izolovanost a trvanlivost. Základní smyslem transakčního zpracování je zajištění, že se požadované operace provedou všechny nebo ani jedna. Tímto se dosahuje konzistentního stavu systému.

Každý `PersistenceManager` obsahuje jeden objekt `javax.jdo.Transaction`, což je právě rozhraní pro práci s transakcemi. V konkrétním okamžiku může probíhat pouze jedna transkace, jejíž objekt lze získat pomocí `PersistenceManager` objektu a jeho metody `currentTransaction`. Pokud je v programu více instancí `PersistenceManager`, je možné provádět paralelně odpovídající počet operací.

JDO rozlišuje dvě strategie provádění transakcí. Každá implementace povinně obsahuje podporu tzv. *pesimistických transakcí* a volitelně i *optimistických transakcí* [13].

- *Pesimistické transakce* – charakteristickým znakem pesimistických transkací je jejich krátká životní doba, během které se nepředpokládá žádná blokující činnost. Pokud dojde ke změnám v databázi, je ostatním transakcím odepřen přístup k těmto datům, dokud není původní transkace dokončena.

- *Optimistické transakce* – v situacích, kdy je nevhodné během transakce data zamykat, přichází na pomoc transakce optimistické. Typickým příkladem je čekání na uživatelský vstup během transakčního zpracování, kdy je zatím možné s daty pracovat. Přívlastek „optimistický“ poukazuje na fakt implicitního předpokladu, že nedojde k dalším modifikacím stejných dat z jiných transakcí. Toto bývá ve většině případů

splněno, ale zajisté ne vždy. Proto se před skutečným zápisem změn do databáze tento předpoklad ještě ověřuje.

Na obrázku 4.7 je stručný popis metod, které jsou definovány v rozhraní `Transaction`.

- `begin` – zahájení transkace.
- `commit` – úspěšné ukončení transakce, potvrzení provedených změn.
- `rollback` – neúspěšné ukončení transakce, zrušení provedených změn.
- `isActive` – vrací `true`, pokud jde o právě probíhající transakci.
- `setNontransactionalRead / getNontransactionalRead` – nastavení / získání příznaku, který určuje, zda je možné číst atributy perzistentních objektů ve stavu *Hollow* nebo *Non-transactional* mimo transakci.
- `setNontransactionalWrite / getNontransactionalWrite` – nastavení / získání příznaku, který určuje, zda je možné zapisovat atributy perzistentních objektů ve stavu *Persistent-Nontransactional* mimo transakci.
- `setRetainValues / getRetainValues` – nastavení / získání příznaku ovlivňujícího výkon při úspěšném ukončení transakce. Pokud bude tento příznak nastaven na `false`, budou instance automaticky odstraněny z cache. V případě `true` však všechna související data zůstanou v paměti pro další použití.
- `setRestoreValues / getRestoreValues` – nastavení / získání příznaku ovlivňujícího výkon při neúspěšném ukončení transakce. Pokud bude tento příznak nastaven na `true`, budou atributy načteny z cache, aby měly své původní hodnoty. V případě `false` však bude celý objekt převeden do stavu *Hollow*, což znamená, že není potřeba data obnovovat.
- `setOptimistic / getOptimistic` – nastavení / získání příznaku určujícího, zda se jedná o optimistickou nebo pesimistickou transakci.
- `setSynchronization / getSynchronization` – nastavení / získání objektu třídy definující rozhraní `javax.transaction.Synchronization`. Toto rozhraní poskytuje metody `afterCompletion` a `beforeCompletion`, které upozorňují na zahájení nebo ukončení transkace. Díky nim je možné tyto okamžiky zachytit a reagovat na ně.
- `getPersistenceManager` – získání rodičovského objektu `PersistenceManager`.

Obrázek 4.7: Metody rozhraní Transaction

4.8 Mapování do relační databáze

V JDO lze použít i jiná datová úložiště než relační databázi, avšak tato bývá nejčastějším případem. Objekty a jejich atributy je tedy třeba přetransformovat (namapovat) do tabulek a jejich sloupců. Pro každou třídu existuje jedna hlavní a potenciálně několik vedlejších tabulek, kdy jednoduché datové typy mají své protějšky přímo v databázi a ty složené jsou pak rozprostřeny do vlastních tabulek. Definice transformace je uvedena v XML formátu.

Na obrázku 4.8 je uvedena ukázka jednoduché transformace třídy, která má atributy základních datových typů. V tomto případě byla použita JDO implementace *DataNucleus (http://www.datanucleus.org)*.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
3 <orm>
4   <package name="persons">
5     <class name="Person" identity-type="datastore" table="PERSONS">
6       <inheritance strategy="new-table"/>
7       <field name="name">
8         <column name="NAME"/>
9       </field>
10      <field name="age">
11        <column name="AGE"/>
12      </field>
13     <field name="cv">
14        <column name="CV"/>
15      </field>
16    </class>
17  </package>
18 </orm>
```

```
1 package persons;
2 import javax.jdo.annotations.PersistenceCapable;
3 @PersistenceCapable; // anotace pro enhancer
4 public class Person {
5   String name;
6   int age;
7   String cv;
8   // ... (definice get / set metod)
9 }
```

```
mysql> describe PERSONS;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| PERSON_ID | bigint(20)   | NO   | PRI | NULL    |       |
| AGE       | int(11)      | NO   |     | NULL    |       |
| CV        | varchar(256) | YES  |     | NULL    |       |
| NAME      | varchar(256) | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

Obrázek 4.8: Definice Java třídy a její namapování do MySQL databáze.

Na obrázku 4.9 je pak uveden způsob mapování stejné třídy (`Person`) s použitím více tabulek (struktura je na obrázku 4.10), kdy budou životopisy uloženy zvlášť.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
3  <orm>
4    <package name="persons">
5      <class name="Person" identity-type="datastore" table="PERSONS">
6        <inheritance strategy="new-table"/>
7        <field name="name">
8          <column name="NAME"/>
9        </field>
10       <field name="age">
11         <column name="AGE"/>
12       </field>
13       <join table="CVS" column="PERSON_CV"/>
14       <field name="cv" table="CVS" column="CURICULUM"/>
15     </class>
16   </package>
17 </orm>
```

Obrázek 4.9: Ukázka ORM, kdy je použito více tabulek v databázi.

```
mysql> describe CVS;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| PERSON_CV | bigint(20)   | NO   | PRI | NULL    |       |
| CURICULUM | varchar(256) | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+

mysql> describe PERSONS;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| PERSONS_ID | bigint(20)   | NO   | PRI | NULL    |       |
| AGE        | int(11)      | NO   |     | NULL    |       |
| NAME       | varchar(256) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

Obrázek 4.10: Struktura databáze při mapování třídy do více tabulek.

4.8.1 Mapování vazeb

Mají-li objekty mezi sebou nějaký vztah, říkáme, že jsou ve vazbě. Lze rozlišovat varianty *N:1*, *1:N* a *M:N*. Pro mapování vztahů, které jsou v relační databázi vyjádřeny pomocí cizích klíčů, se v Javě používá buď jednoduchý atribut základního datového typu (v případě jednoduché vazby), pole nebo jeden z množinových datových typů implementujících rozhraní `Collection` či `Map`. V případě oboustranné vazby stačí, když bude namapován alespoň jeden směr vztahu [14].

Pro mapování jednoduché vazby se využívá jedna ze tří následujících strategií. Odkazující instance se mapuje do tabulek, kterým se říká *hlavní* a případně *vedlejší*.

- *Serializace* – celá odkazovaná instance je pomocí serializace převedena na proud dat, který je uložen do jednoho sloupce hlavní nebo vedlejší tabulky.
- *Vložení* – všechny atributy odkazované instance jsou přidány jako další sloupce do hlavní nebo vedlejší tabulky odkazujícího objektu.
- *Odkaz* – pro odkazovanou instanci se vytvoří vlastní tabulka, která je svázána s hlavní nebo vedlejší tabulkou pomocí cizího klíče.

V případě mapování vícenásobné vazby je k dispozici pět strategií.

- *Serializace* – celá odkazovaná množina je pomocí serializace převedena na proud dat, který je uložen do jednoho sloupce hlavní nebo vedlejší tabulky.
- *Serializace s využitím spojovací tabulky* – obdoba výše uvedené serializace s tím rozdílem, že se vytvoří speciální tabulka, která obsahuje serializované záznamy (jeden sloupec) a cizí klíče do hlavní nebo vedlejší tabulky.
- *Vložení do spojovací tabulky* – celá odkazovaná množina je převedena na záznamy ve speciální tabulce, kde je pro každý atribut samostatný sloupec. Provázání s hlavní nebo vedlejší tabulkou je opět pomocí cizího klíče.
- *Odkaz do hlavní tabulky* – odkazovaná instance obsahuje cizí klíč do hlavní tabulky.
- *Odkazy ve vazební tabulce* – vytvoří se vazební tabulka, která má dva sloupce. Prvním je odkaz na hlavní nebo vedlejší tabulku odkazující instance a druhým je odkaz na hlavní nebo vedlejší tabulku odkazované instance.

Na obrázku 4.11 je uvedena ukázka mapování vazby N-1. Uvažujme třídu `Person`, která má atributy `String name` a `Job job`. Dále uvažujme třídu `Job` s jedním atributem `String jobname`. Vztah je takový, že každý osoba (`Person`) má právě jedno zaměstnání `Job`. Stejné zaměstnání ale může mít libovolný počet osob.

Na obrázku 4.12 je uvedena ukázka mapování vazby 1-N. Uvažujme třídu `Person`, která má jediný atribut `String name` a třídu `Job` s atributy `String jobname` a `List<Person> persons`. Vztah je takový, že každé zaměstnání (`Job`) může vykonávat více osob (`Person`). Jde o opačnou vazbu, než je uvedena na obrázku 4.11.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
3 <orm>
4   <package name="persons">
5     <class name="Person" identity-type="datastore" table="PERSONS">
6       <field name="name">
7         <column name="NAME"/>
8       </field>
9       <field name="job">
10          <column name="JOB_ID"/>
11      </field>
12    </class>
13    <class name="Job" identity-type="datastore" table="JOBS">
14      <field name="jobname">
15        <column name="JOBNAME"/>
16      </field>
17    </class>
18  </package>
19 </orm>
```

Obrázek 4.11: Mapování vazby N-1.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
3 <orm>
4   <package name="persons">
5     <class name="Job" identity-type="datastore" table="jobs">
6       <field name="jobname">
7         <column name="JOBNAME"/>
8       </field>
9       <field name="persons" persistence-modifier="persistent">
10        <collection element-type="persons.Person"/>
11        <element column="JOB_ID"/>
12      </field>
13    </class>
14    <class name="Person" identity-type="datastore" table="persons">
15      <field name="name">
16        <column name="NAME"/>
17      </field>
18    </class>
19  </package>
20 </orm>
```

Obrázek 4.12: Mapování vazby 1-N.

4.8.2 Verzování a dědičnost

JDO podporuje tzv. *verzování* záznamů, kdy je možné detekovat změny v databázi na základě *porovnání stavů*, *časových razítek* či *verzovacích čísel*. Při každé editaci záznamu je verzování aktualizováno.

Pokud je požadováno mapování tříd včetně svých předků, zvolí se jedna ze tří strategií pro dědičnost.

- *Nová tabulka* – atributy jsou uloženy do vlastní tabulky.
- *Tabulka nadtřídy* – atributy jsou uloženy do tabulky předka.
- *Tabulka podtřídy* – atributy jsou mapovány do tabulky třídy.

Na obrázku 4.13 je uvedena ukázka mapování dědičnosti (strategie s novou tabulkou) s využitím verzování (strategie s číslem verze). Uvažujme základní třídu `Person` s atributem `name` a odvozenou třídu `Employee`, která přidává atribut `job`.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
3  <orm>
4    <package name="persons">
5      <class name="Person" table="persons">
6        <inheritance strategy="new-table">
7          <discriminator strategy="class-name" column="JAVA_CLASS"/>
8        </inheritance>
9        <version strategy="version-number"/>
10       <field name="name">
11         <column name="NAME"/>
12       </field>
13     </class>
14     <class name="Employee" table="employees">
15       <inheritance strategy="new-table">
16         <join column="PERSON_ID"/>
17       </inheritance>
18       <field name="job">
19         <column name="JOB"/>
20       </field>
21     </class>
22   </package>
23 </orm>
```

Obrázek 4.13: Mapování dědičnosti (nová tabulka) s využitím verzování.

4.9 JDOQL

JDO Query Language (JDOQL) je objektově orientovaný dotazovací jazyk syntakticky podobný Javě. Největší výhoda jeho použití plyne z nezávislosti na datovém úložišti nižší vrstvy, kdy překlad do konkrétního dialektu SQL řeší JDO implementace. V případě ukládání do datových souborů ani žádný SQL jazyk není k dispozici, ale přesto je možné JDOQL používat.

Základním konceptem je tzv. *kolekce kandidátů*, což je množina výsledků dotazu, na jejíž prvky se aplikuje filtr. Samotné dotazy jsou parametrizovatelné a skládají se z následujících částí.

- *Kandidátní třída* – všechny prvky ve výsledné množině jsou objekty právě této třídy.
- *Kolekce kandidátů* – množina výsledků. Může jít přímo o třídu implementující rozhraní `Collection` nebo o extent kandidátní třídy.
- *Filtr pro dotazy* – jde o pravdivostní výraz, který se aplikuje při dotazu. Výsledné instance jsou právě ty, které zadanou podmínku splňují.
- *Deklarace parametrů* – v této části se deklarují parametry ve formátu *identifikátor* a *datový typ*.
- *Hodnoty parametrů* – přiřazení hodnot k deklarovaným parametrům.
- *Deklarace proměnných* – obdoba deklarace parametrů. Proměnné však slouží pro vnitřní potřeby dotazu.
- *Deklarace externích tříd* – tyto třídy mohou být použity pro parametry nebo proměnné.
- *Způsob řazení* – volitelně je možné specifikovat, jak se mají výsledné prvky sežadit.
- *Jmenné prostory* – slouží k přehlednější práci s identifikátory. Rozlišuje se jmenný prostor pro datové typy a pro identifikátory parametrů, proměnných či atributů.

4.10 Dotazování

Pro vytváření databázových dotazů a získávání výsledků slouží v JDO rozhraní `Query`. Instance dotazů se vytváří pomocí továrních metod specifikovaných v rozhraní `PersistenceManager`. Je možné nastavit požadovanou třídu, extent, kolekci kandidátních objektů či způsob řazení. Dále je umožněno definovat vlastní proměnné nebo parametry, které budou doplněny při provedení dotazu. Hlavní vlastností je ale definice vlastního dotazu v jazyce JDOQL, který byl popsán v předchozí sekci.

Na obrázku 4.14 jsou stručně popsány všechny veřejné metody, které umožňují práci s dotazy [13].

- `getPersistenceManager` – vrací instanci `PersistenceManager`a, se kterým je dotaz spojen.
- `setClass` – nastavuje kandidátní třídu.
- `setCandidates` – dvě verze pro nastavení kandidátní kolekce.
- `setFilter` – umožňuje specifikovat filtr.
- `declareImports` – deklarace externích tříd.
- `declareVariables` – deklarace proměnných.
- `declareParameters` – deklarace parametrů.
- `setOrdering` – nastavuje způsob řazení výsledků.
- `setIgnoreCache` – nastavuje příznak ovlivňující práci s vyrovnávací pamětí. Pokud má hodnotu `true`, neuvažují se změny, které jsou pouze ve vyrovnávací paměti a nejsou ještě v datovém úložišti.
- `getIgnoreCache` – vrací příznak ovlivňující práce s vyrovnávací pamětí.
- `compile` – kompilace dotazu. Pro provedení dotazu není kompilace nutná, ale po jejím provedení jsou známy případné syntaktické chyby a vlastní provedení dotazů je následně rychlejší.
- `execute` – čtyři varianty pro provedení dotazu.
- `executeWithArray` – provedení dotazu se specifikovanými parametry ve formě pole.
- `executeWithMap` – provedení dotazu se specifikovanými parametry ve formě datového typu implementujícího rozhraní `Map`.
- `close` – uzavření jednoho zadaného zdroje spojeného s dotazem.
- `closeAll` – uzavření všech zdrojů spojených s dotazem.

Obrázek 4.14: Metody rozhraní Query

Na obrázku 4.15 je ukázka jednoduchého dotazu, který vrátí osoby s věkem vyšším než 30 let.

```
1 package persons;
2
3 import javax.jdo.*;
4 import java.util.*;
5
6 public class Main {
7   public static void main(String[] args) {
8     PersistenceManagerFactory pmf = (PersistenceManagerFactory)
9       JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
10    PersistenceManager pm = pmf.getPersistenceManager();
11    pm.currentTransaction().begin();
12
13    Person petr = new Person("Petr", 42);
14    pm.makePersistent(petr);
15    Person josef = new Person("Josef", 24);
16    pm.makePersistent(josef);
17    Person david = new Person("David", 32);
18    pm.makePersistent(david);
19
20    Extent personsExtent = pm.getExtent(Person.class, true);
21    String filter = "age > 30";
22    Query query = pm.newQuery(personsExtent, filter);
23    query.setOrdering("age ascending");
24    Collection result = (Collection) query.execute();
25    Iterator iterator = result.iterator();
26
27    System.out.println("Osoby nad 30 let: ");
28
29    while (iterator.hasNext()) {
30      Person person = (Person) iterator.next();
31      System.out.println(person.getName() + " (" +
32        person.getAge() + ")");
33    }
34
35    query.close(result);
36    pm.currentTransaction().commit();
37    pm.close();
38    pmf.close();
39  }
40 } // class
```

Obrázek 4.15: Ukázka jednoduchého dotazu.

Kapitola 5

Návrh temporálního rozšíření

Tato kapitola otevírá prakticky zaměřenou část práce. Výsledným produktem je systém tříd a rozhraní realizujících temporální rozšíření standardu JDO [14]. V následujících sekcích budou uvedeny hlavní požadavky na tento systém a nastíněny postupy, jak jich docílit. Dále budou popsány nejdůležitější třídy včetně jejich rolí v systému a nakonec budou uvedeny UML diagramy znázorňující architekturu a jednotlivé třídy systému.

5.1 Vlastnosti a návrh jejich realizace

V této sekci budou popsány hlavní požadavky na funkcionalitu výsledného rozšíření. U každé položky (v každé podsekci) bude naznačen způsob její realizace tak, jak byl navržen v počátečních fázích vývoje.

5.1.1 Ukládání temporálních dat

Ukládání dat včetně jejich časových aspektů představuje základní požadavek navrhovaného systému. Smyslem je umožnit uživateli uchovávat historii jednotlivých objektů s co možná nejmenším úsilím.

Srdcem celého rozšíření je rozhraní `TemporalPersistenceManager`, které představuje temporální variantu JDO rozhraní `PersistenceManager`. Jsou zde využity návrhové vzory *Adapter* a *Factory*. Instance TemporalPersistenceManager poskytuje upravené verze většiny operací deklarovaných v rozhraní PersistenceManager, ale některé z nich jsou úmyslně vypuštěny. Jde například o fyzické mazání z databáze, neboť operace tohoto druhu jsou v temporálních databázích řešeny jinak. Dalším účelem třídy implementující toto rozhraní je pak vytváření instancí realizujících databázové dotazy. Tato problematika bude blíže vysvětlena v kontextu temporálního dotazování v následujících podsekcích.

Aby bylo možné se všemi temporálními objekty pracovat jednotným způsobem, byla navržena základní datová třída `TemporalBase`, kterou bude rozšiřovat každý uživatelský temporální objekt. Obsahuje atributy a metody pro jednoznačnou identifikaci objektu (k tomu je mimo jiné využita třída `TemporalCounter`), číslo temporální verze, příznaky aktuálnosti nebo smazání a samozřejmě intervaly platnosti i transakčního zpracování.

Proces temporálního uložení objektu je dále popsán na straně 50 a znázorněn v diagramu na obrázku 5.6 na straně 51.

5.1.2 Bitemporální model dat

Systém je navržen jako bitemporální, což znamená, že podporuje intervaly časů platnosti a transakce. Uložení potřebných metadat je realizováno jako dvojice atributů třídy `TemporalIntervalList` obsažených v základu (TemporalBase) každé instance. Vnitřně jde o seznam intervalů reprezentovaných třídou `TemporalInterval`. Tato třída poskytuje metody pro porovnávání, množinové operace (průnik a sjednocení) a ověření platnosti zadaných okrajových hodnot intervalu (počátek musí předcházet konci nebo s ním být shodný). Čas platnosti může být rozdělen do několika intervalů, které nemají společný průnik. Všechny tyto jednotlivé intervaly jsou pak zapouzdřeny a spravovány právě třídou `TemporalIntervalList`. Transakční čas prakticky nenabývá podoby více intervalů, ale z důvodů jednotného zpracování temporálních metadat je také implementován jako `TemporalIntervalList`.

5.1.3 Dotazování temporálních dat

Dotazování temporálních dat je hned po jejich ukládání druhou nejdůležitější vlastností navrhovaného systému. Umožňuje uživateli získat libovolný předem uložený objekt v požadovaném časovém kontextu. Dále je možné z databáze načíst celou historii zadané instance. Výsledkem je kolekce, kdy každá položka charakterizuje další provedenou změnu. Způsob zadávání dotazů je obdobný tomu z klasického JDO. Navíc je však možné zadávat intervaly platnosti nebo transakce, které řeší temporální aspekty dotazu. Bez jejich použití jsou vráceny buď aktuální verze instancí (netemporální dotaz) nebo celá historie všech odpovídajících instancí (temporální dotaz).

Při dotazech bude uživatel pracovat s objekty třídy `TemporalQuery`, která představuje temporální alternativu JDO rozhraní `Query`. Tato vnitřně využívá instance třídy `TemporalFilter`, ve které se realizuje převod temporálního dotazu na dotaz klasický.

Posloupnost úkonů při dotazech je blíže popsána na straně 52. Na obrázku 5.7 na straně 52 je potom naznačena spolupráce objektů, které se dotazu účastní.

5.1.4 Temporální mazání

Mazání v temporální databázi v podstatě znamená nastavení příznaku smazání u zadaných instancí. Díky hodnotě tohoto příznaku je pak s objektem pracováno příslušným způsobem. Atribut realizující tento příznak je obsažen v základu každé temporální instance (`TemporalBase`).

5.1.5 Fyzické mazání

Fyzické mazání neboli „odsávání“ je skutečné odstranění záznamů z databáze. Z hlediska principů temporální databáze je tato operace zbytečná, ale v praxi se používá pro uvolnění prostoru na disku (obecně v databázi) v případě, že některá data již pro účely aplikace nejsou potřeba a zabírají tedy zbytečně paměťové médium.

Tato funkcionalita je implementována v metodě rozhraní `TemporalPersistenceManager`, která je parametrizována časovým okamžikem. Ten představuje hranici, po kterou budou vybírány objekty ke smazání. Celá tato operace pracuje pouze s transakčním časem, platnost odstraňovaných dat zde nehraje žádnou roli. Smazány jsou vždy pouze neaktuální verze objektů, které představují needitovatelnou historii, takže nedojde k porušení referenční integrity, která je udržována na úrovni aktuálních instancí.

5.1.6 Temporální referenční integrita

Temporální referenční integrita představuje z hlediska návrhu nejnáročnější vlastnost systému. Jde o zajištění takového stavu databáze, aby nevznikla situace, kdy existuje reference (odkaz) na neexistující položku, a to včetně uvažování intervalů platnosti. Způsob dosažení tohoto požadavku je založen na implementaci třídy `TemporalIntegrity`, která má za úkol při každé databázové operaci měnící stav uložených záznamů projít a ověřit ukládaná nebo mazaná data.

Aby bylo možné kontrolovat jednotlivé reference mezi objekty, dochází k ukládání všech vazeb v podobě dvojice *(zdroj, cíl)* do databáze, odkud jsou později při kontrole načítány. V návrhu jsou tyto dvojice reprezentovány objektem třídy `TemporalBind`.

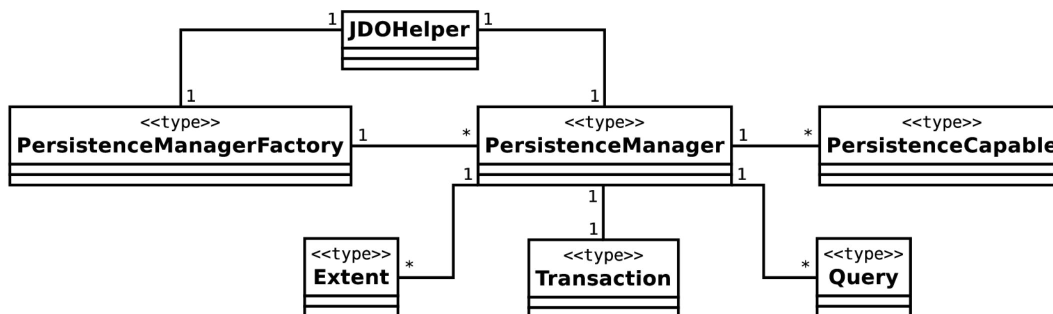
5.2 UML diagramy

V této sekci budou nejprve uvedeny diagramy znázorňující důležité třídy návrhu včetně jejich provázanosti. Kompletní diagram tříd je uveden v příloze na obrázku B.1 na straně 74. Zde bude z důvodu čitelnosti rozdělen do několika menších částí, které budou postupně okomentovány.

V dalších podsekcích budou rozebrány složitější procesy – ukládání temporální instance a provedení temporálního dotazu.

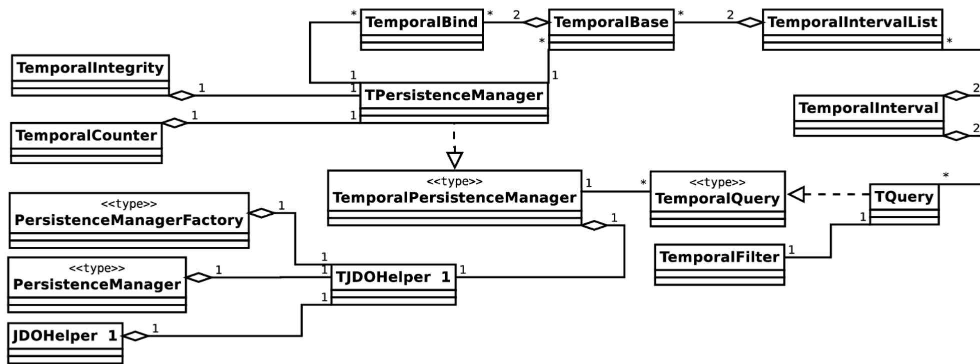
5.2.1 Architektura

Na obrázku 5.1 je uveden přehled jádra JDO. Základ je tvořen rozhraními `PersistenceManagerFactory` a `PersistenceManager`. Uvedené rozhraní `PersistenceCapable` implementují všechny ukládané objekty.



Obrázek 5.1: Přehled jádra architektury JDO.

Na obrázku 5.2 je uvedeno jádro navrhovaného temporálního rozšíření JDO. Základ tvoří třída `TemporalPersistenceManager`, která představuje temporální alternativu k rozhraní `PersistenceManager`.

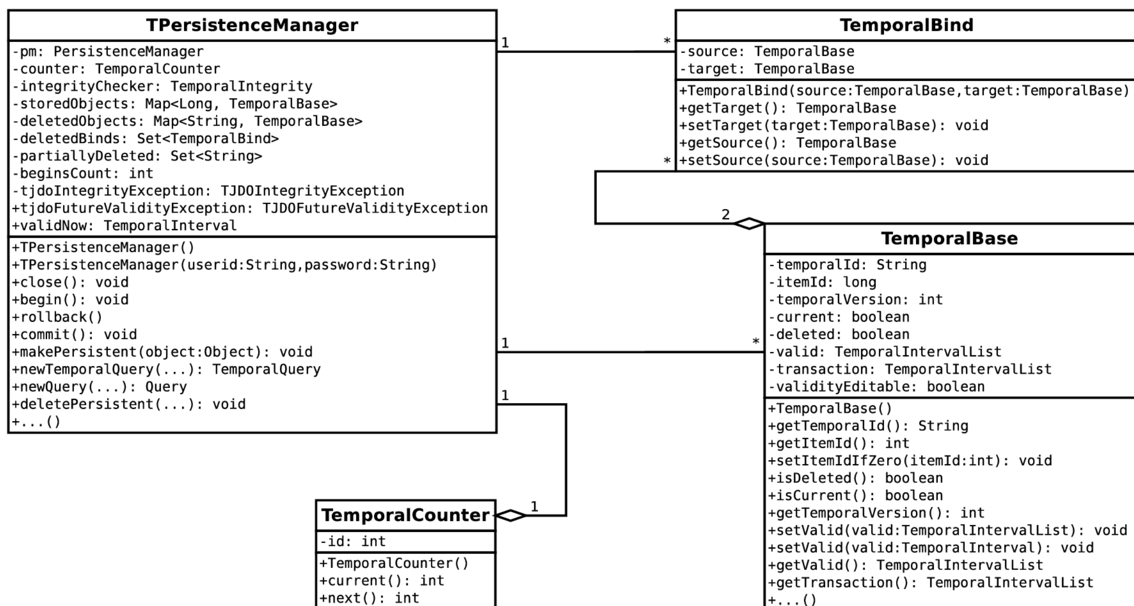


Obrázek 5.2: Jádro navrhovaného temporálního rozšíření JDO.

5.2.2 Diagramy tříd

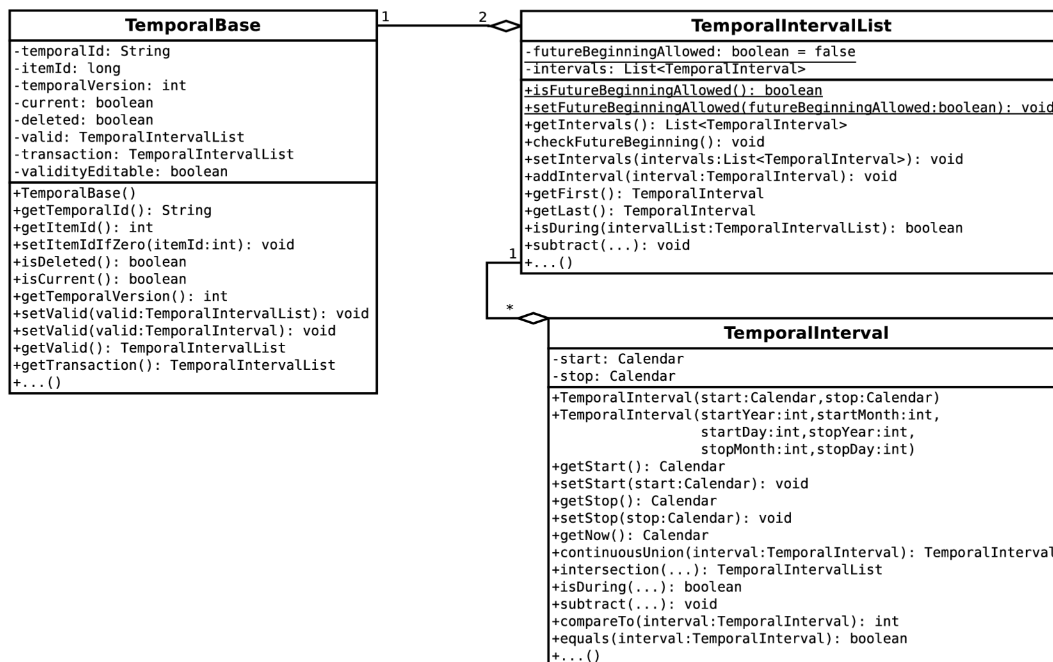
Na následujících diagramech bude rozebrán návrh tříd včetně jejich nejdůležitějších metod a atributů. Z důvodů zachování přehlednosti však nebudou uvedeny všechny. Úplný seznam metod a atributů je možné nalézt v programátorské dokumentaci.

Diagram na obrázku 5.3 ukazuje vztahy mezi temporálním správcem a datovými objekty. Pro generování identifikátorů `itemId` slouží třída `TemporalCounter`. Jedna instance `TPersistenceManager` může spravovat libovolný počet temporálních entit (`TemporalBase`) a jejich případných vazeb (`TemporalBind`). Každá vazba potom obsahuje právě dvě reference na datové objekty – zdroj a cíl vztahu.



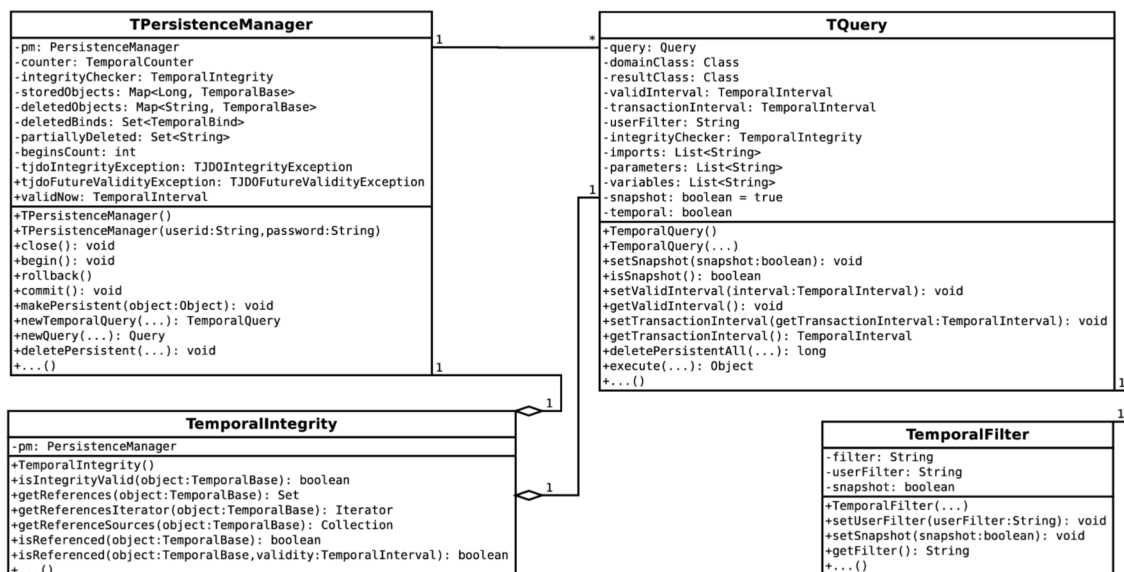
Obrázek 5.3: Část diagramu tříd týkající se datových objektů.

Obrázek 5.4 znázorňuje, jakým způsobem jsou implementovány intervaly transakce a platnosti. V obou případech se jedná o seznam jednoduchých intervalů (`TemporalInterval`), který je zapouzdřen do třídy `TemporalIntervalList`.



Obrázek 5.4: Část diagramu tříd týkající se intervalů u datových objektů.

Poslední část diagramu tříd (obrázek 5.5) zobrazuje vztahy potřebné pro provádění temporálních dotazů. Každý správce (`TPersistenceManager`) disponuje sadou metod pro vytváření instancí `TQuery`. Obě tyto metody vnitřně využívají prostředek pro kontrolu referenční integrity (`TemporalIntegrity`). Součástí každého dotazu je právě jedna instance temporálního filtru (`TemporalFilter`).



Obrázek 5.5: Část diagramu tříd týkající se temporálních dotazů.

5.2.3 Ukládání temporální instance

Na obrázku 5.6 je znázorněn průběh procesu uložení nových nebo pozměněných objektů. Zpracovávají se pouze instance tříd, které dědí temporální základ `TemporalBase`. Ostatní objekty, pokud jsou takové pod správou temporálního manažeru, se tohoto procesu neúčastní. Jsou ale zpracovány klasickým manažerem a do databáze se tedy uloží. Pro načtení však bude potřeba využít klasické JDO rozhraní, protože TJDO předpokládá u všech entit datový základ `TemporalBase`.

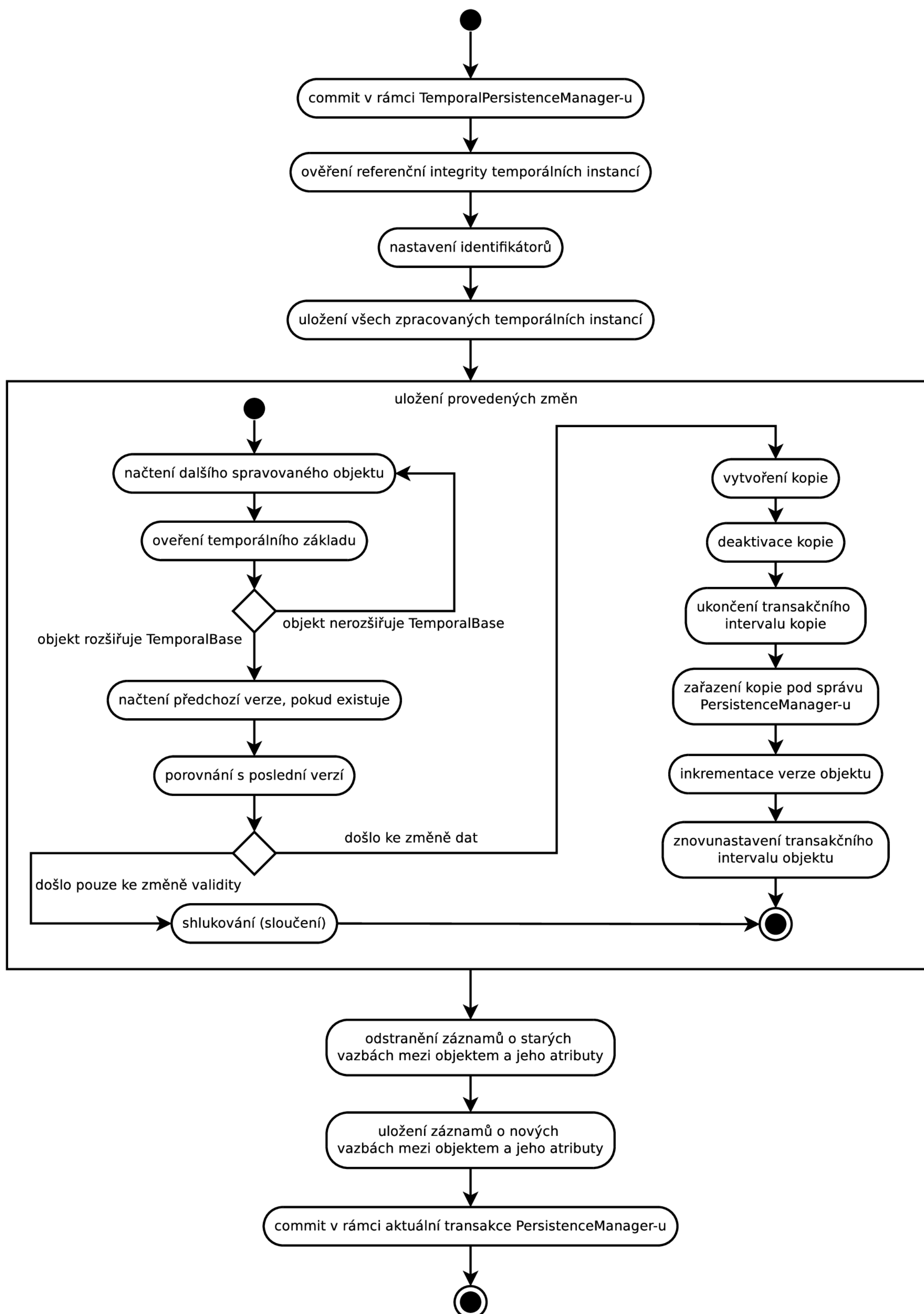
Volání metody `commit` třídy implementující rozhraní `TemporalPersistenceManager` spustí sadu kontrol a úkonů, které ověří požadované vlastnosti ukládané instance a nastaví hodnoty atributů pro vnitřní potřeby systému. Celá operace je rozložena do dvou transakcí.

Během první transakce dojde k ověření referenční integrity, kdy jsou z databáze načteny a ověřeny všechny vazby vztahující se k právě kontrolovanému objektu, a nastavení temporálních identifikátorů, které jsou automaticky generovány systémem a slouží k jednoznačnému rozlišení i v rámci historie stejné instance. Všechny takto zpracované objekty jsou uloženy pro následné zpracování ve druhé transakci.

Důvodem rozdělení celé operace do dvou transakcí je získání referencí na všechny spravované objekty v rámci zpracování JDO událostí *preStore* a *postStore*, které nastávají až během ukončování transakce na úrovni `PersistenceManager`.

K vlastnímu uložení temporálních změn tedy dojde až ve druhé transakci. V případě, že se zpracovává objekt bez temporálního základu, je tento okamžitě přeskočen, jak už bylo uvedeno výše. Následuje načtení předchozí verze objektu, pokud tato v databázi existuje, a její porovnání s aktuálně zpracovávanou instancí. Pokud jsou změny pouze na úrovni času platnosti, dojde ke sloučení, kdy je starší verze upravena pouze z hlediska platnosti verze aktuální. Ve všech ostatních případech (tedy změna dat nebo neexistující předchozí verze) dojde k vytvoření kopie, která reprezentuje poslední ukončený stav instance. Nastaví se konec transakčního intervalu a zařadí se pod správu manažera, aby došlo k jejímu uložení. Originální objekt nyní představuje nejnovější stav v rámci své historie, a proto dojde k inkrementaci verze a znovunastavení transakčního času (na aktuální čas).

Nakonec je ještě potřeba upravit záznamy o vazbách. Odstraní se všechny staré a vytvoří se nové. Posledním krokem je volání metody `commit` rozhraní `PersistenceManager`, což je klasické potvrzení transakce v JDO. Tím je ukončena druhá transakce a vlastně i celý proces uložení.

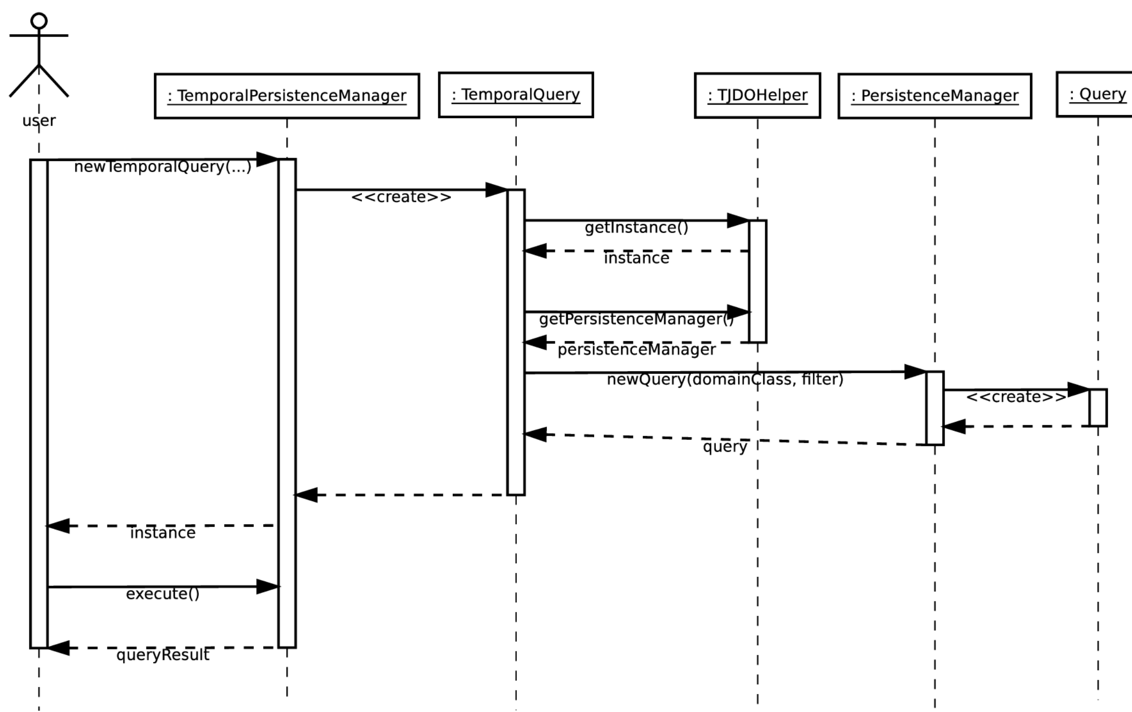


Obrázek 5.6: Diagram aktivity znázorňující uložení temporální instance.

5.2.4 Provedení dotazu

Sekvenční diagram na obrázku 5.7 znázorňuje průběh vytvoření a provedení jednoduchého dotazu. U složitějších dotazů se navíc specifikují další parametry jako např. interval platnosti, interval transakce nebo deklarace proměnných a objektů využitých v dotazu.

Každý temporální dotaz začíná požadavkem uživatele na vytvoření instance třídy implementující rozhraní `TemporalQuery` voláním metody `newTemporalQuery` s patřičnými parametry. Na pozadí proběhne komunikace mezi několika v diagramu zobrazenými objekty, jejímž cílem je vytvoření instance třídy implementující rozhraní `Query`, která bude nastavena tak, aby vykonala požadovaný temporální dotaz. Nejprve je potřeba přes globální prostředí (`TJDOHelper`) získat referenci na vnitřní instanci `PersistenceManager`. Poté je možné zavolat tovární (návrhový vzor Factory) metodu `newQuery` a vytvořit tak klasický JDO dotaz. Ten je v rámci `TemporalQuery` rozšířen o temporální aspekty a vrácen uživateli. Pro vykonání dotazu je nutné zavolat metodu `execute`, která až v této chvíli převede zadaný dotaz do temporální podoby a ten okamžitě odešle do databáze. Získaný výsledek je v podobě kolekce vrácen uživateli.



Obrázek 5.7: Sekvenční diagram znázorňující proces vytvoření a provedení dotazu.

Kapitola 6

Implementace

V této sekci budou přiblíženy a popsány jednotlivé vlastnosti a postupy použité během vývoje *Temporálního rozšíření Java Data Objects (dále jen TJDO)*. Výsledným produktem je knihovna pro programovací jazyk *Java*, která vnitřně využívá a rozšiřuje vlastnosti JDO specifikace. Jako referenční JDO implementace byla použita objektová databáze *ObjectDB*, na které byl celý systém vyvíjen a testován.

6.1 Základní vlastnosti

Celý systém je od začátku navržen jako *bitemporální*, což znamená, že poskytuje uživateli z časového hlediska dvojí náhled na ukládaná data. Prvním z nich je čas transakce, který říká, kdy byl objekt v daném stavu do databáze vložen a případně nahrazen novější verzí. Druhým je potom čas platnosti. Ten uchovává informace o tom, kdy jsou data platná v modelovaném světě. Výsledný systém implementuje všechny vlastnosti uvedené v sekci 5.1 na straně 45. V následujících odstavcích budou některé z nich detailněji rozbrány z implementačního hlediska.

6.1.1 Bázová třída TemporalBase

U každé temporální instance se předpokládá, že bude rozšířovat základní bázovou třídu `TemporalBase`, která implementuje všechny požadované temporální atributy a metody, které jsou vnitřně v systému využívány. Jednotlivé atributy jsou uvedeny a vysvětleny v následujícím přehledu.

> `String temporalId` – Unikátní identifikátor sestávající z `itemId` a `temporalVersion`. Například u druhé temporální verze v historii instance 8 je `temporalId` rovno *8v1* (indexováno od nuly). Hodnota tohoto atributu slouží jako jednoznačný primární klíč všech odvozených entit. Z důvodů neustálých kopií objektů není možné používat vlastní primární klíče. Tato problematika bude dále rozebrána v samostatné podsekci.
>
> `long itemId` – Identifikátor všech temporálních verzí jedné instance. V ralační databázi by hodnota tohoto atributu odpovídala klasickému identifikátoru, který je v dané tabulce zcela unikátní. Zde má však stejnou hodnotu hned několik objektů, které dohromady představují všechny změny provedené na jedné konkrétní položce (objektu).
>
> `long temporalVersion` – Identifikátor temporální verze. Nově uložený objekt začíná s hodnotou 0 a při dalších změnách je tato hodnota inkrementována.

`boolean current` – Příznak specifikující aktuálnost temporální verze. Pro danou množinu objektů se stejnou hodnotou `itemId` existuje maximálně jedna instance s hodnotou `current` rovnou `true`.

`boolean deleted` – Příznak určující existenci objektu v relačním pohledu. Hodnota `true` znamená, že byl objekt smazán. V temporální databázi však běžně nedochází k fyzickému odstranění, aby bylo možné provádět dotazy na historii, a proto se pouze nastaví určitý příznak, který tento stav specifikuje. V TJDO je tímto příznakem právě `deleted`.

`TemporalIntervalList valid` – Seznam intervalů platnosti. Jednotlivé intervaly v tomto seznamu jsou realizovány jako uspořádané dvojice bodů na časové ose.

`TemporalIntervalList transaction` – Interval transakčního času. Z důvodů jednotného zpracování společně s `valid` je tento atribut opět realizován jako seznam intervalů. Uživatel však nemůže ručně tento seznam rozšiřovat, a tak `transaction` obsahuje vždy maximálně jeden interval. Jeho hraniční hodnoty představují vložení do databáze a nahrazení novější verzí.

`boolean validityEditable` – Příznak určující, zda je možné upravovat čas platnosti. Pokud je objekt načten z databáze pomocí netemporálního rozhraní `newQuery`, je nastaven na `false`. Metoda `setValid` změnu platnosti neprovede a vyvolá výjimku. V temporálním dotazu `newTemporalQuery` je tento příznak nastaven na `true` a úpravy platnosti jsou tedy povoleny.

Z hlediska funkcionality poskytuje třída `TemporalBase` operace pro získání nebo nastavení jednotlivých atributů, temporální odstranění a vytvoření kopie objektu, čehož se využívá při tvorbě nových temporálních verzí objektu při ukládání změn do datového úložiště. Podrobný přehled všech veřejných metod je dostupný v programátorské dokumentaci.

6.1.2 Ukládání temporálních verzí

Nová temporální verze objektu vznikná při ukládání změn do databáze. Výjimku tvoří případ, kdy nebyly změněny uživatelské atributy, ale pouze platnost objektu, tedy pouze jeden konkrétní atribut temporálního základu uživatelského objektu. Tato situace je blíže popsána v podsekci 6.1.4. Ve všech ostatních případech dochází k vytvoření úplné kopie objektu pomocí serializace a opětovné deserializace. Takto nově vytvořená kopie od daného okamžiku představuje starší verzi zpracovávané instance. Dojde k ukončení transakčního intervalu, kdy koncovým bodem je aktuální časový okamžik, a zneplatnění příznaku `current`. Původní (zdrojový) objekt nyní představuje nejnovější temporální verzi, kdy opět dojde k úpravě transakčního intervalu. Tentokrát je na aktuální časový okamžik nastaven počáteční bod. Koncový bod je beze změny nastaven na nekonečno (v kódu maximální možné datum pro datový typ `java.util.GregorianCalendar`).

Ukládání z hlediska transakčního času

Každý nově vzniklý objekt začíná s transakčním časem nastaveným od aktuálního času po nekonečno. Při jeho změně, kdy dochází k vytvoření kopie a vytvoření nové temporální verze, je konec transakčního času změněn z nekonečna na aktuální čas.

Tuto situaci si ukážeme na příkladu. Uvažujme třídu `Temperature`, která pro jednoduchost uchovává pouze jednu hodnotu, která představuje aktuální teplotu.

Čas provedení	**Akce**	**Obsah databáze – transakční čas**
1.4.2012 10:00	Temperature t = new Temperature(15.3);	[15.3, <1.4.2012 10:00, ∞)]
1.4.2012 12:00	t.setValue(17.1)	[15.3, <1.4.2012 10:00, 1.4.2012 12:00)] [17.3, <1.4.2012 12:00, ∞)]

Aby byl obsah databáze skutečně takový, jaký je uveden, bylo by nutné při editaci ještě upravit čas platnosti. Takto by byla hodnota pouze přepsána a v databázi by byl pouze jeden objekt s poslední hodnotou. V tomto případě ale šlo o pohled z transakčnho času, a proto zde platnost pro jednoduchost neuvažujeme.

Ukládání z hlediska času platnosti

Platnost objektu je implementována jako seznam intervalů. Pokud není explicitně uvedeno jinak, dojde k nastavení platnosti na jeden interval od aktuálního času po nekonečno. Při změně dochází ke kontrole, zda byla upravena pouze platnost a data zůstala beze změny, nebo zda došlo k úpravě uživatelských dat. V prvním případě nedochází k vytvoření kopie, ale pouze se přepíše platnost u již uloženého objektu. V druhém případě však dojde ke kopii objektu a uložení nové temporální verze. Je důležité, aby jednotlivé temporální verze v rámci historie stejné instance neměly průnik ve svých intervalech platnosti. Proto při ukládání nové verze dochází k odečtení celé nejnovější platnosti od všech starších verzí.

Tuto situaci si opět ukážeme na příkladu. Budeme uvažovat stejnou třídu `Temperature`, jako v předchozí podsekci. Tentokrát však uchovávaná hodnota bude znamenat průměrnou teplotu za celý rok. Dále budeme používat třídu `Validity`, která bude představovat seznam intervalů platnosti. V TJDO pro tuto funkcionalitu existuje třída `TemporalInervalList`, ale zde si pro přehlednost vystačíme bez ní.

Čas provedení	**Akce**	**Obsah databáze – čas platnosti**

1.4.2012 10:00	Temperature t =
	new Temperature(7.8);
	Validity v1 =
	new Validity(1990, 1992);
	v1.addInterval(1994, 1996);
	t.setValidity(v1);
		[7.8, <1990, 1992) ∪ <1994, 1996)]

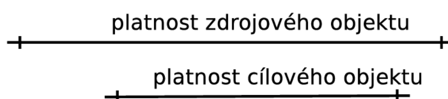
1.4.2012 10:01	t.setValue(8.1);
	Validity v2 =
	new Validity(1991, 1995);
	t.setValidity(v2);
		[7.8, <1990, 1991) ∪ <1995, 1996)]
		
		[8.1, <1991, 1995)]

6.1.3 Kontrola temporální referenční integrity

Dodržování temporální referenční integrity, zjednodušeně řečeno, znamená, že se v datovém úložišti nevyskytuje takový vztah dvou objektů, kdy existuje podinterval platnosti zdrojového objektu, který je mimo platnost cílového objektu. Tato situace je znázorněna na obrázku 6.1. Formálněji by bylo možné tento požadavek zapsat následovně:

$$\forall o1, o2 : bind(o1, o2) \Rightarrow valid(o1) - valid(o2) = \emptyset \quad (6.1)$$

Symboly $o1$ a $o2$ představují temporální objekty, *bind* je binární predikát určující vztah mezi zadanými objekty tak, že jeho první parametr je zdrojový objekt a druhý parametr cílový objekt. Unární predikát *valid* určuje čas platnosti zadaného objektu.



Obrázek 6.1: Porušení temporální referenční integrity

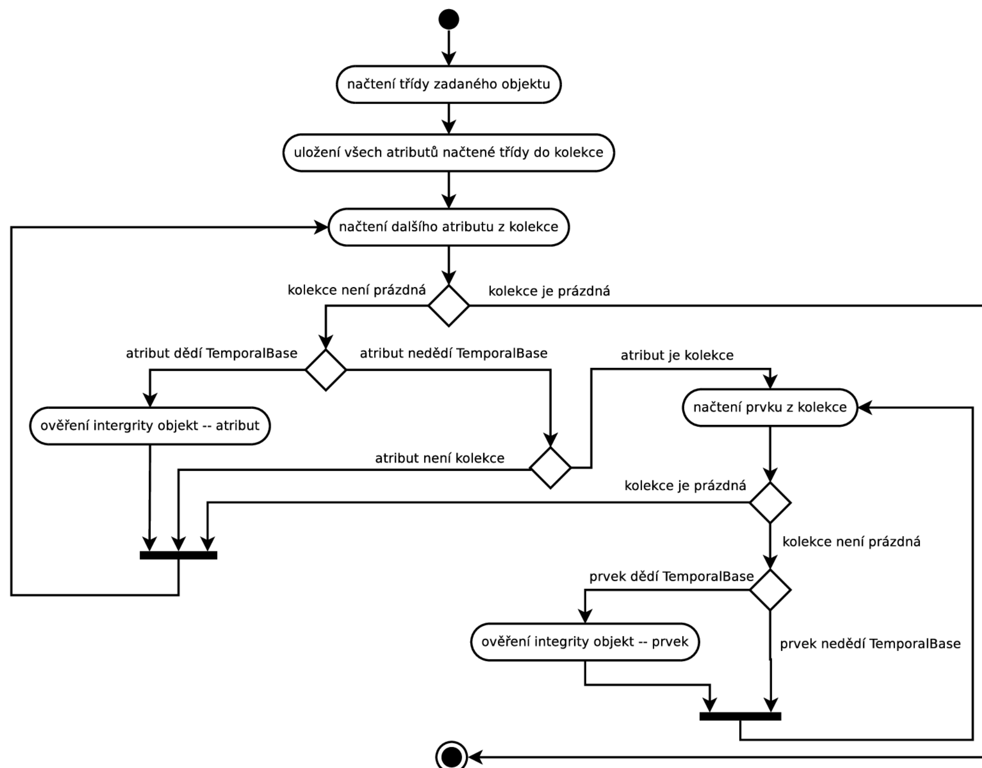
Kontrola referenční intergrity v *TJDO* probíhá vždy během operace ukládání změn do datového úložiště a sestává ze dvou fází. V první fázi se prověří časy platnosti všech temporálních atributů vůči právě ukládanému objektu. Tuto funkcionalitu zajišťuje třída `TemporalIntegrity` a její metoda `boolean isIntegrityValid(TemporalBase object)`. Ta realizuje rekurzivní průchod všemi členskými prvky zadaného objektu, které představují jiné temporální objekty, a ověření jejich časů platnosti. Z tohoto důvodu je nutné, aby pro všechny takovéto atributy byly naimplementovány veřejné metody pro čtení, které jsou pojmenovány `get<NazevAtributu>`. Pro příklad uveďmě situaci, kdy každé auto (třída `Car`) bude mít jako atribut `type` svůj model (třída `Type`). Samotný atribut může být neveřejný, ale musí existovat metoda `public Type getType()`, která tento atribut zpřístupní. V opačném případě nebude referenční integrita tohoto vztahu vůbec uvažována.

Algoritmus průchodu je naznačen na obrázku 6.2. Nejprve se zjistí třída zpracovávaného objektu. Díky ní je poté možné načíst seznam atributů, které budou dále jeden po druhém ověřeny. Pokud se jedná o jednoduchý temporální atribut dědící od třídy `TemporalBase`, je okamžitě provedena kontrola času platnosti zdrojového objektu a tohoto atributu. Pokud se ale jedná o některou z kolekcí, dojde k postupnému ověření všech jejich prvků.

Jednotlivé vazby jsou v *TJDO* ukládány do databáze ve dvojicích *zdroj – cíl*, které jsou zapouzdřeny objektem třídy `TemporalBind`. Ve druhé fázi ověřování referenční temporální integrity se využívá právě těchto vazeb. Z databáze se načtou všechny dříve uložené vazby, které obsahují jako svůj cíl právě ukládaný objekt, a ověří se jeho čas platnosti vůči všem načteným zdrojům. Pokud není z temporálního hlediska zjištěn žádný problematický vztah, dojde k uložení provedených změn včetně aktualizace tabulky vazeb týkajících se právě uložené instance.

6.1.4 Shlukování

Shlukování v TJDO je proces sloučení více temporálních verzí stejné instance v případě, že se tyto liší pouze ve své platnosti. Z datového hlediska jde tedy o shodné objekty a bylo by zbytečné ukládat je do databáze všechny, když je možné nezbytné množství dat redukovat. Ke shlukování dochází při úspěšném ukončení transakce (*commit*), kdy je ověřována platnost nové a předešlé verze dané instance. Pokud se zjistí, že uživatelská data nebyla modifikována, nedojde k vytvoření kopie a uložení nové verze objektu, ale pouze k úpravě platnosti stávající verze. V temporálních databázích může shlukování probíhat i během temporálních dotazů, kdy jsou sloučeny výsledky, jejichž data jsou stejná a liší se opět pouze platností. Tato varianta však není v TJDO implementována.



Obrázek 6.2: Porušení temporální referenční integrity

6.1.5 Temporální dotazování

Temporální dotazy jsou v TJDO založeny na třídě `TemporalFilter`, která se stará o vytvoření upravené verze klasického JDO dotazu. V podstatě jde o specifikaci hodnot atributů `current` a `deleted`. Dále je zde uvedena požadovaná platnost nebo transakční čas. Pomocí tohoto filtru je tedy možné realizovat jak klasické (JDO), tak temporální (TJDO) dotazy. Celá třída je dostupná pouze pro vnitřní potřeby systému a uživatel si tedy nemůže sám vytvářet vlastní instance.

6.1.6 Odsávání

Odsávání v TJDO umožňuje na základě zadaného objektu třídy `java.util.Date` odstranit všechny neaktuální verze temporálních instancí, jejichž čas transakce je ukončen a předchází zadanému parametru. Nedovoluje tedy odstranit nejnovější verze instancí, mezi kterými mohou existovat vazby, a jejichž odstranění by tedy mohlo způsobit problémy v referenční integritě.

6.1.7 Historie dat

Historie konkrétní instance je tvořena jednotlivými verzemi, které vznikají postupně během současných úprav v datech a platnosti objektu. Při každé takové editaci je během ukončení transakce vytvořena kopie objektu, která bude uložena jako z hlediska transakčního času právě ukončená verze. Originální objekt se tak vlastně nikdy nestává historickou verzí, ale je vždy aktuální a dostupný až do doby svého smazání. Všechny temporální verze

jsou přístupné pomocí metod `newTemporalQuery` temporálního správce implementujícího rozhraní `TemporalPersistenceManager`. Takto načtené objekty jsou však hlídány během ukládání změn a jejich případně upravený obsah není brán v potaz. Celá historie je tedy pouze ke čtení. Povolení editace neaktuálních verzí by vedlo k větvení historie, což je v TJDO nežádoucí.

6.2 Způsoby práce s entitami

S datovými objekty (entitami) je možné manipulovat z temporálního hlediska více způsoby, které budou rozebrány v této sekci. Dělení spočívá v temporálním nebo netemporálním režimu práce, způsobu správy jednotlivých verzí objektů nebo dostupnosti v závislosti na času platnosti a aktuálním okamžiku.

TJDO podporuje dva z těchto režimů. Temporální, kde jsou verze spravovány automaticky, a netemporální, kde se verzování neprovádí. V netemporálním režimu jsou v TJDO temporální atributy sice dostupné, ale není povolena jejich editace. Dosahuje se toho pomocí příznaku ve třídě `TemporalBase`, jehož editace je možná pouze v rámci knihovny, takže uživatel nemá možnost toto chování z vnějšku ovlivnit. V temporálním režimu je úprava temporálních atributů automaticky povolena.

6.2.1 Temporální s ručním verzováním

V temporálním režimu s ruční správou verzí je možné libovolně upravovat vlastnosti objektu včetně času platnosti. Pokud je však vyžadováno vytvoření nové verze objektu, je potřeba, aby tak uživatel učinil svépomocí voláním příslušné metody. Ta zajistí vytvoření kopie, inkrementaci verze a úpravu temporálních vlastností. Vazby mezi objekty zůstávají na úrovni starších verzí a jejich přenesení na úroveň aktuálních verzí je případně nutné řešit ručně.

6.2.2 Temporální s automatickým verzováním

Další temporální způsob práce s objekty řeší verzování automaticky. Opět je možné libovolně upravovat vlastnosti objektu včetně času platnosti. Pokud je ale úprava spojena s novou platností, dochází automaticky k vytvoření nové verze objektu. Čísla verzí jsou uživateli běžně k dispozici a může s nimi pracovat, ne je však měnit. Vazby s ostatními entitami zůstávájí na úrovni posledních verzí, takže není nutné, aby se o to uživatel staral sám. Tento mód je v TJDO aktivní v situacích, kde se používá rozhraní `TemporalQuery`.

6.2.3 Netemporální bez verzování

V netemporálním režimu, kde nedochází k vytváření nových verzí objektu, jsou viditelné pouze poslední provedené změny. Platnost je možné ručně specifikovat v době vzniku nebo se použije implicitní (od aktuálního okamžiku do nekonečna). Dostupné jsou pouze takové entity, jejichž platnost obsahuje aktuální okamžik, ve kterém jsou načítány z databáze. Temporální vlastnosti není možné upravovat a případné vazby zůstávají na úrovni posledních úprav. Uživatel nemůže ručně vytvářet nové verze, protože nemá žádné takové metody k dispozici. Tento mód je v TJDO aktivní v situacích, kde se používá rozhraní `Query`.

6.2.4 Netemporální s verzováním

V netemporálním režimu, který podporuje verzování objektů, jsou opět viditelné pouze poslední provedené změny. Ty jsou však dostupné bez ohledu na čas jejich platnosti a každá úprava způsobí vytvoření další verze. Dojde k ukončení předchozí verze a zahájení platnosti verze nové. Uživatel nemá možnost editovat temporální vlastnosti a ani nemůže ručně spravovat verzování, které se provádí automaticky.

Pomocí uvedeného způsobu je možné realizovat transakční temporálnost, kdy se jednotlivé změny vždy liší automaticky přiřazeným časem platnosti. V TJDO je však transakční čas realizován samostaným atributem a je na času platnosti zcela nezávislý.

6.3 Ukázka použití TJDO

Na obrázku 6.3 je ukázka jednoduchého použití knihovny *TJDO*. Na řádku 8 je deklarována privátní proměnná temporálního správce, který zprostředkovává dále uvedné operace. Řádek 11 ukazuje nejjednodušší způsob vytvoření připojení, kdy se nespecifikují žádné dodatečné informace jako například název a umístění databázového souboru. V takovém případě se použije implicitní cesta, která je nastavena na *db/tjdo.odb*. Oblast řádků 13 až 16 ohraničuje jednoduchou transakci, ve které dojde k uložení vytvořené instance třídy `Person`.

Dále následuje prostor pro další operace, jakými jsou editace nebo smazání existující instance. Tato funkcionalita bude rozebrána v dalších příkladech, které budou předpokládat tuto základní kostru.

Oblast řádků 20 až 24 ukazuje obecný temporální dotaz, který načte všechny historické verze všech objektů třídy `Person`, a jejich postupný výpis na standardní výstup.

Řádek 26 ukazuje ukončení práce s databází. Tento způsob je preferovaný, i když by bylo možné korektně ukončit spojení i pomocí `tpm.close()`. V takovém případě by ale bylo potřeba uzavřít ještě instanci třídy implementující rozhraní `PersistenceManagerFactory`, která vytvořila `PersistenceManager`, jenž je vnitřně používán. Uvedená situace může nastat v okamžiku, kdy je připojení k databázi vytvořeno pomocí metody `getTemporalPersistenceManager(PersistenceManagerFactory pmf)` instance `TJDOHelper`. Poté má uživatel referenci na tovární objekt a může ho ručně uzavřít metodou `close` stejně jako temporálního správce. Vnitřně používaný `PersistenceManager` je uzavřen vždy automaticky.

Na konec je v kódu ukázána definice perzistentní třídy, která je oanotována `@PersistenceCapable` a rozšiřuje temporální základ `TemporalBase`.

```
1 import cz.vutbr.fit.tjdo.core.TJDOHelper;
2 import cz.vutbr.fit.tjdo.core.TemporalBase;
3 import cz.vutbr.fit.tjdo.interfaces.TemporalPersistenceManager;
4 import java.util.List;
5 import javax.jdo.annotations.PersistenceCapable;
6
7 public class TjdoExample {
8   private TemporalPersistenceManager tpm;
9
10   public TjdoExample() {
11     tpm = TJDOHelper.getInstance().getTemporalPersistenceManager(); // vytvoreni spojeni
12
13     tpm.begin(); // zahajeni transakce
14     Person person = new Person(”John”);
15     tpm.makePersistent(person); // ulozeni instance
16     tpm.commit(); // ukonceni transakce
17
18     // dalsi operace
19
20     // temporalni dotaz na vsechny osoby
21     List<Person> list = (List) tpm.newTemporalQuery(Person.class).execute();
22
23     for (Person p˜: list) {
24       System.out.println(p.getName() + ”␣” + p.getValid().toString());
25     }
26
27     TJDOHelper.getInstance().close(); // ukonceni spojeni s˜databazi
28   }
29
30   @PersistenceCapable
31   private static class Person extends TemporalBase {
32     private String name;
33
34     public Person(String name) {
35       this.name = name;
36     }
37
38     public String getName() {
39       return name;
40     }
41
42     public void setName(String name) {
43       this.name = name;
44     }
45   }
46 } // class
```

Obrázek 6.3: Jednoduchá ukázka použití TJDO.

Na obrázku 6.4 je ukázána editace již uloženého objektu. Nejprve je potřeba vytvořit a provést dotaz, který vrací aktuální verze objektů, které je možné editovat (řádky 1 a 3). Pokud bychom nepotřebovali měnit platnost, stačil by netemporální dotaz. V tomto případě je potřeba použít temporální dotaz s nastaveným příznakem *snapshot*, který editaci umožní.

Pokud je objekt úspěšně načten, dojde k zahájení nové transakce, během které dojde k vlastní editaci (řádky 5 až 13). Na řádku 8 je úprava jména osoby klasickým způsobem, tedy pomocí metody *set*. Dále je ukázána změna času platnosti. Nejprve se vytvoří požadovaný interval (řádek 9). Tento je dále předán do konstruktoru třídy `TemporalIntervalList`, což je typ atributu času platnosti v datovém základu `TemporalBase`. Nově vytvořená platnost se nastaví opět pomocí metody *set* (řádek 11).

```
1  TemporalQuery query = tpm.newTemporalQuery(Person.class, "this.name.equals(\"John\")");
2  query.setSnapshot(true);
3  List<Person> results = (List) query.execute();
4
5  if (! results.isEmpty()) {
6    tpm.begin();
7    Person john = results.get(0);
8    john.setName("John Doe");
9    TemporalInterval interval = new TemporalInterval(2000, 1, 1, 2008, 1, 1);
10   TemporalIntervalList validity = new TemporalIntervalList(interval);
11   john.setValid(validity);
12   tpm.commit();
13 }
```

Obrázek 6.4: Ukázka editace objektu.

Obrázek 6.5 ukazuje temporální smazání objektu v zadaném intervalu platnosti. Požadovaný objekt je opět nutné nejprve načíst (řádky 1 až 3). Je zde opět použit temporální dotaz, u kterého se nastaví příznak `snapshot`, jenž dovolí vrátit aktuální verzi objektu. Na řádku 8 je potom vytvořen interval, ve kterém se bude požadovaný objekt mazat. Samotné odstranění zadané platnosti zařídí volání na řádku 9.

Pokud je původní platnost objektu větší než zadaný interval, nedojde k úplnému smazání objektu, ale pouze ke snížení rozsahu platnosti. V opačném případě je platnost zachována, ale dojde k nastavení příznaku `deleted` a objekt už nebude načítán v rámci dotazů.

```
1  TemporalQuery tq = tpm.newTemporalQuery(Person.class, "this.name.equals(\"John Doe\")");
2  tq.setSnapshot(true);
3  results = (List) tq.execute();
4
5  if (! results.isEmpty()) {
6    tpm.begin();
7    Person john = results.get(0);
8    TemporalInterval interval = new TemporalInterval(2002, 1, 1, 2006, 1, 1);
9    tpm.deletePersistent(john, interval);
10   tpm.commit();
11 }
```

Obrázek 6.5: Ukázka temporálního smazání objektu.

6.4 Omezení implementovaného systému

V této sekci budou uvedeny nevýhody a omezení, které jsou na uživatele v rámci TJDO kladeny.

Dostupnost všech temporálních atributů

Zejména z důvodu kontroly referenční integrity systém vyžaduje, aby uživatelské třídy implementovaly metody pro získání všech temporálních atributů (tzv. *gettery*). Pokud by tomu tak nebylo, systém by nedokázal zjistit a kontrolovat vztahy mezi objekty. Databáze se poté může dostat do nekonzistentního stavu, například porušením temporální referenční integrity.

Požadavky na uživatelské třídy

Uživatel systému je při definici vlastních temporálních tříd nucen rozšiřovat `TemporalBase`, tedy třídu implementující požadované temporální atributy. Z důvodů použití specifikace *JDO* je dále potřeba u každé datové třídy uvádět anotaci `@PersistenceCapable`, která umožní její zpracování v rámci `PersistenceManager` objektu.

Dvoufázové zpracování provedených změn

Úspěšné ukončení uživatelské transkace (*commit*) je na úrovni TJDO rozděleno do dvou fází, které jsou blíže popsány v podsekci 5.2.3 na straně 50. K porušení refereční integrity by nemělo nikdy dojít, protože ta je kontrolována v rámci první transakce, a k vlastním změnám v databázi dochází zase až ve druhé fázi. Nevýhodou tohoto přístupu je ale dvojí průchod všemi ukládanými položkami.

Alternativou, jak získat všechny temporální objekty, by bylo rekurzivní procházení a hledání temporálních atributů u každé z instancí. V případě složitějších vazebních vztahů by však s tímto přístupem mohly být některé objekty zpracovávány vícekrát.

Větvení historie není podporováno

TJDO neumožňuje editaci neaktuálních (z hlediska transakčního času) temporálních verzí. Měnitelná je tedy vždy pouze poslední provedená úprava. V opačném případě by mohlo docházet k větvení historií, což s sebou přináší řadu komplikací nejen z hlediska dotazování či referenční integrity. Z důvodů komplexnosti tohoto problému bylo dělení historie v TJDO zakázáno. Dosahuje se toho mimo jiné pomocí příznaku `current` třídy `TemporalBase` a jeho kontrolou v rámci zpracování provedených změn (*commit*).

6.5 Temporální vlastnosti systému

Objekty s budoucím časem platnosti

TJDO implicitně nedovoluje vytvářet temporální objekty, jejichž čas platnosti začíná v budoucnu vůči aktuálnímu sytémovému času. Nicméně je toto chování možné změnit editací statického příznaku `futureValidityAllowed` třídy `TemporalIntervalList` pomocí opět statické metody `setFutureValidityAllowed`.

Pokud není při ukládání nové temporální instance uvedeno jinak, je její čas platnosti nastaven od aktuálního okamžiku po nekonečno.

Netemporální dotazy

Z důvodů zachování zpětné kompatibility s klasickými JDO dotazy, jsou prostřednictvím temporálního správce k dispozici metody `newQuery()` s různými parametry, které vracejí pouze aktuální nesmazané verze instancí. Při těchto dotazech je však kontrolován čas platnosti. Aby mohl být objekt načten z databáze pomocí těchto metod, musí jeho platnost obsahovat aktuální okamžik, ve kterém je dotaz prováděn. Například objekty s platností končící v minulosti tedy nebudou nikdy pomocí těchto netemporálních dotazů načteny.

Změna platnosti odkazovaného objektu

Další vlastností, na kterou je vhodné upozornit, je porušení integrity změnou času platnosti u odkazovaného objektu. Uvažujme například vazbu mezi automobilem a jeho značkou, kdy každý objekt třídy `Car` obsahuje jeden atribut třídy `Brand`. Předpokládejme jednu takovou dvojici s časem platnosti $\langle 2000, 2010\rangle$. Pokud nyní změníme platnost u objektu reprezentující značku třeba tak, že vznikne nová verze s platností $\langle 2005, 2008\rangle$, dojde k porušení temporální referenční integrity, protože automobil s původní platností $\langle 2000, 2010\rangle$ odkazuje na značku s platností $\langle 2005, 2008\rangle$. V databázi by sice existovala starší verze značky s platností $\langle 2000, 2005) \cup (2008, 20010\rangle$, ale automobil odkazuje pouze verzi aktuální. Proto jsou podobné operace v TJDO zakázány jako porušení temporální referenční integrity a je vyvolána příslušná výjimka.

6.6 Možná rozšíření systému

V této poslední sekci budou navržena rozšíření, která by TJDO mohla dále vylepšit.

Temporální omezení

V TJDO je implementována temporální referenční integrita. V temporálních databázích obecně se ale mohou vyskytovat další omezení, jejichž dodržení systém automaticky hlídá. Mezi taková omezení může patřit například některá z následujících situací.

- Pracovníkovi nesmí být v jednom okamžiku přidělena více než jedna úloha.
- Firma nepřijímá osoby, které u ní již v minulosti pracovaly a byly propuštěny.
- Lékař nesmí předepsat zdravotní pobyt osobě, které ho již někdy během posledních 5 let předepsal.
- Osoba nesmí provádět určitou činnost, dokud není plnoletá.

Optimalizace ukládání temporálních verzí

Aktuální stav TJDO při vytváření nových temporálních verzí ukládá celé nové objekty, které vytvoří pomocí serializace a kopie instance. Pokud však dojde například ke změně několika málo z mnoha atributů, dochází k redundanci, která zbytečně zatěžuje datové úložiště a zvyšuje požadavky na diskový prostor. Užitečným rozšířením by tedy bylo implementovat vytváření nových verzí tak, aby se ukládaly pouze změněné atributy.

Větvení historie

Netriviálním rozšířením by byla implementace řešící problém větvení historie, který je v TJDO zakázán. Tato problematika byla zmíněna v sekci 6.4 na straně 62.

Bytecode enhancement

ObjectDB (objektová databáze, na které byl systém vyvíjen) poskytuje tzv. *bytecode enhancement*, což je automatická úprava zkompilovaných *.class* souborů. Tímto způsobem jsou do datových tříd s anotací `@PersistenceCapable` vloženy členské atributy a metody, které jsou pro manipulaci s objekty potřebné. TJDO nutí uživatele tuto situaci řešit pomocí rozšíření datového základu `TemporalBase`. Další vhodnou úpravou by tedy bylo implementovat vlastní *enhancement*, který by zajistil vložení atributů a metod z `TemporalBase`, aniž by uživatel musel tuto třídu dědit.

Kapitola 7

Závěr

Tato práce se v teoretické části věnovala temporálním databázovým systémům, jazyku Java a propojením těchto technologií. Čtenáři byly vysvětleny základní pojmy probíraných oblastí a na příkladech ukázány způsoby použití. V prakticky zaměřené části byl popsán návrh a implementace temporálního rozšíření standardu *Java Data Objects (JDO)*.

Druhá kapitola se zabývala obecně problematikou temporálních databázových systémů. Byly zde vysvětleny důvody jejich vzniku, potřeba či způsoby nasazení a ve stručnosti byl shrnut historický vývoj. Dále bylo uvedeno, jakým způsobem je možné čas reprezentovat a členit. Z praktického hlediska byla vysvětlena problematika referenční integrity, která se v těchto systémech zásadně liší od té, která je implementována v relačních databázích. Následovala sekce o temporální logice, která je formálním základem všech temporálních databázových systémů. Vycházejí z ní dotazovací jazyky popsané v další části. Nakonec byly zmíněny některé existující implementace, které svými vlastnostmi spadají mezi temporální databázové systémy.

Třetí kapitola se zaměřila na jazyk Java a způsoby, kterými je možné v tomto programovacím jazyce trvale uchovávat data. První sekce ukázala základní metody jako je manuální ukládání dat do souborů nebo klasické využití relační databáze. Ve druhé sekci byly popsány složitější mechanizmy, které poskytují jednodušší a částečně automatizovanou podporu pro perzistenci dat. Konkrétně zde byly popsány techniky serializace a objektově-relačního mapování. V závěru sekce byly uvedeny nejrozšířenější rámce a standardy v této oblasti.

Čtvrtá kapitola se celá věnovala detailnějšímu přiblížení standardu Java Data Objects. V úvodních sekcích byla stručně zmíněna historie a vysvětleny základní pojmy. Hlavní část této kapitoly tvořila sekce věnující se celkové architektuře, nejdůležitějším třídám a rozhraním, životnímu cyklu konkrétních instancí a v neposlední řadě transakčnímu zpracování. Na závěr byly uvedeny ukázky mapování do relační databáze.

Pátá kapitola čtenáři prezentovala návrh temporálního rozšíření standardu JDO tak, aby bylo možné uchovávat a dotazovat temporální objektová data. Nejprve byly uvedeny požadavky na výsledný produkt včetně návrhu jejich realizace. Dále pak byly uvedeny UML diagramy, které ukázaly architekturu a rozhraní navrženého systému.

V šesté kapitole byly uvedeny a vysvětleny techniky a postupy použité při implementaci výsledného temporálního rozšíření JDO. Úvod patřil přehledu základních vlastností, kde se čtenář dozvěděl, jakým způsobem se realizují a spravují temporální aspekty uživatelských objektů. Následovala sekce s ukázkou použití, kde bylo na příkladech vysvětleno, jak je možné temporální data ukládat, upravovat a dotazovat. Další sekce uvedly omezení vytvořeného systému a hlavní temporální vlastnosti. V poslední sekci pak bylo navrženo

několik možných cest, kterými by se bylo vhodné ubírat v případě rozšíření stávající funkconality nebo optimalizace.

7.1 Testování

Celý systém byl vyvíjen a testován s využitím objektové databáze *ObjectDB*, která poskytuje implementaci specifikací *JPA* a *JDO*. Jedná se o komerční software, ale pro malé projekty nebo akademické účely, kde vyhovuje omezení kladené na počet entitních tříd a objektů, je zdarma. Součástí výsledných zdrojových kódů jsou jednotkové a funkční testy využívající knihovnu *JUnit*.

Literatura

[1] Temporal PostgreSQL Summary. http://pgfoundry.org/projects/temporal/, 2009 [cit. 2011-08-11].

[2] Bauer, C.; King, G.: *Java Persistence with Hibernate*. New York: Manning, 2007 [cit. 2011-08-17], iSBN 1-932394-88-5.

[3] Chomicki, J.; Toman, D.: Temporal Logic in Database Query Languages. In *Encyclopedia of Database Systems*, 2009 [cit. 2011-08-08], s. 2987–2991.

[4] Chomicki, J.; Toman, D.; Böhlen, M. H.: Querying ATSQL databases with temporal logic. *Transactions on Database Systems (TODS)*, ročník 26, 2001 [cit. 2011-08-11], iSSN 0362-5915, EISSN 1557-4644.

[5] DeMichiel, L.: JSR 317: Java Persistence API. http://download.oracle.com/otn-pub/jcp/persistence-2.0-fr-eval-oth-JSpec/persistence-2_0-final-spec.pdf, 2009-03-13 [cit. 2011-08-16].

[6] ISO: SQL-92 Standard. www.contrib.andrew.cmu.edu/ shadow/sql/sql1992.txt, 1992-07-30 [cit. 2011-08-07].

[7] JBoss: Hibernate. http://www.hibernate.org/, 2011-02-17 [cit. 2011-08-17].

[8] JDO, A.: Which Persistence Specification? http://db.apache.org/jdo/jdo_v_jpa.html, 2011 [cit. 2011-08-19].

[9] Kolář, D.: *Pokročilé databázové systémy (prostorové databáze)*. Brno: FIT VUT v Brně, 2006 [cit. 2011-08-01], studijní opora předmětu PDB.

[10] Oracle: Oracle Database 11g Workspace Manager Overview. http://www.oracle.com/technetwork/database/twp-appdev-workspace-manager-11g-128289.pdf, 2009-09-01 [cit. 2011-08-11].

[11] Oracle: Workspace Manager Developer's Guide. http://download.oracle.com/docs/cd/E11882_01/appdev.112/e11826.pdf, 2010-09-01 [cit. 2011-08-11].

[12] for Relational Databases, J. P.: *Richard Sperko*. Apress, 2003 [cit. 2011-08-13], iSBN 1590590716.

[13] Roos, R. M.: *Java Data Objects*. London: Addison Wesley, 2003 [cit. 2011-08-19], iSBN 0-321-12380-8.

[14] Russel, C.: Java Data Objects 3.0 (JSR 243). 2010-04-09 [cit. 2011-08-19], http://download.oracle.com/otndocs/jcp/jdo-3.0-mrel3-eval-oth-JSpec/.

[15] Snodgrass, R.: The Temporal Query Language TQUEL. *Transactions on Database Systems (TODS)*, ročník 12, č. 2, 1987 [cit. 2011-08-07], iSSN 0362-5915, EISSN 1557-4644.

[16] Snodgrass, R.: *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 2000 [cit. 2011-08-03], iSBN 1-55860-436-7.

[17] Snodgrass, R. T.; Ahn, I.; Ariav, G.; aj.: TSQL2 Language Specification. *ACM SIGMOD Record*, ročník 23, č. 1, 1994 [cit. 2011-08-10].

[18] Snodgrass, R. T.; Ahn, I.; Ariav, G.; aj.: A TSQL2 Tutorial. *ACM SIGMOD Record*, ročník 23, č. 3, 1994 [cit. 2011-08-10].

[19] Steiner, A.: *A Generalisation Approach to Temporal Data Models and their Implementations*. Swiss Federal Institute of Technology Zürich, 1998 [cit. 2011-08-03], disertační práce.

[20] Teradata: Teradata TDW. http://developer.teradata.com/general/training/temporal-edw, 2011-02-09 [cit. 2011-08-11].

[21] TimeConsult: TimeDB – A Bitemporal Relational DBMS. http://www.timeconsult.com/Software/AboutTimeDB1.0.html, 2009 [cit. 2011-08-11].

[22] Tomek, J.: *TSQL2 interpret nad relační databází*. Brno: FIT VUT v Brně, 2009 [cit. 2011-08-10], diplomová práce.

[23] Wikipedia: QUEL Query Languages. http://en.wikipedia.org/wiki/QUEL_query_languages, 2011-03-29 [cit. 2011-08-07].

[24] Wikipedia: Java Data Objects. http://en.wikipedia.org/wiki/Java_Data_Objects, 2011-06-18 [cit. 2011-08-18].

[25] Wikipedia: Temporal Database. http://en.wikipedia.org/wiki/Temporal_database#History, 2011-06-29 [cit. 2011-08-01].

[26] Wikipedia: SQL. http://cs.wikipedia.org/wiki/SQL, 2011-07-02 [cit. 2011-08-07].

Příloha A

Ukázky způsobů perzistence objektů v jazyce Java

```
1 import javax.xml.bind.annotation.XmlRootElement; // potrebne pro XML
2 import java.io.Serializable; // potrebne pro serializaci
3
4 @XmlRootElement
5 public class Person implements Serializable {
6   private long id; // identifikator
7   private String name; // cele jmeno
8
9   public Person(long id, String name) {
10     this.id = id;
11     this.name = name;
12   }
13
14   public Person() {
15   }
16
17   public long getId() {
18     return id;
19   }
20
21   public void setId(long id) {
22     this.id = id;
23   }
24
25   public String getName() {
26     return name;
27   }
28
29   public void setName(String name) {
30     this.name = name;
31   }
32 } // class
```

Obrázek A.1: Třída *Person* bude použita v některých z následujících ukázek.

```
1  import java.io.*;
2
3  class DataFile {
4    private static final int EOF = −1; // konec souboru
5    private static final String FILENAME = ”data.dat”; // nazev souboru
6
7    public static void main(String args[]) {
8      // a) ulozeni dat
9      try {
10       FileOutputStream outFile = new FileOutputStream(FILENAME);
11       int dataToStore = 42;
12       outFile.write(dataToStore);
13       outFile.close();
14     }
15     catch (IOException ioe) {
16       System.err.println(ioe.toString());
17     }
18
19     // b) nacteni ulozenych dat
20     try {
21       FileInputStream inFile = new FileInputStream(FILENAME);
22       int buffer = 0;
23
24       while ((buffer = inFile.read()) != EOF) {
25         System.out.println(”Nacteno:␣” + buffer);
26       }
27
28       inFile.close();
29     }
30     catch (IOException ioe) {
31       System.err.println(ioe.toString());
32     }
33   } // main
34 } // class
```

Obrázek A.2: Ukázka manuálního ukládání a načítání dat ze souboru v Javě

```java
import javax.xml.bind.*;
import java.io.*;

class JAXB {
  private static final String FILENAME = "person.xml"; // nazev souboru

  public static void main(String [] args) {
    // a) ulozeni objektu
    try {
      JAXBContext context = JAXBContext.newInstance(Person.class);
      Marshaller marshaller = context.createMarshaller();
      FileWriter writer = new FileWriter(FILENAME);
      marshaller.marshal(new Person(1, "Alois"), writer);
    }
    catch (Exception e) {
      e.printStackTrace();
    }

    // b) nacteni objektu
    try {
      JAXBContext context = JAXBContext.newInstance(Person.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      FileReader reader = new FileReader(FILENAME);
      Person newPerson = (Person) unmarshaller.unmarshal(reader);
      System.out.println("Nacteno: id = " + newPerson.getId() +
        ", name = " + newPerson.getName());
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  } // main
} // class
```

Obrázek A.3: Uložení a načtení objektu pomocí JAXB.

```
1 import java.sql.*;
2
3 class Main {
4   private static final String DRIVER = "com.mysql.jdbc.Driver";
5   private static final String URL = "jdbc:mysql://localhost:3306/java";
6   private static final String USER = "username";
7   private static final String PASS = "password";
8   private static Connection connection = null; // spojeni s db
9   private static Statement statement = null; // dotaz pro db
10  private static ResultSet result = null; // vysledek dotazu
11
12  public static void main(String args[]) {
13    try {
14      Person person = new Person(1, "Alois Andrle"); // ukladany objekt
15      Class.forName(DRIVER); // nacteni ovladace
16      connection = DriverManager.getConnection(URL, USER, PASS);
17      statement = connection.createStatement();
18
19      // a) ulozeni stavu objektu
20      statement.execute("truncate persons"); // pro ucel ukazky
21      String insertSql = "insert into persons (id, name) values (" +
22        person.getId() + ", '" + person.getName() + "')";
23      statement.execute(insertSql); // vlozeni zaznamu
24      // b) nacteni stavu objektu
25      String selectSql = "select id, name from persons where id = '1'";
26      result = statement.executeQuery(selectSql); // ziskani zaznamu
27
28      while (result.next()) { // zpracovani vysledku
29        Person personLoaded = new Person(result.getInt("id"),
30          result.getString("name"));
31        System.out.println("Nacteno: id = " + personLoaded.getId() +
32          ", name = " + personLoaded.getName());
33      }
34    }
35    catch(Exception e) { System.err.println(e.toString()); }
36    finally {
37      try {
38        if (result != null) { result.close(); }
39        if (statement != null) { statement.close(); }
40        if (connection != null) { connection.close(); }
41      }
42      catch (SQLException sqle) { System.err.println(sqle.toString()); }
43    }
44  } // main
45 } // class
```

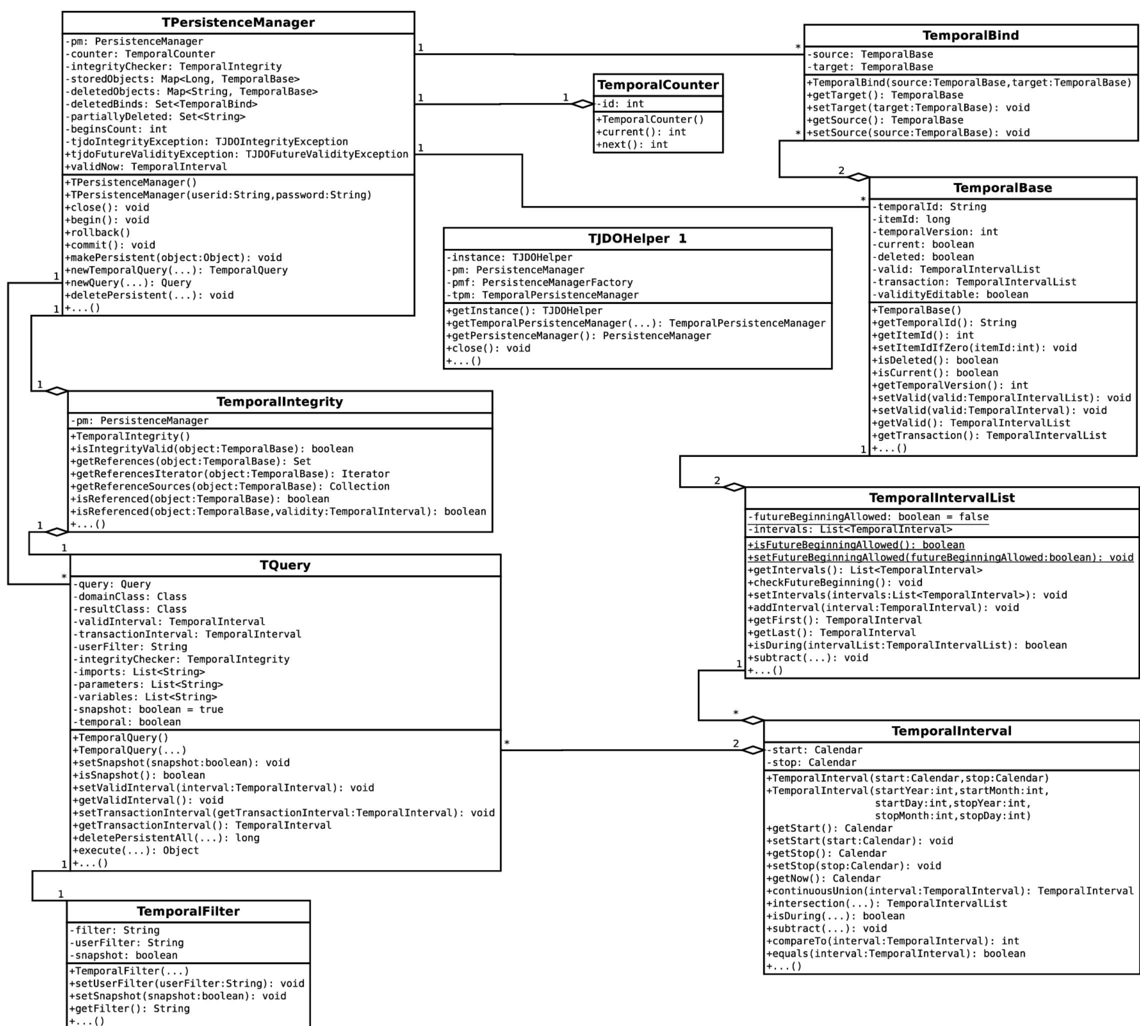
Obrázek A.4: Uložení a načtení stavu objektu pomocí JDBC.

```
1 import java.io.*;
2
3 class Serialization {
4   private static final String FILENAME = "person.dat"; // nazev souboru
5
6   public static void main(String args[]) {
7    // a) vytvoreni a~ulozeni stavu objektu
8     try {
9       FileOutputStream outFile = new FileOutputStream(FILENAME);
10      ObjectOutputStream outObject = new ObjectOutputStream(outFile);
11      Person person = new Person(1, "Alois␣Andrle");
12      outObject.writeObject(person);
13      outObject.close();
14      outFile.close();
15    }
16    catch (IOException ioe) {
17      System.err.println(ioe.toString());
18    }
19
20    // b) nacteni stavu objektu
21     try {
22      FileInputStream inFile = new FileInputStream(FILENAME);
23      ObjectInputStream inObject = new ObjectInputStream(inFile);
24      Person personLoaded = (Person) inObject.readObject();
25      System.out.println("Nacteno:␣id␣=␣" + personLoaded.getId() +
26        ",␣name␣=␣" + personLoaded.getName());
27      inObject.close();
28      inFile.close();
29    }
30    catch (Exception e) {
31      System.err.println(e.toString());
32    }
33
34  } // main
35 } // class
```

Obrázek A.5: Ukázka využití serializace objektu.

Příloha B

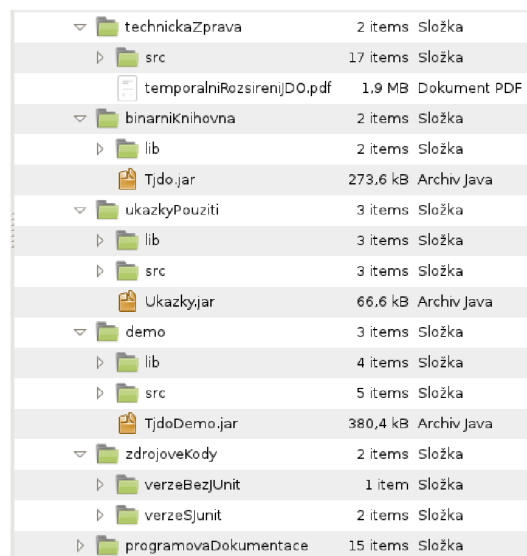
Kompletní diagram tříd



Obrázek B.1: Diagram hlavních tříd návrhu.

Příloha C

Obsah DVD



Obrázek C.1: Obsah DVD

- `technickaZprava` – zdrojové kódy v LaTeXu a finální PDF soubor.
- `binarniKnihovna` – finální Java archív použitelný jako externí knihovna. Dále jsou zde další potřebné knihovny v binární podobě.
- `ukazkyPouziti` – několik příkladů, na kterých je ukázáno, jak je možné s TJDO pracovat.
- `demo` – jednoduchá desktopová aplikace demonstrující vlastnosti TJDO.
- `zdrojoveKody/verzeBezJUnit` – zdrojové kódy, ze kterých byl vytvořen finální Java archív.
- `zdrojoveKody/verzeSJUnit` – zdrojové kódy obsahující balíček `debug` pro debugování celého systému a adresář s JUnit testy.
- `programovaDokumentace` – vygenerovaná JavaDoc HTML dokumentace.