VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PERZISTENTNÍ PROSTŘEDÍ PRO JAVA VIRTUAL MACHINE

PETRI NETS VIRTUAL MACHINE PERSISTENCY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAN BAYER**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Bayer Jan**
Obor: Informační technologie
Téma: **Perzistentní prostředí pro Java Virtual Machine**
Petri Nets Virtual Machine Persistency
Kategorie: Operační systémy

Pokyny:

1. Seznamte se s virtuálním strojem jazyka Java (JVM) a prostudujte možnosti perzistence v prostředí Java (Java Persistence API, Hibernate, apod.)
2. Proveďte analýzu silných a slabých stránek stávajících řešení perzistence.
3. Navrhněte systém pro perzistenci objektů JVM, který umožňuje ukládat a načítat stav vybrané podmnožiny objektů Javy. Systém musí pracovat bez využití dalších nástrojů (databáze apod.)
4. Navržený systém implementujte a otestujte na vhodné množině testových sad.
5. Analyzujte dosažené výsledky a navrhněte možnosti dalšího vývoje.

Literatura:

- The Java EE 6 Tutorial, Introduction to the Java Persistence API, http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html, 2017.
- Domovská stránka projektu Hibernate, http://hibernate.org/, 2017.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato bakalářská práce se věnuje problematice perzistence v jazyce Java. Představuje a porovnává existující standardy a systémy, a zabývá se návrhem a implementací perzistenčního prostředí pro platformu Java SE s výstupem ve formě XML souborů.

Abstract

This Bachelor's thesis deals with persistence for Java language. It presents and compares already existing standarts and systems, and introduces a design and implementation of persistence for Java SE platform with object data stored in XML files.

Klíčová slova

perzistence, jazyk Java, jvm, jpa, jdo, ejb, orm, serializace

Keywords

persistence, Java language, jvm, jpa, jdo, ejb, orm, serialization

Citace

BAYER, Jan. *Perzistentní prostředí pro Java Virtual Machine*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Perzistentní prostředí pro Java Virtual Machine

Prohlášení

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

.....
Jan Bayer
22. května 2018

Poděkování

Touto cestou bych chtěl poděkovat vedoucímu mé práce Ing. Radku Kočímu PhD. za jeho ochotu a odborné rady, které mi poskytl v průběhu její tvorby. Také bych chtěl poděkovat Guillaumovi Marymu ze společnosti Eloquant za jeho odborné konzultace.

Obsah

1 Úvod **3**
- 1.1 Cíl práce . . . 3
- 1.2 Struktura práce . . . 4

2 Jazyk Java **5**
- 2.1 Základy jazyka Java . . . 5
- 2.2 Bajtkód a Java Virtual Machine . . . 5
- 2.3 Java Reflection . . . 6
- 2.4 Anotace . . . 7

3 Perzistence **8**
- 3.1 Principy perzistence . . . 8
- 3.2 Přístupy k perzistenci a jejich vlastnosti . . . 9
  - 3.2.1 Serializace . . . 9
  - 3.2.2 Relační databáze . . . 10
  - 3.2.3 Objektové databáze . . . 12
  - 3.2.4 Cíl práce . . . 13
- 3.3 Standardy perzistence jazyka Java . . . 13
  - 3.3.1 Java Database Connectivity . . . 14
  - 3.3.2 Java Data Object API . . . 14
  - 3.3.3 Enterprise Java Bean 2.0 . . . 15
  - 3.3.4 Enterprise Java Beans 3.0 a Java Persistence API . . . 17
- 3.4 Existující systémy a jejich porovnání . . . 19
  - 3.4.1 Hibernate . . . 20
  - 3.4.2 EclipseLink . . . 20
  - 3.4.3 DataNucleus . . . 20
  - 3.4.4 Shrnutí a porovnání cílem práce . . . 21

4 Návrh a implementace **22**
- 4.1 Analýza požadavků . . . 22
- 4.2 Východiska návrhu . . . 22
- 4.3 Architektura tříd a rozhraní . . . 23
- 4.4 Jádro perzistence . . . 24
- 4.5 Strategie ukládání . . . 24
- 4.6 Události . . . 25
- 4.7 Třída StorageContext . . . 25
- 4.8 Třída ClassManager . . . 25
- 4.9 Použité technologie . . . 26

4.10 Testování 27

5 Závěr **28**

Literatura **29**

A Manuál **31**

B Obsah CD **33**

Kapitola 1

Úvod

V aplikacích, u kterých nelze zaručit nebo je nežádoucí jejich neustálý běh, vzniká problém s uchováváním dat i po konci jejich běhu. Během něj jsou data přístupná v operační paměti, avšak po ukončení aplikace je jejich alokovaný prostor v paměti uvolněn a data jsou ztracena. V případě potřeby dále s těmito daty pracovat nebo je uchovávat je proto třeba je uložit do perzistentního, tedy stálého prostředí, které nabídne přístup k těmto datům nezávisle na životním cyklu aplikací.

Problematika trvalého uchovávání dat doprovází svět výpočetních systémů již od jeho začátku. Komplexní řešení tohoto problému se začalo vyvíjet v 60. letech 20. století v podobě databázových systémů, které přinesly možnost uchovávat data nezávisle na jejich zdroji po neomezenou dobu. Od té doby se systémy pro perzistenci vyvinuly do projektů velkých rozměrů, jejichž funkci využívají mnohé profesionální aplikace, a stala se z nich důležitá samostatná disciplína.

V jazyce Java je tomuto tématu přikládána velká váha. Aplikace v ní vytvořené jsou často robustními implementacemi informačních systémů či webových aplikací, ve kterých je nutné uchovávat data po dlouhou dobu, a to i po konci aplikace. Pro platformu Java Enterprise Edition, která tvoří prostředí pro vývoj těchto aplikací, vzniklo několik systémů nabízející různé možnosti perzistence dat, z nichž některé následně velkou mírou přispěly k vytvoření standardů a iniciovaly Java Specification Request.[1]

1.1 Cíl práce

Cílem této práce je seznámit se s přístupy k perzistenci v jazyce Java. Na základě porovnání principů posléze navrhnout a implementovat systém perzistence, který bude vhodný pro menší a střední aplikace v rámci Java Standard Edition a nebude vyžadovat žádné další zdroje či služby běžící na cílové platformě.

Motivací pro tvorbu takovéhoto systému je simulace Petriho sítí reprezentovaných objekty v jazyce Java, které se provádí v určitých časových intervalech, a po skončení těchto simulací je třeba uchovat stav celého systému. Proto jsme se rozhodli vytvořit perzistentní prostředí, které bude pro ukládání dat využívat XML soubory.

[1] Java Specification Request je žádost o změnu nebo přidání některé části knihoven a dalších komponent jazyka Java, kterou zpracovává Java Community Process.

1.2 Struktura práce

Tato práce je rozdělena do několika kapitol. Kapitola 2 uvádí základní informace o jazyku Java, o jeho virtuálním stroji a o dalších částech Javy důležitých pro tuto práci. V kapitole 3 bude objasněn pojem perzistence a budeme se věnovat již existujícím poznatkům o perzistenci v jazyce Java a porovnání konceptů, které používají jednotlivé systémy. Kapitola 4 potom popisuje návrh vytvořeného systému, jeho strukturu, popis implementačních detailů a testování.

Kapitola 2

Jazyk Java

V této kapitole se budeme stručně věnovat základním principům jazyka Java a jeho interpretu, a představíme důležité funkce a API, na kterých tato práce staví.

2.1 Základy jazyka Java

Java je interpretovaný, objektově orientovaný programovací jazyk inspirovaný jazyky C, C++, C#, Smalltalk a dalšími. Jeho vývoj začal v 90. letech 20. století s původním cílem vytvořit jazyk, který by byl nezávislý na platformě a poskytoval by vysokou mírou přenositelnosti. Tyto dvě vlastnosti byly v počátcích důležité zejména pro řešení problémů při vývoji software, jež by mohl být distribuován po síti a byl by spustitelný na všech jejích prvcích. Java také zásadně zlepšila bezpečnost aplikací odebráním možnosti přímé práce s pamětí, jenž zůstala pouze v režii interpretu. Přímý přístup aplikace do fyzického paměťového prostoru cílové platformy tak byl zcela znemožněn. Tento způsob operace s pamětí byl důležitou podmínkou pro bezpečnost appletů,[1] které zejména v začátcích Internetu zaznamenaly obrovský úspěch a pomohly k dalšímu vývoji Javy. [16]

Pro dosažení zmiňované přenositelnosti a bezpečnosti je Java vyvíjena jako interpretovaný jazyk, o jehož interpretu pojednává další kapitola.

2.2 Bajtkód a Java Virtual Machine

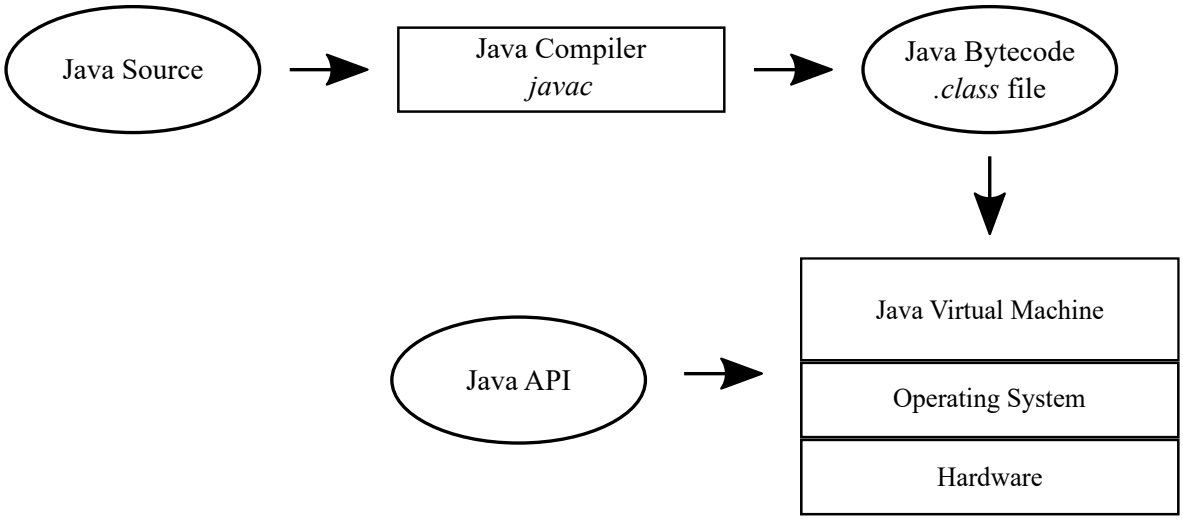
Jazyk Java byl od začátku záměrně vyvíjen jako interpretovaný jazyk, což typicky přináší problémy zejména v oblasti rychlosti. Výhodou však je univerzálnost napsaného kódu. Zdrojový kód je primárním kompilátorem přeložen do bajtkódu[2] v `class` souborech reprezentující jednotlivé třídy, které je následně možné spouštět na různých zařízeních disponující interpretem bajtkódu. V případě jazyka Java se interpret nazývá Java Virtual Machine a spolu s knihovnami Java API tvoří *Java Runtime Enviroment*, tedy prostředí pro interpretaci bajtkódu a běh aplikací.

Java Virtual Machine (dále jen JVM) je abstraktní virtuální stroj s vlastní instrukční sadou a paměťovým prostorem. Jako vstup očekává `class` soubory, což jsou binární data obsahující instrukce, tabulku symbolů a další dodatečné informace o programu. Tento for-

[1] Applet je multiplatformní program v Javě určený pro přenos přes Internet a automatické spuštění v prohlížeči.

[2] Pojem bajtkód, anglicky *byte code*, označuje instrukční sadu pro určitý interpret, v případě Javy pro Java Virtual Machine.

mát je nezávislý na platformě a odpovídá konstrukcím vytvořeným v jazyce Java. JVM po načtení `class` souboru interpretuje kód a přeloží ho do jazyka pro cílovou platformu, vizte obrázek 2.1. Jednou z implementací JVM je HotSpot,[3] který vyhledává nejpoužívanější kód a interpretuje jej pomocí tzv. *Just-in-time* metody (dále jen JIT). Ta umožňuje přímý překlad bajtkódu do nativního kódu v době provádění, což se liší od klasické interpretace, která kód překládá několika fázemi pro různé virtuální stroje. JIT je velmi efektivní technikou, jak docílit vyšší rychlost interpretace a používá se pro nejčastěji používané části kódu (JVM při prvním průchodu zjistí, které části jsou vhodné pro JIT). [10]



Obrázek 2.1: Schéma překladu v jazyce Java

Důležitou částí JVM je *garbage collector*, což je automatizovaný systém pro správu paměti. Jeho základní funkcí je dealokace paměti objektů, které již nejsou v programu používané. Tento princip je velmi důležitý, protože umožňuje vyšší míru abstrakce a odděluje programátora od nízkoúrovňových operací s pamětí.

2.3 Java Reflection

Java Reflection API[4] je API přinášející do jazyka Java možnost reflexe, tj. přístup k objektům a struktuře programu za běhu programu. Umožňuje programátorovi introspekci existujících objektů - jejich tříd, metod, atributů či konstruktorů. Se všemi výhodami však přináší i úskalí. Pomocí Java Reflection API je možné porušit například principy zapouzdření a to přístupem k privátním atributům objektů. Navíc je tento přístup náročný na režii ze strany JVM, a proto je doporučeno jeho použití dobře promyslet a odůvodnit.

V našem systém jako i v ostatních technologiích pro perzistenci hraje introspekce a reflexe velmi důležitou roli a její užití bývá v těchto případech opodstatněné.[5] Důvodem je nutnost introspekce objektů za běhu aplikace pro práci s atributy, metodami, datovými typy, a jejich následného zpracování.

[3] HotSpot je implementace JVM vyvíjená firmou Oracle.
[4] Java Reflection API tvoří package `java.lang.reflect`.
[5] Výjimkou je přístup skrze *bajtkód enhancement* využívaný např. JDO viz kapitola 3.3.2.

2.4 Anotace

Anotace byly v Javě uvedeny ve verzi 5 a představují formu metadat, která nemají přímý vliv na vykonávání programu. Umožňují přidání dodatečných informací pro atributy, metody či celé třídy, které mohou být následně využity např. při kompilaci nebo za běhu programu (*runtime* prostředí). Při tvorbě anotace je nutné definovat její viditelnost v rámci JVM, a to pomocí meta-anotace[6] `@Retention`, která nabízí tři možnosti: a) anotace může plnit pouze informativní účel ve zdrojovém kódu a bude ignorována kompilátorem, b) anotace je zachována pro kompilátor, ale skryta pro JVM a c) anotace je zachována po celou dobu běhu programu a je viditelná pro JVM.

V ukázce 2.1 můžeme vidět definici anotace `@ObjectId`. V tomto případě se jedná o anotaci viditelnou pro JVM (`RetentionPolicy.RUNTIME`). Meta-anotací `@Target` je možno definovat cíl anotace. V našem případě je tedy anotace určena pro atributy.[7] Příklad aplikace takovéto anotace je uveden v ukázce 2.2, kde pomocí anotace `@ObjectId` označujeme atribut *objectId*.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface ObjectId {}
```

Výpis 2.1: Ukázka definice anotace

```
public class Test1 {
    @ObjectId
    public Long objectId;

    private String text;
    private int number;
}
```

Výpis 2.2: Použití anotace ve zdrojovém kódu

[6] Meta-anotace jsou používány pro anotování jiných anotací.
[7] V anglické terminologii se atributy tříd často označují jako *field*.

Kapitola 3

Perzistence

Tato kapitola se věnuje definici perzistence, jejím základním principům a představuje několik známých standardů a jejich implementací.

Pojem perzistence se dá uchopit dvěma způsoby. Může to být schopnost dat přežívat i po konci běhu aplikace, což dělí data na tzv. *transient*[1] data, která existují pouze do konce procesu a následně jsou smazána JVM pomocí *garbage kolektoru*, a na perzistovaná data, které existují i nadále. Perzistenci lze však pochopit taky jako vlastnost systému umět data perzistentními učinit, tzn. možnost data uložit na úložiště nezávislé na běhu aplikace.

3.1 Principy perzistence

Perzistence dle původního konceptu podle Atkinsona a spol. [5] by měla být založena na těchto 3 principech:

- **Ortogonalita** - všechna data mají možnost perzistence bez ohledu na jejich typ.
- **Tranzitivita** - délka života dat[2] je dána jejich dostupností pro ostatní data (tzn. v případě, že data určená k perzistenci odkazují na data neperzistovaná, je třeba tyto data také začlenit do systému perzistence a tím sjednotit jejich délku života).
- **Nezávislost** - perzistence musí pracovat stejným způsobem se všemi daty bez ohledu na jejich zdroj. Data získaná za běhu aplikace budou perzistována stejně jako ta, která byla načtena z jiného zdroje (např. disk nebo jiná úložiště).

Jak ukazuje tabulka 3.1, naprostá většina dnes používaných přístupů však ortogonalitu a nezávislost perzistence nesplňují, neboť obě tyto vlastnosti silně ovlivňují virtuální stroje programovacích jazyků (v případě Javy či C#), které by musely integrovat možnost perzistovat jakákoli data v aplikaci. Naproti tomu tranzitivita u existujících systémů existuje téměř vždy.

[1] Pro *transient*, angl. pomíjivý, přechodný, se nepoužívá český ekvivalent.

[2] V tomto kontextu rozlišujeme data s délkou života do konce aplikace a data určená pro perzistenci, tj. přežívající konec aplikace.

	Ortogonalita	Tranzitivita	Nezávislost
Hibernate	Ne	Ano	Ne
EclipseLink	Ne	Ano	Ne
DataNucleus	Ne	Ano	Ne
PJama	Ano	Ano	Ano

Tabulka 3.1: Tabulka ukazuje vlastnosti nejznámějších systémů pro perzistenci, kterými jsou Hibernate, EclipseLink a DataNucleus. Tyto tři systémy představují profesionální řešení často užívané i v praxi. Případ PJama je zde uveden pouze jako zajímavost, protože se jedná o experimentální projekt a jeho autoři se zaměřili na plnou podporu všech tří principů perzistence [11]. Vývoj PJama a obecně ortogonální perzistence však byl zastaven kvůli příchodu standardu JDO. Převzato z [19].

3.2 Přístupy k perzistenci a jejich vlastnosti

Potřeba uchovávat objekty v Javě stoupla s příchodem enterprise prostředí, ve kterém jsou tvořeny webové aplikace či informační systémy. S tím také souvisí rozvoj perzistence a jejích standardů, kterým se budeme věnovat v kapitole 3.3. Přístupy k perzistenci se liší s nároky na perzistentní prostředí. Je zřejmé, že aplikaci malého rozměru vystačí jednoduché prostředí využívající k ukládání dat soubory, zatímco pro velké aplikace s vícevláknovým přístupem k perzistovaným datům bude třeba robustnější prostředí s ohledem na možné problémy (transakce, paralelní přístup atp.)

V této části práce se budeme zabývat několika nejběžnějšími přístupy, které jsou pro perzistenci objektů využívány.

3.2.1 Serializace

Serializace je konverze objektu do sekvence bytů, popřípadě do jiného formátu, který lze formou souboru perzistovat nebo přenášet (např. po síti). Tento způsob je základní možností, jak uskutečnit perzistenci, a nevyžaduje žádné další nástroje. Serializace je v Javě přítomna od verze 1.1[3] a nevyžaduje téměř žádnou změnu v programovacích návycích.

Základním prvkem serializace v Javě je rozhraní `java.io.Serializable`. Neobsahuje žádnou metodu a indikuje JVM možnost serializace instance třídy, která jej implementuje. Jedinou nutnou podmínkou pro možnost serializace je právě implementace tohoto rozhraní. Serializace přináší ortogonální přístup k perzistenci dat, neboť neexistují data, která by serializace nebyla schopna zpracovat. Serializace je také tranzitivní - v případě, že serializovaný objekt odkazuje na jiný objekt, je referencovaný objekt serializován taktéž. Podmínkou je však i v tomto případě implementace rozhraní `Serializable`.

Výhodou serializace je jednoduchost jejího použití a minimální požadavky na prostředky. Pro aplikace, které potřebují perzistovat jen malé množství dat a nepotřebují je obnovovat (deserializovat) často, je tato cesta zcela dostačující. Avšak v případě, že požadavkem je např. načtení jen části dat z úložiště se stává serializace nevhodným způsobem, neboť systém ukládání při serializaci umožňuje pouze monolitické uložení serializovaných objektů. Nevýhodou je také možná nekompatibilita s dalšími verzemi serializované třídy. Jakákoli změna v definici třídy tak znehodnotí celý záznam vytvořený před její modifikací. [16] Serializace neumožňuje transakce a implementace rozhraní `Serializable` nutí programátora změnit zdrojový kód objektu.

[3] Stejně jako I/O package `java.io` jehož je serializace součástí.

V praxi se serializace používá pro uchování malého množství objektů, které není třeba často nahrávat zpět do paměti. Objekt se všemi svými atributy a asociacemi je uložen jako jednolitá sekvence bytů, což v případě, že programátor má zájem jen o některý z referencovaných objektů, znamená nutnost načíst celou sekvenci zpět do paměti, a až posléze najít požadovaný objekt. V této práce se budeme principem serializace do jisté míry inspirovat, avšak její kritická místa nahradíme jinými přístupy.

3.2.2 Relační databáze

Použití databází při perzistenci dat je časté a v porovnání s použitím souborů přináší především větší robustnost a zlepšení výkonu. Vzhledem k povaze databází je jasné, že data v nich uložená nejsou nijak závislá na běhu aplikace a jsou trvalá. Tento fakt byl motivací pro jejich zapojení do systémů perzistence. Relační databáze jsou postaveny na relačním datovém modelu, který byl vyvinut Edgarem Coddem v rámci jeho činnosti v IBM v roce 1979 [7] a přinesl zvrat v pojetí databází. Zavedl základní stavební prvky relačních databází jako jsou relace, schémata nebo tabulky a ustanovil standard, který se dodnes používá pro implementaci a práci s databázemi. Jeho základ je odvozen z matematického modelu. Naproti tomu objektový model byl vytvořen na základě potřeb programátorů, čímž vzniká mezi těmito dvěma modely neshoda v pojetí základních principů.

Object Relational Mapping

Databáze hrály v perzistenci dat velkou roli již v počátcích vývoje programovacích jazyků. Už v roce 1983 implementoval tým kolem M. P. Atkinsona z University of Edinburgh [4] přístup k perzistentnímu programování formou jazyka PS-algol,[4] o což se snažili právě ukládáním proměnných za pomocí databází. Problémem však zůstává způsob, jak propojit aplikaci s daty určenými k perzistenci a databázi tak, aby nevznikaly problémy spojené s použitím různých paradigmat. Motivace pro uložení dat do databáze a perzistence dat v aplikaci je ve své podstatě podobná. Oba dva přístupy mají za cíl udržet data přístupná po dlouhou dobu a uložit je do nevolatilního prostředí, a proto jsou databáze často základními prvky perzistence a mnoho standardů a technologií je na nich také založeno.

Na perzistenci v Javě mají databáze výrazný vliv a jejich použití je zakotveno v několika standardech (dále v kapitole 3.3). Perzistence objektů do relačních databází však přináší problémy s koncepčními rozdíly mezi relačním a objektovým modelem a tvoří samostatnou disciplínu. Tato netriviální konverze se nazývá *Object-relational mapping*, česky objektově-relační mapování (dále jen ORM), a je podkladem pro většinu profesionálních frameworků.

ORM řeší způsob, jak automaticky konvertovat data mezi relačním a objektovým modelem a skrýt tuto konverzi před programátorem. Zatímco atributy objektů nemusí být skalární a mohou obsahovat další objekty, záznamy v databázi mohou obsahovat pouze jednu hodnotu daného datového typu. Konverze spočívá v namapování objektů na entity v relačním modelu jako je naznačeno v následující ukázce kódu 3.1. [17]

[4] PS-algol je derivát z imperativního programovacího jazyka S-algol z rodiny ALGOL vyvinutý v roce 1979 na University of St Andrews a používaný pro výuku a experimenty.

```
public class TestClass {
    private String text;
    private int number;
}
```

```
CREATE TABLE test_class (
       text VARCHAR(20) NOT NULL,
       number INT NOT NULL
);
```

Výpis 3.1: Definice třídy v Javě (vlevo) a její ekvivalent jako tabulka v SQL (vpravo)

Výše zobrazené mapování je zcela triviální a v tomto případě nepředstavuje konverze třídy na tabulku v databázi žádný problém. Avšak již v tomto příkladě si můžeme všimnout zdánlivého detailu omezení délky řetězce u atributu `text`, protože SQL databáze vyžadují definici maximální délky řetězce (zde např. 20). Tímto problematika mapování pouze začíná, protože v OOP existují koncepty, které v relačním modelu nemají obdoby, a tvoří pak neshody při konverzi modelů. V současné době řeší frameworky postavené na ORM zejména následující problémy. [6]

Granularita Granularita udává míru do jaké může být objektový model rozložen na dílčí části. Může však nastat problém s rozdílem této míry u objektového a relačního modelu. Objektově orientovaný přístup často volí větší míru granularity a přispívá tím k znovupoužitelnosti kódu a vytvoření celků dále přístupných pro ostatní objekty. Avšak u relačního modelu je typicky míra granularity nižší a tím vzniká problém, kdy v objektově orientovaném modelu může být více tříd, než kolik je tabulek v relačním schématu.

Podtypy Jedním ze základních principů v objektově orientovaném modelu je dědičnost tříd, která umožňuje objektům tvořit podtypy a sdílet a rozšiřovat vlastnosti jiných tříd. Tato vlastnost však tvoří pro konverzi objektového modelu na relační velký problém, protože schéma databáze neumožňuje žádnou míru dědičnosti.

Identita V Javě je identita objektů daná adresou. Objekty mají alokovaný prostor v paměti a jejich adresu je možné přiřadit některé proměnné. Takovýmto způsobem je možno referencovat jeden objekt více proměnnými, které jsou tedy identické. Je třeba uvést, že v Javě existuje rozdíl mezi identitou (operátor `==`) a shodností (implementace metody `equals()`), kdy shodnost zaručuje pouze ekvivalenci dvou objektů, nemusí však kontrolovat jejich identitu. V relačním modelu je identita záznamů daná shodností primárního klíče. Zdánlivá podobnost identity dané adresou v OOP a shodností primárního klíče v relačním modelu je zavádějící a kvůli složitějším případům je třeba tyto pojmy rozlišovat. Příkladem může být třeba vícevláknový přístup, kdy různé objekty mohou odkazovat na jeden záznam v databázi pro režii paralelizace.

Asociace Zatímco v objektovém modelu jsou asociace objektů realizovány za pomocí referencí, které ukazují přímo na referencovaný objekt (tj. mají směr), relační databáze namísto toho používají cizí klíče a následná omezení zaručující integritu. Při návrhu schématu databáze se často střetneme s problémem transformace ER diagramu[5] na schéma relační databáze, zvláště pak při přítomnosti vztahů typu `m:n`, kdy je třeba pro validní schéma

[5] ER diagram, z anglického *entity-relationship diagram*, je diagram vytvořený při tvorbě enitity-relationship modelu jako konceptuálního schéma dat.

vytvořit tabulku navíc tak, aby všechny vztahy byly typu `1:n` nebo `1:1`. Podobný problém vzniká při ORM, kdy se snažíme uložit objekt obsahující vazbu typu `m:n` (tj. objekt odkazuje na jiný objekt, který však může obsahovat vazbu na původní objekt).

Přístup k datům Přístup k datům se v obou konceptech výrazně liší. Java umožňuje přistupovat k datům přímo skrze atributy nebo metody. Mějme příklad objektu zpřístupněného skrze proměnnou `test`, která obsahuje několik dalších referencí na jiné objekty. Konstrukce pro získání jedné z nich by tedy vypadala následovně: `test.getItem().getItem2()`, přičemž následujeme ukazatele do paměti až dojdeme k požadovaným datům. Tento princip je však v relačním modelu nemyslitelný a pro ekvivalentní chování je třeba použít operaci `join`, jehož vykonání je velmi nákladné na režii, nebo více operací `select`, což přináší podobný problém.

ORM představuje robustní princip, který se do hloubky zabývá neshody v konceptech a snaží se je řešit různými cestami. Frameworky, které jsou na ORM postaveny, potom přicházejí se samotnými implementacemi, které jsou přizpůsobeny specifickým potřebám a výkonu.

Ve zkratce můžeme říci, že ORM se snaží nabídnout prostředek pro perzistenci všech tříd definovaných programátorem dodržující jistá pravidla. Avšak problémy popsané výše jasně ukazují, že tento přístup vyžaduje dobrou znalost relačního modelu, a zatíží programátora nutností myslet na problémy spojené s ORM. Mapování je také důležitým pojmem, neboť jeho použití většinou implikuje právě nutnost se zamyslet nad samotnou konverzí a nespoléhat na blackboxing[6] nabízený moderními frameworky. ORM je oblast, o které by měl mít povědomí každý programátor pracující se standardy pro perzistenci jako je Java Persistence API a jejími implementacemi.

3.2.3 Objektové databáze

Objektově orientovaný systém řízení báze dat[7] je takový databázový systém, který pracuje na základě objektového modelu, tj. ukládá informace ve formě objektů. Uplatňuje stejné principy jako objektově orientované programovací jazyky. Podporuje definici složitých objektů, jejich manipulaci a celkově všechny principy objektově orientovaného přístupu jako jsou polymorfismus, dědičnost nebo zapouzdření. Tyto vlastnosti jsou často využívány pro implementaci perzistence do objektově orientovaných jazyků, protože při ukládání objektů nedochází k žádným nesouladům v modelech, jako je tomu třeba u ORM.

Objektové databáze vytváří dobré podmínky pro přímou integraci perzistence do objektově orientovaných jazyků. Objekt je v objektové databázi uložen stejným způsobem jako v paměti, čímž odpadá problém většiny předchozích přístupů, které často řeší neshody modelů a jejich konverzi. Objektové databáze podporují abstraktní datové typy a operace nad nimi, čímž se značně odlišují od relačních databází, které typicky podporují jen primitivní datové typy.[8]

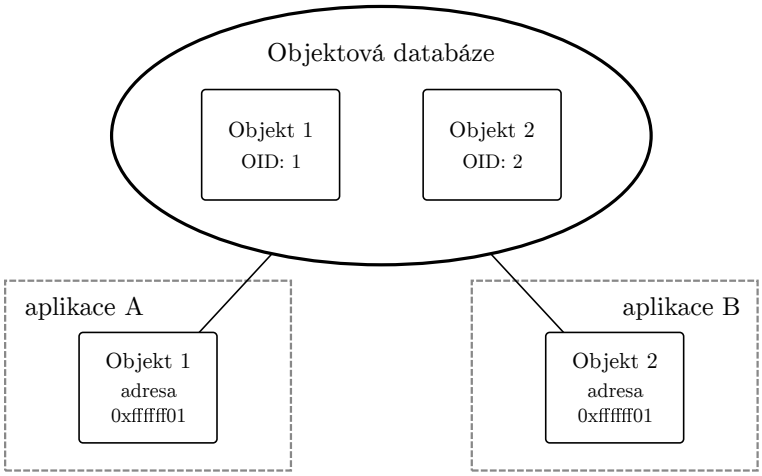
Základním rozdílem mezi systémem ukládání do paměti a do objektové databáze je nutnost identifikace objektů. V paměti má každý objekt svou adresu, což v rámci běhu aplikace stačí k jednoznačné identifikaci. Avšak po uložení objektu do databáze je nutné tomuto objektu přiřadit jiný jednoznačný identifikátor, který bude unikátní v rámci perzistentního

[6] Blackboxing je anglický pojem používaný pro označení funkce se známými vstupy a výstupy, ale se skrytou implementací.

[7] Odvozeno z anglického *Object-Oriented Database Management System*.

[8] V SQLv3 existuje podpora pro tvorbu abstraktních datových typů, avšak jejich užití není příliš rozšířené.

systému. Tento identifikátor se ve většině objektových databází nazývá *Object Identifier*, zkráceně OID, a je přiřazen každému nově uloženému objektu bez ohledu na jeho zdroj.[9] V obrázku 3.1 můžete vidět schéma perzistovaných objektů. Aplikace *aplikace A* a *aplikace B* běží na rozdílných JVM, a jejich adresové prostory jsou tedy odlišné. Objekt *Objekt 1* má stejnou hodnotu adresy jako *Objekt 2*, ale adresy ukazují na jiné objekty. To by v případě nepřítomnosti OID představovalo konflikt a právě proto objektové databáze zavádí OID pro zcela deterministickou identifikaci objektů.



Obrázek 3.1: Uložení objektů do objektové databáze

V jazyce Java nemají objektové databáze nativní podporu a tím pádem není možné přímo ukládat objekty do objektových databází. Avšak Java Data Object API popsané v kapitole 3.3.2 nabízí jako jeden z typů úložiště právě objektovou databázi a umožňuje tak perzistenci objektů bez problémů s mapováním či koncepčními rozdíly.

3.2.4 Cíl práce

Náš cíl je kombinací prvků z popsaných přístupů. Systém může využít lehkost principu serializace co se týče náročnosti na doplňující nástroje jako jsou databáze. Rozdělení objektů do částí podle jejich tříd je analogií tabulek z relačního modelu a ORM, které zavede vyšší granularitu výsledného systému perzistence, jež chybí u serializace. Avšak výhody relačních databází nejsou pro náš případ tolik podstatné, neboť požadavek na nezávislost na ostatních nástrojích a celková robustnost systému nemá potenciál je využit. Důležitým prvkem zavedeným v objektových databázích je potom celková kompatibilita s objektově orientovaným systémem a zavedení OID. I v tomto případě ale náš systém převezme pouze část konceptu a oblasti jako jsou transakce či dotazování objektů vynechá.

3.3 Standardy perzistence jazyka Java

K perzistenci slouží v Javě přímo či nepřímo několik standardů a API. V této podkapitole si představíme některé z nich. Většina používá k perzistenci již popsané ORM, které představuje nejrobustnější a nejrozšířenější možnost. Objektové databáze sice v porovnání s

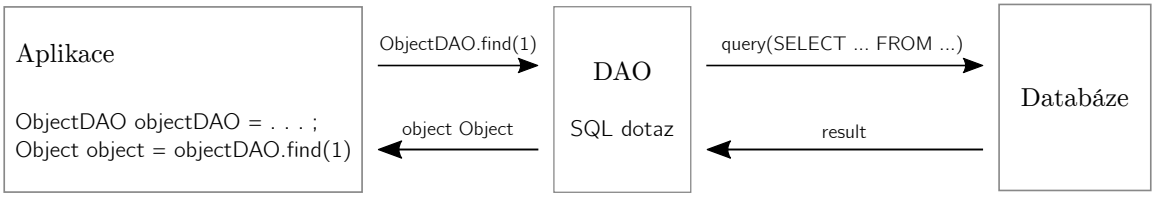
[9] OID je obdoba primárního klíče v relačních databázích.

relačními databázemi ušetří mnoho problémů, avšak vzhledem k pokročilejším technologiím a větší zkušenosti programátorů s relačním modelem je tato možnost stále ojedinělá.

3.3.1 Java Database Connectivity

Java Database Connectivity API (dále jen JDBC) je API, které nabízí třídy a rozhraní pro komunikaci s databázemi. Tvoří součást Javy již od prvních verzí a představuje základ pro všechny standardy perzistence pracující s ORM a s databázemi obecně. Hlavními funkcemi JDBC je navázání komunikace s databází, dotazování a zpracování výsledků. Neméně důležitá je přímá podpora transakcí. [8]

Pro částečně oddělení programátora od SQL příkazů se zejména v enterprise aplikacích často objevují tzv. Data Access Object (dále jen DAO). [3] Ty skrývají před programátorem samotné SQL příkazy a vytváří API pro zápis a čtení dat z databáze jak je znázorněno v obrázku 3.2.



Obrázek 3.2: Funkce DAO

Persistence přímo přes JDBC je většinou specifická pro danou aplikaci a definici tříd, se kterými pracuje. Její výhodou je oproti univerzálním systémům proto rychlost, kde je aplikovatelnost kompenzována vyšší režií pro načítání a ukládání objektů. (např. reflexe). Naproti tomu míra abstrakce bývá v tomto případě omezená.

3.3.2 Java Data Object API

Java Data Object (dále jen JDO) je API pro perzistenci objektů v Java Enterprise Edition. Pro perzistenci používá JDO obdobný princip abstrakce jako JDBC, a to práci s úložištěm, tvorbu dotazů a získávání objektů. Na rozdíl od JDBC však přidává možnost zobecnění typu úložiště a nerozlišuje tak mezi dotazem pro soubory XML nebo pro relační databáze. Hlavním přínosem JDO je zejména *transparentní perzistence*. Tento pojem je ekvivalentem pro *nezávislou perzistenci* popsanou v kapitole 3.1, a tudíž přináší možnost perzistence bez nutnosti přidávat metody nebo měnit viditelnost atributů na úrovni zdrojového kódu. [15]

Pro rozpoznání objektů určených k perzistenci používá JDO proces zvaný *bytecode enhancement*[10]. Ten umožňuje v době překladu či za běhu aplikace modifikovat `.class` soubory a tedy také přidání metod pro přístup k objektům využívané implementací JDO. Tímto způsobem je možno oprostit programátora od nutnosti implementovat `public` metody či jinak upravovat třídu pro perzistenci, a umožnit tak transparentnost perzistence.

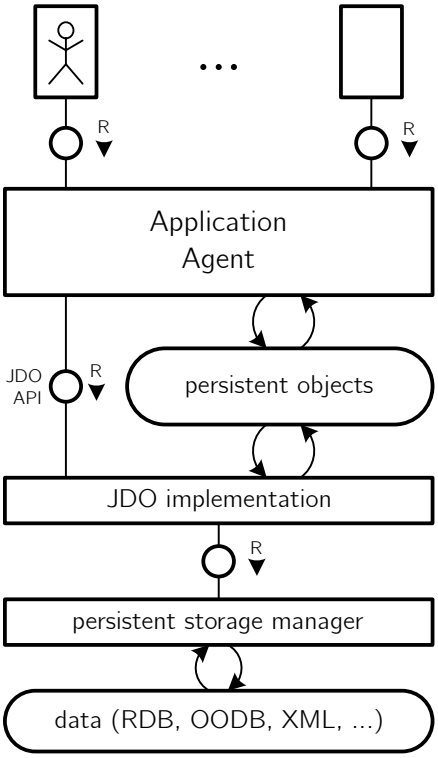
Postup pro perzistenci je následující:

1. Označit perzistované třídy anotací `@PersistenceCapable` pro indikaci perzistence
2. Přeložit třídy aplikace do bajtkódu, tzn. do `.class` souborů.
3. Provést *bytecode enhancement* za použití dodaného postprocesoru.

[10] Český překlad pro *bytecode enhancement* se nepoužívá, doslova "vylepšení bajtkódu".

4. Perzistovat a získávat objekty pomocí metod rozhraní `PersistenceManager`

Způsob použití JDO je nastíněn v obrázku 3.3. V aplikaci (v obrázku znázorněné jako *Application Agent*) se nacházejí objekty s anotací `@PersistenceCapable`, čímž se z nich stávají perzistované objekty (*persistent objects*). Následně s těmito objekty pracuje implementace JDO, která s nimi manipuluje pomocí metod přidaných při *bytecode enhancementu*.



Obrázek 3.3: Struktura aplikace používající JDO. [12]

Standard JDO je méně užívaný než níže popsaný konkurenční přístup JPA popsaný v kapitole 3.3.4. *Bytecode enhancement* sice lehce prodlužuje dobu překladu, avšak následně zcela předchází užití reflexe, což vzhledem k její režii značně přispívá k výkonu.

3.3.3 Enterprise Java Bean 2.0

Enterprise Java Bean (dále jen EJB) je jedno z mnoha API[11] jazyka Java, které je zaměřeno na konstrukci distribuovaných enterprise aplikací. Základ tvoří *JavaBean*, což je třída, která představuje znovupoužitelnou programovou komponentu a která je vytvořena v souladu se specifikací JavaBeans API vyžadující následující vlastnosti: [18]

- nabízí veřejný konstruktor bez argumentů
- implementuje rozhraní `Serializable` pro účely serializace
- může mít atributy, tzv. *properties*, které mohou být modifikovány
- *properites* jsou přístupny skrze *getter* a *setter* metody

[11] Ekvivalentem může být dnes často používaný Spring.

Všechny tyto vlastnosti vedou k principu univerzálního objektu, jehož funkce jsou nezávislé na platformě a jehož chování lze následně specifikovat modifikací patřičných *properties*. Standard EJB přišel s typem objektu označovaným jako *EntityBean*, což je objekt umožňující modelování dat vhodných k perzistenci. Mimo *EntityBean* EJB přináší také *SessionBean*, které reprezentují neperzistentní serverovou komponentu a *MessageDrivenBean* sloužící pro zpracovávání asynchronních zpráv.

Předmětem této práce není podrobné zkoumání principů představených standardem EJB, neboť ten definuje komplexní přístup k tvorbě enterprise aplikací a je samonosným tématem. Důležitý je však pro nás důraz kladený na perzistenci, který byl v EJB představen.

V rámci verze 2.0 standardu EJB byla představena možnost perzistence *EntityBean* pomocí dvou způsobů: [13]

Perzistence spravovaná kontejnerem je technologie,[12] nabízející perzistenci entit řízenou EJB konteinerem,[13] který automaticky zpracovává provedené změny *EntityBean* a reflektuje je do perzistentního prostředí. Atributy jednotlivých perzistovaných objektů rozděluje na *container-managed perzistent fields* indikující pole přímo určené pro perzistenci a *container-manager relationship fields* označující referencované objekty. Oba dva typy perzistovaných dat mají různé rozhraní a různé přístupy k jejich zpracování.

V ukázce 3.2 je zobrazena definice *EntityBean* v podobě XML souboru, který pomocí tagů `<perzistence-type>` určuje typ perzistence - v tomto případě právě perzistence skrze kontejner. Další tagy určují např. třídu primárního klíče nebo jednotlivá pole (`cmp-field`).

```
<ejb-jar>
 <enterprise-beans>
   <entity>
      <ejb-name>TestEntiyBean</ejb-name>
      <home>com.test.TestBean</home>
      <remote>testclasses.Test</remote>
      <ejb-class>com.test.TestBean</ejb-class>
      <perzistence-type>Container</perzistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>false</reentrant>

      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>test1</field-name></cmp-field>
      <cmp-field><field-name>number</field-name></cmp-field>
      <cmp-field><field-name>text</field-name></cmp-field>
   </entity>
 </enterprise-beans>}
```

Výpis 3.2: Ukázka definice *EntityBean* pro EJB kontejner.

Perzistence spravovaná beany je jednodušší svým přístupem, ale složitější implementací. Objekty musí obsahovat *call-back* metody `ejbLoad` a `ejbStore`, které implementují

[12] Používanější je anglické označení *Container-managed Persistence*.

[13] EJB kontejner poskytuje prostředí pro běh *EJB* na aplikačním serveru a spravuje nastavení a přístup k nim.

veškerou logiku pro uložení objektu do perzistentního prostředí a zpětného nahrání např. použitím JDBC. Tento způsob se v praxi příliš nepoužívá z důvodu nízké univerzálnosti kódu.

3.3.4 Enterprise Java Beans 3.0 a Java Persistence API

Ve verzi EJB 3.0, která byla součástí převratné Java EE 5 Specification, došlo k výrazné změně ve specifikaci perzistence pro EJB. Týmem vývojářů EJB3 bylo vytvořeno nové Java Persistence API (dále jen JPA) jakožto samostatná větev pro propojení Javy a relačních databází za použití ORM. EJB se tím pádem nově mohla začít soustředit pouze na část mezi JPA a *entity beany*. Pro tuto práci je důležitá zejména specifikace JPA, a proto se dále integrací JPA v EJB3 nebudeme věnovat.

Java Persistence API je standard pro perzistenci využívající ORM. Je dostupné pro Java EE i pro Java SE jako samostatný celek a definuje rozhraní pro implementaci perzistence. JPA přímo nepracuje s *EntityBeany*, ale do velké míry jejich funkci přebírá. Neobsahuje žádné implementující třídy, avšak nabízí programátorovi prostředky pro vytvoření vlastní perzistenční vrstvy pro popis mapování objektů na relační model a práci s nimi, a to za užití anotací nebo XML souborů. Základní jednotkou pro JPA je entita, což je objekt z dané domény, jehož stav je třeba zachovat. Avšak vzhledem k objektovému modelu je důležitá hlavně třída této entity, která definuje atributy a její metody. Právě tato třída je pro JPA klíčovým prvkem, neboť při její tvorbě programátor určí, zda-li budou všechny její instance součástí perzistentního systému, a to bez toho, aby si toho samotné instance byly vědomy. Zde můžeme rozpoznat princip *transparentní perzistence* uvedené v kapitole 3.1, který se velmi liší od principu perzistence spravované beany, kde se o perzistenci starají samotné instance.

Zásadní podmínkou pro možnost perzistence skrze JPA je splnění vlastností POJO[14], tedy v podstatě stejných vlastností jako JavaBean [14]. Jediným, a to velmi nepodstatným rozdílem, je nutnost implementace rozhraní `Serializable` pro JavaBeans, a vzhledem k povaze tohoto rozhraní, které neimplementuje žádné metody a plní pouze informativní funkci, je možno tento rozdíl zanedbat.

JPA nabízí dva způsoby, jak určit, která entita bude perzistována, a jak nastavit mapování objektů na relační model. První způsob je pomocí anotací, které programátor používá přímo při tvorbě tříd. Druhá možnost je potom externí definice za použití XML souborů. Obě dvě možnosti jsou sémanticky ekvivalentní, avšak existují různé motivace pro každou z nich. Použití anotací je často používané pro svou názornost a dobrou čitelnost kódu, zatímco XML nabízí lepší řešení v případě velkých projektů, kde by možná modifikace parametrů perzistence přímo v definici tříd mohla být zdlouhavá. Princip dvojího způsobu definice vznikl s příchodem Java EE 5 a v ní představených anotací a používají ho i implementace, jako je Hibernate. V této práci budeme používat především anotace, neboť jsou vhodnější pro svou názornost. Následuje demonstrace definice entity a práce s nimi, na které objasníme principy představené JPA. [2]

Použití ORM

Z ukázky 3.3 je zřejmé, že jedná o třídu `Employee` modelující zaměstnance, která obsahuje jeho oddělení `department` a množinu projektů `projects`. Celá entita je uvozena anotací

[14] *Plain Old Java Object* je pojem označující základní objekt v Javě, nabízející bezargumentový konstruktor, přístup k atributům skrze *gettery* a *settery* a možnost serializace [9].

@Entity, která označuje objekty dané domény určené k perzistenci skrze ORM. Anotace @Table mapuje danou třídu na jméno tabulky v relační databázi. Tato anotace obsahuje další modifikovatelné vlastnosti jako schema nebo catalog pro upřesnění umístění tabulky a relačního schématu. Anotace @Id potom indikuje primární klíč tabulky v databázi.

```
@Entity
@Table(name="EMPLOYEE")
public class Employee {

    @Id
    private Long id;

    @OneToMany(mappedBy="department") // vlastnik vztahu je trida Employee
    private Department department;

    @ManyToMany
    private Set<Project> projects;
}
```

```
@Entity
@Table(name="DEPARTMENT")
public class Department {

    @Id
    private Long id;

    @ManyToOne
    private Set<Employee> employees;
}
```

Výpis 3.3: Definice třídy s anotacemi pro JPA.

Zmíněné anotace pro definici entit a jejich primárních klíčů se příliš zvlášť za použití XML konfigurace příliš neliší od způsobu definování těchto informací v EJB 2.0. Na rozdíl od EJB 2.0 však JPA přidalo transparentní způsob pro definici vztahů mezi objekty a jejich mapování za použití ORM. EJB 2.0 umožňovalo reference pomocí XML tagu <ejb-ref> v EJB deskriptoru, který však byl zaměřen spíše na obecný princip EJB, než-li na transparentní konfiguraci závislých entit. Proto JPA přineslo několik typů referencí:

- @OneToOne značí atribut, který referencuje jiný objekt a jedná se o vztah 1:1. Obousměrná vazba je docílená anotací na obou dvou stranách vazby, tzn. i odkazovaný i odkazující objekt obsahuje anotaci @OneToOne. Pro objekt, který nevlastní referenci je nutné definovat pomocí elementu mappedBy vlastníka tohoto vztahu.
- @OneToMany anotuje atribut, který odkazuje na objekt referencovaný více objekty, tzn. vztah 1:m. Stejně jako u @OneToOne je třeba definovat vlastníka vztahu elementem mappedBy na straně, která vztah nevlastní. V našem případě by se tak jednalo o anotování atributu seznamu zaměstnanců v oddělení anotací @OneToMany s parametrem

`mappedBy="departement"` určující vlastníka vztahu jako atribut `departement` třídy `Employee`.

- `@ManyToMany` je nejsložitějším případem vztahů mezi dvěma objekty. Implikuje nutnost vytvoření pomocné tabulky jako řešení `m:n` vztahů v relačním modelu. JPA určuje, že název pomocné tabulky má formát `<owner>_<referenced>`, kde `<owner>` je jméno vlastnící třídy a `<referenced>` je jméno referencované třídy. Tato tabulka obsahuje cizí klíče obou entit.

Operace s entitami

Základním prvkem pro práci s entitami je `EntityManager`, který přímo komunikuje s jádrem perzistence, tedy s perzistenčním kontextem. Ten obsahuje pro každou perzistovanou entitu unikátní instanci a stará se o jejich životní cyklus. `EntityManager` poskytuje pro přístup k instancím entit a ke kontextu několik metod, z nichž některé nejznámější a důležité pro tuto práci jsou:

- `public void persist(Object entity);`
  - metoda pro perzistování entity
- `public <T> T merge(T entity);`
  - metoda pro zanesení úprav entity do instance v perzistenčním kontextu
- `public void remove(Object entity);`
  - metoda pro odstraněni entity z perzistence
- `public <T> T find(Class<T> entityClass, Object primaryKey);`
  - metoda pro načtení instance entity z perzistenčního kontextu

Mnoho problémů spojených s ORM (vizte kapitolu 3.2.2) je pro začínající programátory v oblasti perzistence s JPA skryto nebo vyřešeno šikovným výchozím nastavením, což umožňuje vcelku rychlé zprovoznění dobře fungujícího perzistenčního systému. Pro větší projekty s více specifickými vlastnostmi však tento systém do jisté míry připomíná blackbox a programátor nemá nad některými částmi kontrolu. Příkladem můžou být vztahy `m:n`, tj. ty, které jsou anotované `@ManyToMany`. Takto uvozené vazby jsou dle konvencí pro relační model převedeny na vztahy mezi třemi tabulkami, což je sice specifikováno ve standardu JPA, avšak znalost tohoto problému není pro vyřešení této závislosti vyžadována. Systém tento převod sám vyřeší, čímž však tuto důležitou část skrývá.

3.4 Existující systémy a jejich porovnání

V této části se budeme věnovat představení několika nejužívanějších systémů pro perzistenci v Javě. Vzhledem k tomu, že požadavky na robustní systém perzistence je především doménou enterprise aplikací, tak všechny níže popsané frameworky implementují a doplňují standardy Java EE. Většinu z nich lze také použít pro desktopové aplikace, avšak jejich užití v nich je často spojeno s neúměrnými požadavky na dodatečné zdroje, tedy databáze.

3.4.1 Hibernate

Hibernate je open source projekt, jehož hlavním cílem je vývoj frameworku pro jazyk Java nabízející perzistenci za pomoci objektově relačního mapování. Projekt Hibernate začal v roce 2001 a nyní jej vyvíjí společnost Red Hat. Mezi nejdůležitější milníky ve vývoji Hibernate můžeme řadit verzi 3.5.0, kdy se Hibernate implementoval JPA.

Hibernate tedy nabízí standardizovaný přístup skrze JPA obohacený o několik proprietárních funkcí, které však již nejsou kompatibilní se zmíněným standardem. Hlavním odklonem od standardu je přítomnost třídy `Session`, která zaobaluje funkce rozhraní `EntityManager` a přidává další funkce, jako je třeba podpora *multitenancy*, tedy možnost nabízet více klientům použití jednoho zdroje. Ve zkratce je tedy možno Hibernate použít jako perzistenční systém pro více aplikací, tedy klientů. Nabízí tři možnosti: a) dedikované databáze pro každého klienta nebo aplikaci, b) rozdílné schéma nebo c) sdílené tabulky pro všechny klienty s rozdílným *tenantID*[15] v záznamu. [1]

Další zajímavou funkcí je Hibernate Enverse, která nabízí verzování entit a přístup k jejich předchozím verzím. Hibernate také uvedl podporu pro full-textové vyhledávání ve frameworku Hibernate Search, který je postaven na knihovně Apache Lucene.[16] V Hibernate existuje také podpora pro ukládání dat do NoSQL databází, která je uvedena v části Hibernate OGM, a umí pracovat např. s dokumentovými databázemi jako je MongoDB nebo grafovými databázemi typu Neo4j.

Hibernate je jedním z nejužívanějších systému pro perzistenci. Nabízí velmi robustní implementaci, která je kompatibilní s dalšími standardy a frameworky jako jsou EJB či Spring, a přináší univerzální možnost perzistence. Tento fakt však může být také nevýhodou, neboť Hibernate často skrývá spoustu implementačních detailů důležitých při práci s perzistencí, a programátor tak nemusí do hloubky rozumět akcím odehrávajícím se při jejím samotném vykonávání. Tato vysoká míra abstrakce může vést k nepochopení a dále mít dopady na výkon.

3.4.2 EclipseLink

EclipseLink je open-source projekt vyvíjený společností *Eclipse Foundation*, který se stejně jako Hibernate zaměřuje na perzistenční framework. Na rozdíl do Hibernate je však EclipseLink referenční implementací JPA. Oproti Hibernate také nabízí možnost mapování tříd Java na XML schémata, čemuž se věnuje část *MOXy*, která mimo jiné implementuje standard Java Architecture for XML Binding (dále jen JAXB). Dále je možné pomocí EclipseLink perzistovat data do NoSQL databází, a to stále s použitím JPA. Stejně jako Hibernate, i EclipseLink uvádí podporu pro více-klientské přístupy.

Obecně je EclipseLink považováno za implementaci bližší k JPA a to taky kvůli role referenční implementace. V profesionálním prostředí je však často užívanější Hibernate, a to kvůli jeho podrobnější dokumentaci a širší komunitě aktivních uživatelů.

3.4.3 DataNucleus

DataNucleus je také open-source projekt zaměřující se na perzistenci dat a jejich zpětné nahrávání do paměti. V porovnání s předchozími dvěma projekty však nabízí mnohem

[15] Číslo *tenantID* je v případě Hibernate identifikátor klienta.

[16] Apache Lucene je knihovna pro podporu full-textového vyhledávání v Javě vyvíjená společností Apache.

univerzálnější přístup co se týče typu úložiště. Celý projekt je v rozdělen na implementace dvou různých standardů, kterými jsou JDO[17] a JPA.

V případě JPA je možné ukládat data pouze do relačních databází. Ačkoli Hibernate a EclipseLink nabízejí možnosti, jak JPA využít pro NoSQL databáze, samotná JPA tuto možnost přímo nepodporuje. Naproti tomu JDO podporuje v podstatě jakýkoli typ úložiště. Největší rozdíl mezi těmito dvěma standardy je především přítomnost interní reprezentace objektů v JDO a jazyk JDOQL, který nabízí univerzální cestu pro dotazování perzistovaných objektů a který je nezávislý na úložišti.

Jak již bylo zmíněno v kapitole 3.3.2, s JDO mizí nutnost reflexe objektů, která je náročná na výkon, a proto JDO představuje konkurence schopnou alternativu k více rozšířenému JPA.

3.4.4 Shrnutí a porovnání cílem práce

Hibernate a EclipseLink jsou si velmi podobné a důvody k užití jednoho nebo druhého systému jsou často otázkou preference a návyků programátora. Pro DataNucleus je však situace odlišná. V případě nevhodnosti užití relačních databází se vývojáři mohou přiklonit právě k implementaci JDO. Pravdou však zůstává, že DataNucleus není zdaleka tolik rozšířený jako Hibernate, a tím pádem je i jeho komunita menší a nabízí menší podporu.

Systém vyvíjený v rámci této práce převážně pobírá inspiraci z Hibernate. Dokazuje to použití reflexe, nepřítomnost *bytecode enhancement* a ekvivalent třídy `EntityManager`. Avšak úložiště ve formě XML souborů a ukládání objektů bez použití mapování jsou přístupy používané spíše JDO a DataNucleus. Náš systém tedy kombinuje tyto dva existující frameworky a vybírá si ty vlastnosti, které lépe odpovídají požadavkům.

[17] Pro JDO je DataNucleus referenční implementací.

Kapitola 4

Návrh a implementace

V této kapitole se zaměříme na návrh vyvíjeného systému a jeho implementaci. Fáze návrhu tvořila důležitou část celého projektu, neboť problematika perzistence objektů je netriviální a návrh bylo třeba domyslet do co možná nejmenších detailů ještě před začátkem samotné implementace.

4.1 Analýza požadavků

Vzhledem k tomu, že požadovaný systém má nabízet perzistenci pro objekty Petriho sítí v rámci simulace, bylo třeba zvolit minimalistický přístup pro zachování účelu simulace bez nutnosti věnovat přílišnou pozornost problematice perzistence. To také koresponduje s požadavkem, aby nebyly využity žádné dodatečné nástroje jako např. databáze. Motivace pro perzistenci objektů Petriho sítí je potřeba spouštět simulaci v určitých intervalech, během kterých je však aplikace pozastavena, a je tedy třeba po tuto dobu zachovat stav objektů. Cílem je perzistovat předem určenou množinu objektů a následně mít možnost ji znovu nahrát do operační paměti, a tím obnovit původní stav testování. Tento fakt do jisté míry snižuje nároky vyhledávání objektů[1] podle kritérií (jako např. přistupuje k načítání Hibernate), a velmi zjednodušuje samotnou implementaci.

4.2 Východiska návrhu

Cílem tedy bylo implementovat odlehčený perzistenční framework pro jazyk Java bez užití dalších nástrojů. Tento fakt společně s výše popsanými požadavky implikuje dvě základní vlastnosti.

První vlastnost se týká především způsobu používání perzistence. S ohledem na požadavky byl systém navržen tak, aby splňoval vlastnosti transparentní perzistence, tedy takové perzistence, která nezatěžuje programátora změnou v programovacích návycích. Tento přístup se osvědčil u všech nyní používaných frameworků a tato práce ho přejímá taktéž. Důvodem je požadavek nízkých nároků na režii perzistence, s čímž koresponduje i absence potřeby měnit programátorský styl zahrnující například např. nezbytnou implementaci některého z rozhraní jako tomu bylo u EJB2. Tato vlastnost je základem našeho systému a jejím prostřednictvím byla dosažena vysoká míra nezávislosti aplikačního kódu na systému perzistence.

[1] I přesto však náš systém disponuje robustním XML modelem, který je připraven na možnost dotazování.

Dalším faktorem pro návrh našeho systému byl systém pro ukládání dat, což velkou mírou ovlivnilo zejména výstup frameworku, tedy objekty uložené v definovaném formátu na perzistentní úložiště. Po vyloučení ostatních nástrojů, a to zejména databází, jsme se uchýlili k nejjednoduššímu a nejpřístupnějšímu řešení inspirovaným serializací (vizte kapitolu 3.2.1), a to k ukládání objektů do lokálního souborového systému. Nevýhoda popsané ve zmíněné kapitole spojené s typem úložiště, tedy nepřítomnost transakcí, je velkým úskalím našeho systému, avšak pro dané užití je riziko spojené s jejich nepřítomností přijatelné. Systém není navržen pro komplexní užití v rámci enterprise aplikací, kde transakce hrají důležitou roli pro bezpečnost dat.

Důležitou inspirací byly také principy objektových databází (vizte kapitolu 3.2.3), které umožní se vyhnout neshodám v modelech, jako je tomu např. u ORM. Tato myšlenka hrála při návrhu důležitou roli a umožnila systému se oprostit od mapování. Cílem je v podstatě uložit obraz vybraných objektů v paměti JVM, avšak na rozdíl od serializace nabídnout vysokou granularitu.

4.3 Architektura tříd a rozhraní

Architektura frameworku se inspiruje architekturou definovanou ve standardu JPA, ale také jeho implementací v Hibernate. Základní princip je jako u ostatních frameworků postaven na návrhovém vzoru *Factory*, který mimo jiné zapouzdřuje inicializace složitých objektů. Náš systém, podobně jako EclipseLink či Hibernate, takto vytváří perzistenční jádro. Jako výchozí bod pro jeho tvorbu slouží vlastnosti definované v *properties*. Existují dvě možnosti jak definovat parametry pro inicializaci perzistence. První vede skrze třídu nastavení, která je vytvořena na základě návrhového vzoru *Builder* jak je nastíněno v ukázce 4.1. Druhou možností je definice parametrů v externím XML souboru.

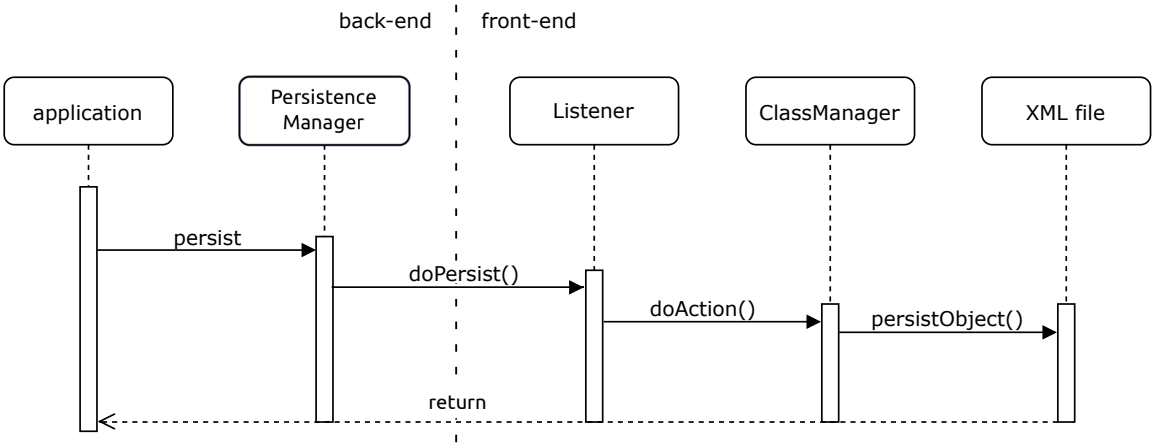
```
PersistenceSettings settings = new PersistenceSettings()
   .setRootPath("some/path/to/store/objects");
PersistenceManagerFactory pmFactory;
try {
   PersistenceManagerFactoryBuilder builder =
          new PersistenceManagerFactoryBuilder(settings);
   pmFactory = builder.buildPersistenceManagerFactory();
} catch (PersistenceCoreException ex) {
   ex.printStackTrace();
}

PersistenceManager pm = pmFactory.getPersistenceManager();
```

Výpis 4.1: Ukázka inicializace jádra perzistence.

Rozhraní `PersistenceManager` je svou podstatou velmi podobné rozhraní `EntityManager` uvedené v JPA. Je vytvářen jádrem perzistence a vytváří rozhraní pro přístup k perzistovaným objektům pomocí metod pro perzistování, modifikaci nebo načtení objektů zpět do paměti.

4.4 Jádro perzistence

Jádro perzistence představuje třída `PersistenceContext`, která spojuje všechny části systému, tedy front-endové prvky spouštějící události s back-endovými částmi pro uložení dat objektů do souborů s přispěním třídy určené pro management vstupů a výstupů, tedy XML souborů s daty objektů. Jádro obsahuje třídu pro práci se soubory, hash mapu se všemi komponenty pro jednotlivé perzistované třídy, a také cache rozpracovaných objektů, která je využívaná při kaskádovém načítání objektů zpět do paměti.



Obrázek 4.1: Sekvenční diagram průběhu perzistence.

4.5 Strategie ukládání

Perzistenční systém ukládá data objektů do souborů XML v lokálním souborovém systému. Avšak systém se nesnaží mapovat objektový model na odpovídající model DOM,[2] nýbrž využívá XML jako vrstvu, která nese strukturu perzistentního prostředí. V rámci vývoje byla struktura XML dokumentu definována pomocí DTD,[3] která zajišťuje univerzálnost uložených dat a možnost jejich přenesení do jiných systémů.

Objekty jsou ukládány podobným způsobem jako v případě JVM, která pro ukládání objektů používá hromadu a referencuje instance pomocí jejich adresy. Přeneseně byl tento způsob použit i v našem systému, kde však unikátnost objektu v podobě adresy nahradilo *object ID* a funkci pomyslné hromady tvoří soubor XML. Tento přístup nám umožňuje uložit stavy objektů s jakkoli složitými asociacemi bez nutnosti řešit neshody v modelech a jejich vzájemném mapování.

Ve stávající verzi budou v složce přítomny soubory odpovídající uživatelsky definovaným třídám jak je naznačeno v ukázce 4.2. Navíc zde budou soubory obsahující referencované pole, kolekce a mapy. Tyto tři třídy jsou zvláštním případem, neboť jsou to třídy Java SE. Jejich instance jsou samostatně existující objekty v paměti, ale programátor nemá přístup k definici jejich třídy a tím pádem nemůže definovat unikátní *objectId* jakožto jediný unikátní identifikátor v systému a proto nepodporujeme přímou perzistenci těchto tříd a jejich nepřímé perzistenci se budeme věnovat v kapitole 4.8.

[2] *Document Object Model* je objektový model XML souborů specifikovaný společnosti W3C pro standardizaci přístupu k XML.

[3] *Document Type Definition* umožňuje popisovat strukturu značek a atributů v daném XML dokumentu.

```
persisted_objects ..... root složka
├── java.util.Collection.xml ..... soubor kolekcí
├── java.util.Map.xml ..... soubor map
├── [Ljava.lang.Object;.xml ..... soubor polí
└── Test.xml ..... soubor uživatelské třídy
```

Výpis 4.2: Struktura složky s perzistovanými objekty.

4.6 Události

Rozhraní `PersistenceManager` komunikuje s perzistenčním jádrem na základě událostí, které generuje. Nyní systém podporuje tři typy událostí, které se pojí se třemi možnostmi pro manipulaci s objekty.

Událost `persist` První typ události je `persist`, tedy samotné předání objektu perzistenci. Ta spustí celý proces perzistence, vytvoří záznam v interním XML modelu a spustí aktualizaci XML souboru v souborovém systému.

Událost `load` Událost typu `load` slouží k načtení objektu z perzistentního prostředí zpět do paměti na základě jeho *objectId*,což implikuje možnost načítat pouze ty objekty, které jím disponují. Tedy kolekce, mapy a pole bez *objecId* nelze samostatně načíst.

Událost `update` Posledním zatím podporovaným typem události je událost `update`. Její funkcí je modifikace již perzistovaného objektu a příliš se neliší od události `persist`. Avšak v případě, že programátor žádá systém o modifikaci objektu, který není součástí perzistentního prostředí, je tato operace považována za chybovou a systém indikuje chybu výjimkou.

4.7 Třída StorageContext

Třída `StorageContext` je prostředníkem mezi jádrem perzistence a složkou, kde se perzistované objekty uchovávají. V aktuální verzi systému tato třída poskytuje pouze omezené možnosti pro práci se soubory, jako je vytvoření streamů pro načítání a ukládání souborů, a skenování složky určené pro soubory XML. Třída však by však v budoucnu mohla obsahovat také podporu pro transakce nebo jiné typy úložiště, např. JSON nebo binární reprezentaci.

4.8 Třída ClassManager

Každá třída, která je určená k perzistenci, má v našem systému tzv. `ClassManager`, což je jednotka, která se stará o všechny interakce se souborovým systémem skrze `StorageContext`, obsahuje model objektů uložený ve formě XML a provádí kaskádové operace pro ukládání a načítání objektů. Aktuálně `ClassManager` podporuje perzistenci následujících datových typů:

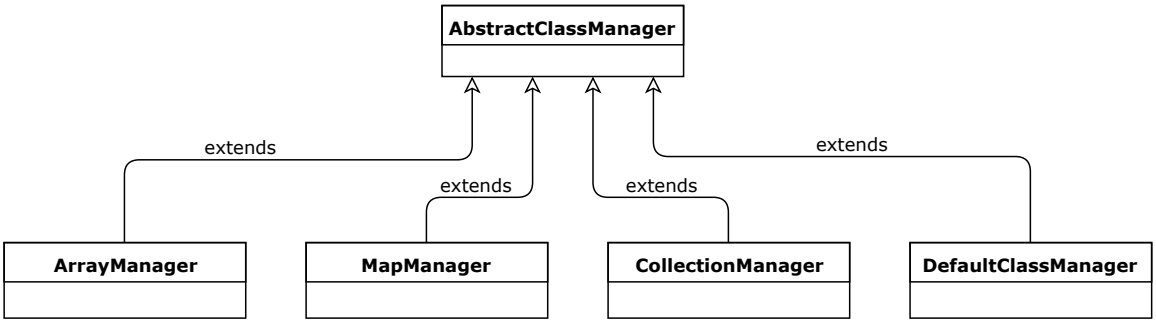
- primitivní datové typy a jejich wrappery[4]
- pole

[4] Např. pro integer `int` třída `Integer`.

- kolekce `java.lang.Collection`
- mapy `java.lang.Map`
- třídy definované programátorem

Pole, kolekce a mapy mají však zvláštní podmínky pro perzistenci. Jak již bylo zmíněno, systém nepodporuje přímou perzistenci některé z těchto tříd, neboť je nemožné tyto objekty jednoznačně identifikovat v rámci perzistenčního modelu a následně nelze identifikovat pro načtení zpět do paměti. Avšak v případě, že instance těchto tříd jsou součástí objektu s definovaným *objectId*, je možné je perzistovat, neboť tím pádem jsou jasně identifikovány daným objektem a budou v XML modelu reprezentovány unikátní referencí. Proto jakékoli kolekce, mapy a pole mají vždy své soubory obsahující všechny takovéto objekty obsažené v perzistovaných objektech.

V ukázce 4.2 je znázorněn diagram tříd pro všechny aktuálně možné `ClassManagery`. Kolekce, mapy a pole mají v systému dedikované třídy, které se od sebe liší implementacemi ukládání a načítání objektů. Všechny společné funkce jako je přístup k jednotlivým atributům pomocí reflexe nebo tvorba XML modelu obsahuje abstraktní třída `AbstractClassManager`.



Obrázek 4.2: Diagram tříd pro třídy `ClassManager`

Třída `ClassManager` obsahuje XML model perzistovaných objektů, který je možno do dotazovat pomocí jazyka XPath. Tento model je možno vytvořit buď perzistováním určitého objektu, nebo jej systém umí automaticky načíst z souboru v systémové složce, který byl vytvořen při jedné z předchozích perzistencí. Dále pak nabízí cache pro objekty, které jsou již v perzistenčním systému přítomny, což urychluje jejich hledání při události `load`.

Aktuální verze třídy `ClassManager` podporuje perzistenci také pro statické atributy. Tyto atributy jsou nyní aktualizovány při perzistenci jakéhokoli objektu třídy obsahující statické atributy. Dále byla implementována podpora pro perzistenci všech modifikátorů viditelnosti, tedy pro `private`, `protected` i `public` a také atributů `final`. U atributů typu `final static` je však třeba být velmi opatrný. Jejich inicializace je totiž spouštěna automaticky při načtení třídy. V případě, že tento atribut je např. `String`, kompilátor automaticky nastaví hodnotu tohoto atributu jako literál a znemožni tím jakoukoli jeho editaci, neboť prostor alokovaný pro tento literál je konstantní a neměnný.

4.9 Použité technologie

Celý systém je založen zejména na užití Java Reflection API, které poskytuje robustní přístup pro introspekci objektů. Skrze něj systém přistupuje k objektům a jejich obsahu, získává hodnoty a následně je zpracovává a zanáší do interního XML modelu.

Pro práci se soubory XML jsme použili Java API for XML Processing, především jeho část zabývající se DOM. Ta umožňuje vytvářet XML dokumenty a ukládat je ve formě XML souborů. Také obsahuje podporu pro jazyk XPath, tedy jazyka, který nabízí dotazování jednotlivých XML elementů. Ačkoli v aktuální verzi systému XPath využíváme jen okrajově, jeho podpora přidává systému např. možnost vyhledávání objektů na základě hodnot jejich atributů jako je tomu u Hibernate nebo EclipseLink.

4.10 Testování

Testování vytvořeného systému probíhalo zejména ke konci vývoje, a to formou akceptačních testů. V rámci něj jsme vytvořili nový testovací projekt vytvářející objekty Petriho sítí. Celý jejich systém jsme perzistovali a následně nahráli zpět do paměti. Ověřovali jsme ekvivalenci objektů na základě implementované metody `equals()`.

Toto testování odhalilo některé nedostatky v systému, zejména pak nemožnost perzistovat statické atributy. Tuto možnost jsme tedy na základě těchto testů doplnili. Samotný výstup ve formě XML souborů nemusel být testován, neboť s každým zpětným nahráváním souborů systém musí XML data parsovat a tím také ověřuje jejich validitu. Část frameworku byla také otestována jednotkovými testy, které byly vytvořeny za pomocí frameworku `jUnit 4.0`. Tyto testy naleznete ve složce `src/test`. Celkové pokrytí systému testy se pohybuje přibližně kolem 80%.

Kapitola 5

Závěr

Tato práce se věnovala průzkumu existujících řešení pro perzistenci v jazyce Java a shrnula jejich výhody, nevýhody a rozdíly. Na základě těchto faktů byl navržen a implementován perzistenční systém pro malé a střední aplikace bez použití dodatečných nástrojů. Součástí práce bylo také vytvoření testovací sady dat, která prověřuje očekávanou funkčnost.

Motivace pro vytvoření systému výše popsaných vlastností vyvstává z již zmiňované simulace Petriho sítí, kde je pozornost programátora kladena zejména na vytváření simulace, a perzistenční systém by tedy měl být transparentní a neměl by příliš ovlivňovat samotné programování simulací. Proto implementovaný systém na rozdíl od frameworků Hibernate, EclipseLink či DataNucleus nevyžaduje žádné mapování ani anotaci perzistovaných tříd. Jediné dvě podmínky pro možnost perzistence třídy je přítomnost *objectId* anotovaného `@ObjectId` a veřejného bezparametrického konstruktoru. Jakékoli ostatní anotace jako třeba `@Entity` v JPA nejsou potřeba. Nevýhodou oproti těmto robustním dlouho vyvíjeným frameworkům je především nepřítomnost transakcí. Framework vyvinutý v rámci této práce se nesnaží být konkurentem těmto profesionálním frameworkům, nýbrž nabízí možnost perzistence i malým aplikacím. S užitím našeho systému v enterprise prostředí se nepočítá.

Cíle práce se podařilo realizovat a vznikl systém pro JVM podporující perzistenci objektů do XML souborů implementovaný v jazyce Java, konkrétně pro jeho verzi 8. V rámci testovacích dat byla také potvrzena funkčnost pro původní záměr, tedy perzistenci Petriho sítí reprezentovaných objekty. Systém si však stejným způsobem poradí s jakýmikoli jinými podporovanými objekty objekty, což z něj dělá univerzální znovupoužitelné perzistentní prostředí.

Do budoucna je možno do systému přidat další podporované typy objektů pro perzistenci, prostředí pro transakce nebo cache pro rychlejší načítání objektů. Na vytvořeném systému bych rád dále pracoval a zdokonaloval jej, neboť perzistence je komplexní problematika dotýkající se spousty přidružených problémů v Javě a skrze vývoj perzistentního prostředí je možné lépe pochopit principy JVM a objektového modelu v Javě.

Literatura

[1] Hibernate ORM, Documentation - 5.2.
URL http://hibernate.org/orm/documentation/5.2/

[2] Java Persistence API. 5 2006, jSR 220: Enterprise JavaBeans^{TM},Version 3.0.
URL http://download.oracle.com/otn-pub/jcp/ejb-3_0-fr-eval-oth-JSpec/ejb-3_0-fr-spec-persistence.pdf

[3] Alur, D.; Crupi, J.; Malks, D.: *Core J2EE patterns*. Upper Saddle River, NJ: Prentice-Hall PTR, druhé vydání, 2003, ISBN 01-314-2246-4.

[4] Atkinson, M. P.; Bailey, P. J.; Chisholm, K. J.; aj.: An Approach to Persistent Programming. *The Computer Journal*, ročník 26, č. 4, 1983-11-01: s. 360–365, ISSN 0010-4620, doi:10.1093/comjnl/26.4.360.
URL https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/26.4.360

[5] Atkinson, M. P.; Daynès, L.; Jordan, M. J.; aj.: An orthogonally persistent Java. *ACM SIGMOD Record*, ročník 25, č. 4, 1996-12-01: s. 68–75, ISSN 01635808, doi:10.1145/245882.245905.
URL http://portal.acm.org/citation.cfm?doid=245882.245905

[6] Bauer, C.; King, G.; Gregory, G.; aj.: *Java persistence with Hibernate*. Shelter Island, NY: Manning, druhé vydání, 2016, ISBN 9781617290459.

[7] Codd, E. F.: *The relational model for database management*. Reading, Mass.: Addison-Wesley, druhé vydání, c1990, ISBN 02-011-4192-2.

[8] Fisher, M.; Ellis, J.; Bruce, J.: *JDBC API tutorial and reference*. Boston, MA: Addison-Wesley, třetí vydání, 2004, ISBN 03-211-7384-8.

[9] Fowler, M.: Plain Old Java Object. 2000.
URL https://www.martinfowler.com/bliki/POJO.html

[10] Lindholm, T.: *The Java virtual machine specification*. Upper Saddle River, NJ: Addison-Wesley, java se 8 edition. vydání, 2014, ISBN 978-0133905908.

[11] Marquez, A.; Blackburn, S.; Mercer, G.; aj.: Implementing Orthogonally Persistent Java. In *Revised Papers from the 9th International Workshop on Persistent Object Systems*, POS-9, London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-42735-X, s. 247–261.
URL http://dl.acm.org/citation.cfm?id=648124.747395

[12] Marr, S.: The Java Data Objects Persistence Model. 2006.
URL http://stefan-marr.de/pages/jdo-persistence-model/

[13] Monson-Haefel, R.: *Enterprise JavaBeans*. Sebastopol, CA: O'Reilly, Čtvrté vydání, c2004, ISBN 978-0596005306.

[14] Richardson, C.: *POJOs in action*. Greenwich, Conn.: Manning, první vydání, c2006, ISBN 19-323-9458-3.

[15] Roos, R. M.: *Java data objects*. London ; Boston: Addison Wesley, první vydání, 2003, ISBN 03-211-2380-8.

[16] Schildt, H.: *Java*. New York: McGraw-Hill Education, 9 vydání, c2014, ISBN 9780071808552.

[17] Silberschatz, A.; Korth, H. F.; Sudarshan, S.: *Database system concepts*. New York: McGraw-Hill, 6 vydání, c2011, ISBN 978-0-07-352332-3.

[18] Sun Microsystems: *JavaBeans*^{TM}. 1997.
URL http://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf

[19] Takasaka, S.: *Survey of Persistence Approaches*. Master thesis, Institute of Technology/Stockholm University, Stockholm, 2005.

Příloha A

Manuál

Knihovnu pro perzistenční systém je možno získat dvěma způsoby:

1. dodaného souboru JAR na DVD ve složce `/nodbpersistence/dest/`
2. spuštěním překladu a sestavení na hostitelském stroji

V případě překladu knihovny na hostitelském stroji je možno složku s projektem nalézt buďto na DVD ve složce `/nodbpersistence/`, nebo aktuální verzi ze vzdáleného git repozitáře.

```
git clone https://github.com/bayerhonza/nodbpersistence.git
```

V obou dvou případech je následně možno sestavit knihovnu příkazem `ant package` s využitím nástroje Ant nebo za použití frameworku Maven, a to příkazem `mvn install`. Obě dvě možnost vytvoří ve složce projektu složku `dest`, kde se po přeložení a sestavení bude nacházet soubor JAR perzistenčního systému, který je následně možno přidat do `classpath` při překladu aplikace využívající perzistenci.

Při tvorbě tříd určených k perzistenci je třeba myslet na dvě nutné podmínky pro možnost perzistence vyvinutým systémem, a to přítomnost *objectId* anotovaného `@ObjectId` a veřejného bezparametrického konstruktoru:

```
public class Test {
    @ObjectId
    public Long objectId;
    public String string;

    public Test() {}

    public Test(String string) {this.string = string;}
}
```

Následně je třeba vytvořit perzistenční systém: Pro vytvoření perzistenčního systému je nutné inicializovat nastavení, k čemuž slouží třída `PersistenceSettings`. Samotné hodnoty nastavení se potom přidávají zřetězováním metod typické pro návrhový vzor *builder*:

```
PersistenceSettings settings = new PersistenceSettings();
    .setRootPath("/root/path");
```

Dalším krokem je vytvoření *factory* pro třídu `PersistenceManager`, k čemuž slouží třída `PersistenceFactoryBuilder` zpracovávající nastavení `settings`:

```
PersistenceManagerFactoryBuilder builder =
    new PersistenceManagerFactoryBuilder(settings);
PersistenceManagerFactory pmFactory =
    builder.buildPersistenceManagerFactory();
```

Pro zpřístupnění jádra persistence je nutné vytvořit `PersistenceManager`, který dává k dispozici `PersistenceManagerFactory`:

```
PersistenceManager pm = pmFactory.getPersistenceManager();
```

Nyní je již možno využívat operaci `persist`, následně propagovat změny použitím `flush()` a načtení objektu zpět pomocí operace `load`:

```
Test o = new Test("Hello world!");
Long objectId = pm.persist(o);
pm.flush();
Test o_loaded = pm.load(objectId,Test.class);
```

Příloha B

Obsah CD

```
/
├── nodbpersistence/ ..... složka projektu s perzistenčním systémem
│   ├── dest/ ..... složka s souborem JAR
│   │   └── nodbpersistence-1.0.jar
│   ├── resources/ ..... složka pro konfigurační soubory
│   │   ├── collections.dtd ..... DTD pro XML soubory s kolekcemi
│   │   ├── maps.dtd ..... DTD pro XML soubory s mapami
│   │   ├── nodbpersistence.xml ..... konfigurační soubor pro perzistenční systém
│   │   └── objects.dtd ..... DTD pro XML soubory s objekty
│   ├── src/ ..... složka se zdrojovými kódy
│   │   ├── main/ ..... složka s třídami systému
│   │   └── test/ ..... složka s testovacími třídami
│   │       └── java/
│   │           └── cz/vutbr/fit/nodbpersistence/
│   │               └── core/
│   │                   └── ComplexTest{1-8}.java ... integrační testy
│   ├── build.xml ..... soubor s nastavením buildu pro Ant
│   ├── LICENSE ..... licence práce
│   ├── pom.xml ..... projektový soubor pro Maven
│   └── README.md. ..... README soubor
├── test_project/ ..... složka projektu pro testování Petriho sítí
│   ├── dest/ ..... složka se souborem JAR
│   ├── lib/ ..... složka s knihovnou nodbpersistence-1.0.jar
│   ├── resources/ ..... složka pro konfigurační soubory
│   ├── src/ ..... složka se zdrojovými kódy testovacího projektu
│   └── build.xml ..... soubor nastavení buildu pro Ant
├── xbayer08/ ..... tato práce v LaTeX
├── class_diagram.png ..... Diagram tříd pro vyvinutý systém
└── xbayer08.pdf ..... tato práce ve formátu PDF
```