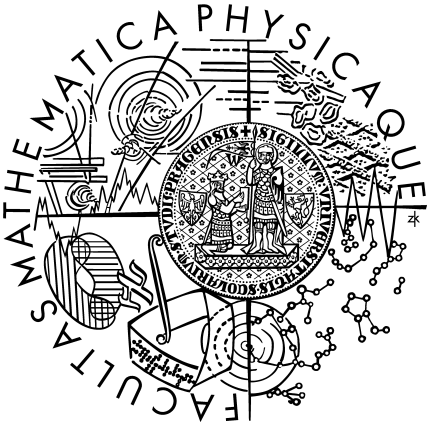
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# DIPLOMOVÁ PRÁCE




Michal Danihelka

## Úložiště pro rozvolněné objekty

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Žemlička, Ph.D.
Studijní program: Informatika

Rád bych tímto poděkoval panu RNDr. Michalu Žemličkovi, Ph.D. za pomoc při psaní této diplomové práce, zejména za několikahodinové diskuze, které se občas protáhly až do pozdních večerních hodin.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7.srpna 2009 Michal Danihelka

# Obsah

# Seznam obrázků

## Abstrakt

Název práce: *Úložiště pro rozvolněné objekty*

Autor: *Michal Danihelka*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Michal Žemlička, Ph.D.*

e-mail vedoucího: *Michal.Zemlicka@mff.cuni.cz*

Abstrakt:

V objektovém modelu je standardně každému objektu přiřazena třída, ve které jsou nejen definována data objektu, ale i metody, které s těmito daty pracují. Tento přístup není vhodný v dynamicky se měnícím heterogenním prostředí. Při práci s reálnými daty nebývají data získaná z různých zdrojů rozsahově homogenní. O některých objektech je známo více informací než o jiných objektech. Množství dostupných informací se může navíc měnit v čase. I přes tato omezení musí být na daných datech poskytnuta maximální možná funkcionalita.

V této práci bude navržen objektový model v Javě, který umí s takto různorodými daty pracovat. Objekty nejsou do tříd přiřazeny explicitně, ale to, které metody je možné zavolat na konkrétním objektu, je určeno dostupností a hodnotami atributů daného objektu. Důraz bude kladen na jednoduchost použití a vysokou odolnost proti chybám vzniklých nejen při vývoji, ale i při běhu aplikací. Součástí této práce bude také návrh aplikace k ověření použitelnosti modelu rozvolněných objektů. Nejdůležitější části modelu i aplikace budou implementovány.

Klíčová slova: Objektový model pro persistentní data, parciálně dostupná data, heterogenní data

## Abstract

Title: *Storage for relaxed objects*

Author: *Michal Danihelka*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Michal Žemlička, Ph.D.*

Supervisor's e-mail address: *Michal.Zemlicka@mff.cuni.cz*

Abstract:

In object model is standardly to each object assigned one class which defines not only the object data but also the methods working with these data. This approach is not practical in dynamic heterogeneous environment. There is a significant problem with real data homogenity. The known data about some objects are different than about the others. The number of accessible information can change in time too. Despite these facts the maximal possible functionality have to be offered.

In this thesis there will be designed an object model in Java, which can work with such a various data. The objects are not explicitly assigned to classes, but the selection of methods which can be called on the concrete object depends on accessibility and values of object's attributes. The emphasis will be put on easy of use and high resistance againts errors caused during development and execution of applications. To prove the usability of relaxed objects model the design of an application will be part of this work. The most important parts of model and application will be implemented.

Keywords: Persistent data object model, partially available data, heterogeneous data

# Kapitola 1

# Úvod

Objektově orientované programování (dále jen OOP) je způsob vývoje software, který patří v současné době k nejpoužívanějším. OOP je dostatečně obecné a použitelné pro většinu dnes vytvářených aplikací. Základní myšlenkou OOP je sestavování aplikací z objektů, které reprezentují prvky modelované reality. Objekty mají definovaný nejen svůj datový obsah, ale také své chování. Skupina objektů, které mají stejný datový obsah i chování, patří do stejné třídy. Objekty jsou od doby svého vzniku pevně svázány s třídami a říkáme, že jsou jejich instancemi. Chování objektu je dáno požadavky, které se mohou danému objektu posílat formou volání jeho metod. Jaké metody je možné na objektu zavolat je dáno třídou objektu. Datová složka tříd je reprezentována sadou vlastností objektů zvaných atributy. Objekty jakožto instance těchto tříd definují konkrétní hodnoty jejich atributů. Například třída objektů "Zaměstnanec" obsahuje atributy "Jméno" a "Mzda". Konkrétní zaměstnanci, tedy instance této třídy, mají těmto atributům přiřazeny hodnoty - například "Jan Čáp" má mzdu "25 000".

Abychom mohli s objekty v tomto pojetí pracovat, musíme nejdříve definovat třídy, které specifikují atributy a chování objektů. Teprve poté můžeme vytvářet či používat objekty jakožto instance těchto tříd. V některých případech by však bylo vhodné tuto explicitní vazbu mezi třídou a objektem zrušit. Třídu můžeme přiřazovat objektu implicitně podle dostupnosti a hodnot atributů, ze kterých se konkrétní objekt aktuálně skládá. Nazvěme objekty, které nemají pevnou vazbu na příslušnou třídu, rozvolněnými objekty. Tyto objekty nemají pevně definovanou sadu atributů a je tedy možné kdykoliv objektu nový atribut dodefinovat. Ani metody, které je možné na objektu zavolat, nejsou pevně dané, ale jsou určeny příslušností objektu k dané implicitní třídě. Tato implicitní třída se ale může kdykoliv v průběhu životního cyklu daného objektu změnit - například nastavením hodnoty zatím nenastaveného atributu objektu nebo změnou hodnoty některého z atributů objektu.

Rozvolněné objekty nám umožňují vyřešit některé problémy při převádění entit reálného světa do světa objektů. Například informace získané z různých zdrojů nemusí být rozsahově homogenní. Pak pro objekty reprezentující tyto informace mohou existovat různé množiny definovaných atributů. V původním pojetí, pokud by bylo známých informací méně než požaduje třída, do které objekt patří, nebudou mít některé atributy nastavené hodnoty a to poté může způsobovat problémy při volání metod, které s těmito atributy pracují. Pokud je známých hodnot ale více, budou hodnoty atributů, které třída neobsahuje, ztraceny. Tomu lze zabránit například definováním dalších tříd, které všechny nové informace pojmou. Dodatečné informace mohou vznikat i během životního cyklu objektů. V původním pojetí je obtížné tyto nové hodnoty atributů k existujícím objektům připojit. U rozvolněných objektů s tím však nemáme žádné potíže, neboť rozvolněné objekty nejsou vázány na pevně definovanou sadu atributů v odpovídající třídě.

Při každém volání metody hrozí nebezpečí, že se nepodaří danou metodu úspěšně dokončit, protože předané parametry metody nebo objekt, na kterém je metoda volána, nesplňují určité podmínky. Kdybychom tyto podmínky měli evidované u jednotlivých metod, pak není problém zjistit, jaké metody je možné zavolat na konkrétním objektu podle jeho aktuálního stavu. Nepotřebujeme tedy mít metody definované příslušenstvím do třídy, ale můžeme volat na objektu všechny metody, které lze úspěšně dokončit. Pokud se tyto podmínky ověřují za běhu aplikace, tak se často může předejít předběžnému ukončení aplikace nebo vyhození vyjímky například tím, že dané volání metody uživateli vůbec nezpřístupníme. Navíc bychom mohli metodu zavolat na všech objektech, které v daný okamžik splňují podmínky metody. Další výhodou by bylo, že při vývoji aplikace v okamžiku, kdy potřebujeme nějakou metodu volat, si přímo u ní můžeme zjistit všechny její požadavky na vstupní data. To bývá většinou velmi složité, neboť požadavky metod bývají uvedeny na různých místech v různých třídách. V některých případech lze požadavky metod zjistit jen z příslušné části zdrojového kódu metody nebo dokonce jen z dokumentace.

Prvním cílem této diplomové práce bude vůbec první realizace konceptů rozvolněných objektů, která bude provedena v programovacím jazyce Java s použitím databázového systému Oracle. Objektový model je realizace konceptů objektově orientovaného programování v konkrétním prostředí. V této diplomové práci bude navržen nový objektový model, který dokáže pracovat s rozvolněnými objekty. Tato diplomová práce přímo navazuje na článek o rozvolněných objektech [1], ze kterého přebírá základní koncepty a dále je rozšiřuje. V novém objektovém modelu nebudou objekty přiřazeny do tříd explicitně, ale rozhraní objektu definující všechny metody, které lze na objektu zavolat, bude dáno dostupností a hodnotami atributů daného objektu. Navržený objektový model tak bude poskytovat na objektech vždy maximální možnou funkcionalitu.

Druhým cílem této diplomové práce bude ověření použitelnosti modelu rozvolněných objektů při tvorbě reálných aplikací. Bude tedy navržena ukázková aplikace Úložiště výsledků testů. Na této aplikaci budou ukázány nejdůležitější vlastnosti modelu rozvolněných objektů. Úložiště výsledků testů bude navrženo standardním způsobem a následně i pomocí modelu rozvolněných objektů. Nakonec budou implementovány nejdůležitější části obou verzí aplikace a budou mezi sebou porovnány.

# Kapitola 2

# Základní koncepty rozvolněných objektů

V této kapitole budou popsány všechny důležité vlastnosti rozvolněných objektů. Tento popis bude proveden nezávisle na programovacím jazyku i databázovém systému. Konkrétní realizace těchto konceptů v programovacím jazyku Java s použitím databázového systému Oracle bude popsána v kapitole 3.

## 2.1 Příklady aplikací

Nejprve bude popsáno několik ukázkových aplikací, které by mohly plně využít vlastností rozvolněných objektů.

### 2.1.1 Evidence přestupků řidičů

Ještě donedávna se za přestupky řidičů udělovaly pouze pokuty nebo v případě velmi závažných přestupků byl rovnou odebrán řidičský průkaz. Jelikož však tento systém nedokázal sledovat opakované páchání přestupků řidičů, tak byl v České republice zaveden 1. července 2006 systém bodového hodnocení. Přestupek řidiče je policistou nahlášen na úřad, kde je řidič evidován, a tento úřad zvolí, kolik přidělí řidiči bodů. Současný systém bodového hodnocení v České republice není zatím tak promyšlený jako v jiných státech, kde se podobný systém používá. Například v Německu jsou jednotlivé bodové postihy mnohem lépe navrženy a lépe vystihují jednotlivá provinění. Zároveň má řidič v Německu více možností, jak si získané trestné body nějakým způsobem odečíst a je tedy více tolerována chyba řidiče za předpokladu, že si bude za chybu nést případné následky. Po dosažení určitého počtu bodů je řidiči v Německu automaticky sděleno varování, zatímco v České republice je řidič informován jen o odebrání řidičského průkazu a jedinou možností řidiče, který chce zjistit svůj aktuální stav bodů, je podání speciální žádosti. Sice od zavedení bodového systému už bylo provedeno několik úprav včetně možnosti zjistit aktuání počet bodů na některých úřadech a poštách, ale dá se předpokládat, že současný bodový systém v České republice bude časem procházet řadou změn.

Protože při použití rozvolněných objektů by přímo u metod byly uvedeny všechny podmínky potřebné pro jejich provedení, tak by bylo mnohem snadnější jednotlivé změny v aplikaci provádět. Rozvolněné objekty navíc velmi jednoduše zpřístupňují všechny potřebné funkce pro všechny zainteresované osoby. Řidiči tak budou mít k dispozici funkci na zjištění aktuálního počtu přidělených bodů a možnost požádat o některé ze školení určené k jejich odečtení. Pro řidiče, kteří už mají odebraný řidičský průkaz, přibývá funkce žádosti o vydání řidičského průkazu nebo zjištění termínu, od kdy by bylo možné o vydání řidičského průkazu

zažádat. Úředníci budou mít zpřístupněny funkce zápisu trestných bodů řidičů a funkce pro vytisknutí dopisů pro všechny řidiče s aktuálně přesaženým limitem bodů pro odebrání řidičského průkazu. Jelikož i úředník může být řidičem, tak pro takového člověka jsou automaticky zpřístupněny všechny potřebné funkce pro řidiče i pro úředníka. Samozřejmě je nutné brát v potaz i přístupová práva, jestli daná osoba je vůbec oprávněna provádět danou operaci.

Každý úřad si vede svou evidenci přestupků řidičů, přičemž každý z těchto úřadů může mít zaznamenány o řidičích jiné informace, které jsou uložené do různých atributů objektů. Kdyby se ale vytvářela centrální evidence, stejně jako je to například v Německu, tak s použitím rozvolněných objektů není problém uložit do centrálního systému všechna doposud nasbíraná data. Jediný problém k vyřešení by byla synchronizace atributů mezi jednotlivými úřady, protože je možné, že jednotlivé úřady používají atributy se stejným názvem, ale zároveň různým významem.

Na této aplikaci budou dále v textu ukazovány vlastnosti rozvolněných objektů. Díky své jednoduchosti a podobnosti s velkým počtem jiných aplikací bude použití této aplikace pro tento účel velmi výhodné.

### 2.1.2 Úložiště výsledků testů

Nezbytnou součástí vývoje dnešních aplikací je proces testování, který je díky složitosti současně vyvíjených aplikací velmi časově náročný a proto je snaha minimalizovat manuální testování a naopak co nejvíce práce si ulehčit pomocí automatizovaných testů. Existuje velké množství různých kategorií testů, jako jsou například funkční testy, zátěžové testy nebo bezpečnostní testy. Bohužel každý druh testů má většinou svou vlastní terminologii a speciální reporty s výsledky testů. Jelikož ale vyhodnocování výsledků testů je ve všech případech podobné, bylo by možné navrhnout sdílené úložiště výsledků všech druhů testů. Jelikož by pak ke všem výsledkům všech druhů testů šlo přistupovat naprosto stejným způsobem, byla by výrazně usnadněna tvorba a úprava reportů o výsledcích testů. Vlastní prohlížení reportů o výsledcích testů by bylo také velmi zpřehledněno, jelikož po naučení čtení jednoho typu reportu o výsledcích testů by se člověk automaticky naučil číst všechny druhy reportů. Reporty s výsledky testů z různých kategorií samozřejmě obsahují různé informace, ale často je žádoucí, aby i reporty více spuštění testů ze stejné kategorie obsahovaly odlišné údaje v závislosti na tom, co se přesně v konkrétním spuštění testů měří.

Rozvolněné objekty by zde byly optimálně využity, protože by jednoduše dokázaly pracovat se všemi výsledky různých druhů testů nezávisle na tom, co všechno výsledky testů obsahují za informace. Navíc by bylo možné napsat takové obecné generování reportů, že by metody reprezentovaly jednotlivé části reportu. Ve výsledném reportu by se však tyto části objevily jen tehdy, když by jednotlivé spuštění testů, ze kterých by se výsledný report generoval, obsahovaly všechny potřebné údaje. Potřebná data by byla definována jako podmínky přímo u metod, tedy velmi přehledným způsobem. Takže po jediném napsání konkrétní části reportu by se tato část mohla objevit ve všech reportech všech druhů testů s podmínkou, že jednotlivé spuštění testů by obsahovaly požadované údaje.

Jak by mohlo vypadat takové sdílené úložiště s výsledky testů ze všech kategorií je popsáno včetně porovnání jeho realizace standardním způsobem s realizací pomocí rozvolněných objektů v kapitole 4.

### 2.1.3 Univerzitní informační systém

Podobná aplikace, jako je evidence přestupků řidičů, je univerzitní informační systém. Jednotlivé fakulty si evidují o zaměstnancích a studentech různé informace. Po vytvoření společného informačního systému by bylo možné pro zaměstnance a studenty, kteří své aktivity provádějí v rámci více fakult, poskytnout všechny potřebné funkcionality v rámci jediného informačního systému. Funkce by byly přístupné jen za předpokladu, že uživatel má nastavená požadovaná práva a jsou splněny všechny ostatní podmínky na data. Studenti by tak měli přístup k funkcím na přihlašování na předměty a zkoušky, pro zkoušející by byla přístupná funkce na zápis výsledku zkoušky studenta a účetní by mohly přistupovat k datům ze všech fakult, kde zaměstnanec pracuje, aby tak mohly podat daňové přiznání. Všichni uživatelé by také mohli mít možnost si zobrazit úkoly od svých nadřízených nebo učitelů a mohli by nastavovat jejich aktuální stav, takže by existoval větší přehled o aktuáním stavu všech plněných úkolů, který by byl také navíc velmi pohodlně přístupný v rámci jediného informačního systému.

### 2.1.4 Dějepisná mapa

Posledním příkladem aplikace, pro kterou by bylo velmi výhodné použít rozvolněné objekty, je dějepisná mapa. Tato mapa umožňuje zobrazovat mapu vybraného území v libovolném historickém okamžiku, a proto je potřeba pro všechny nasbírané informace o objektech ukládat navíc i časy od kdy a do kdy jsou dané informace platné. Jelikož získání historických dat je velmi obtížné, tak se dá předpokládat, že podobné objekty budou mít dostupné různé množiny atributů. Navíc tyto množiny definovaných atributů jsou rozdílné pro různé časové období, přesto je možné při použití rozvolněných objektů přistupovat ke všem funkcím, které jsou na daných objektech proveditelné.

## 2.2 Související práce

Tato práce přímo navazuje na článek o rozvolněných objektech [1], ze kterého přebírá základní myšlenky a dále je rozšiřuje. Hlavním přínosem této práce je vůbec první realizace konceptů rozvolněných objektů a vytvoření složitější aplikace k ověření použitelnosti rozvolněných objektů.

Myšlenkou ukládat data v databázi po sloupcích se zabýval databázový systém C-Store [2]. Ověřil, že pro čtení dat je řádově výhodnější ukládat data po sloupcích než po řádcích. Problém je, že C-Store obsahuje dvě optimalizované části. Jedna je optimalizovaná pro čtení a druhá pro zápis. Část optimalizovaná pro čtení místo, aby vytvářela indexy, tak si udržuje kopie dat optimálně uložené přímo pro konkrétní dotazy, což ale znemožňuje efektivní zápis a změnu dat. Naopak část optimalizovaná pro zápis neumožňuje efektivní čtení. Snaha rozvolněných objektů je převzít výhody sloupcové databáze pro čtení, ale zároveň umožnit použitelný zápis a změnu dat v databázi. Navíc rozvolněné objekty jsou použitelné nad jakoukoliv relační databází a nevyžadují tedy použití speciálního databázového systému.

Stohový systém [3] má za cíl být zastřešujícím systémem pro provozní systémy. Ukládá hodnoty atributů jednotlivých objektů včetně časové platnosti do jediné tabulky. Toto řešení sice přidá časovou podporu datům ze systémů, kde žádná časová podpora nebyla, ale kvůli výkonnostním problémům se dá použít jen jako centrální úložište dat, ze kterého se potřebná data v případě požadavku propagují do provozních databází. Oproti tomu rozvolněné objekty mají za cíl mít v databázi přímo uložená data, které běžící aplikace budou používat.

Způsob vývoje software, který se jmenuje Design by contract, se snaží snížit složitost vytvářených aplikací. Tento způsob vývoje software je podrobně popsán v diplomové práci [4]. Design by contract používá myšlenku kontraktu mezi dvěma moduly, což znamená, že každý modul zaručuje, že pokud jsou splněny požadavky daného modulu na vstupní parametry, tak po spuštění dané funkcionality tohoto modulu jsou zaručeny výstupní podmínky tohoto modulu. Teoreticky je tato problematika prozkoumána detailně, ale v praxi se zatím moc často nepoužívá. Problémem je určitě také fakt, že definice požadavků modulů nejsou povinné, což je způsobeno nejen složitostí všechny potřebné požadavky modulu definovat, ale také tím, že může být časově velmi složité tyto požadavky kontrolovat při běhu aplikace. Proto se použíjají jen pro testovací účely a na produkčním prostředí je tato kontrola podmínek většinou vypnuta. Toto jednoduché rozdělení, kdy jsou podmínky ověřovány, je dáno také přístupem nástrojů, které Design by contract podporují. Tyto nástroje totiž umožňují pouze určit, zda se má nebo nemá kontrola podmínek provádět. Rozvolněné objekty budou definici vstupních podmínek metod vyžadovat, protože podle nich budou zjišťovat, zda lze danou metodu na daném objektu vůbec spustit. Kontrola podmínek, které nejsou pro běh aplikace nutné, půjdou na produkčním prostředí z výkonostních důvodů vypnout. V případě, že metody budou definovat své výstupní podmínky, je možné v některých případech při volání metod vstupní podmínky netestovat, jelikož mohou být součástí výstupních podmínek dříve volaných metod. Zároveň je přímo v kódu aplikaci možné kdykoliv tuto kontrolu podmínek vypnout, protože může být zřejmé, že vstupní podmínky pro dané zavolání metody jsou splněny.

Jelikož je stále ve většině případů k uložení persistentních objektů používána relační databáze, je potřeba vytvářet mapování mezi objekty z objektového prostředí a relacemi z relační databáze. Většina frameworků (například Hibernate[1] pro Javu), které toto mapování poskytují, nabízí více možností jakým způsobem mapování v aplikaci zapisovat i realizovat. Například je možné toto mapování definovat v textových souborech nebo přímo ve zdrojovém kódu u jednotlivých atributů tříd nebo metod, které s persistentními atributy pracují. Možnost výběru mapování do relací může výrazně zvýšit výkon, protože se mapování může přizpůsobit požadavkům aplikace. Jelikož rozvolněné objekty nepoužívají atributy v konkrétních třídách, ale mají je společné pro všechny objekty, je výhodnější, než se snažit použít existující mapování, vytvořit vlastní realizaci mapování přímo optimalizovanou pro rozvolněné objekty. Výhodou je například možnost přidávat atributy za běhu aplikace. Oproti ostatním frameworkům budou rozvolněné objekty nabízet pouze jediné mapování. Největší riziko jediného mapování je výsledná rychlost přístupu k požadovaným datům, a proto bude výkon tohoto mapováni detailně otestován. Toto speciální mapování rozvolněných objektů do relační databáze je popsáno společně s jeho výhodami a nevýhodami v části 2.8.

Spousta požadavků na rozvolněné objekty je velmi podobných jako na ostatní frameworky. Například je potřeba vyhledávat persistentní objekty podle určitých kritérií. Jelikož jsou vývojáři zvyklí přemýšlet spíš ve světě objektů než ve světě relací, je velmi užitečné jim nabídnou dotazovací jazyk, který se co nejvíce podobá objektovým konvencím. Jelikož rozvolněné objekty mají speciální vlastnosti, tak bude výhodné vytvořit vlastní dotazovací jazyk, který bude porovnán s jedním z nejrozšířenějších objektově orientovaných dotazovacích jazyků HQL[2]. Dalším častým požadavkem je podpora pro práva, která by u rozvolněných objektů byla inspirováná systémem práv v IS MU[3].

[1] http://www.hibernate.org

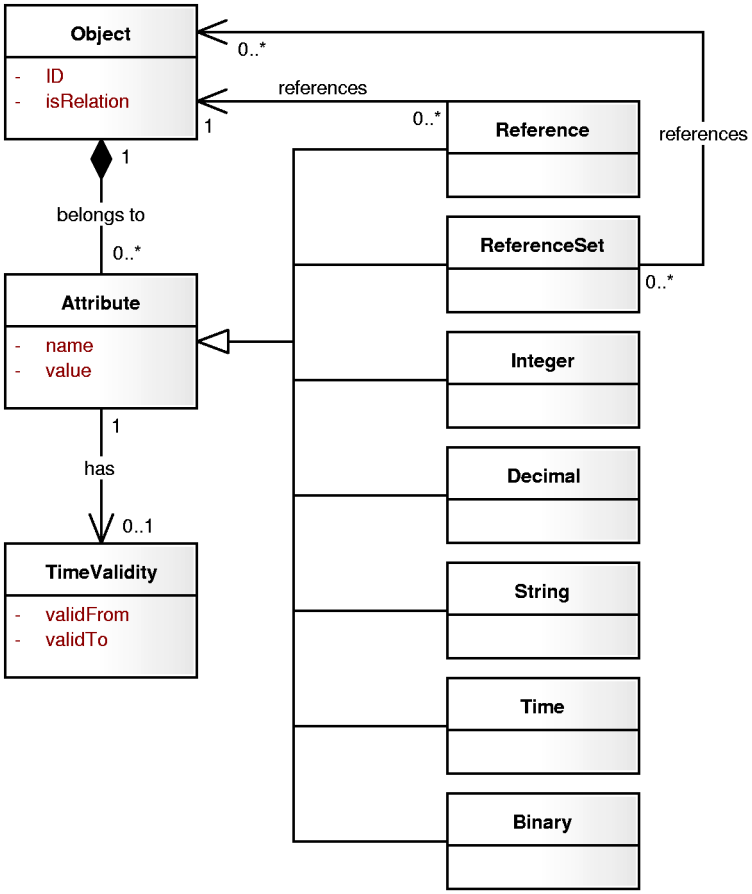
[2] http://docs.jboss.org/hibernate/stable/core/reference/en/html/queryhql.html

[3] http://is.muni.cz/clanky/1999_rufis-pazdziora.pl

## 2.3 Objekty a metody

V této části bude popsán úhel pohledu na objekty, vztahy mezi objekty a metody objektů v modelu rozvolněných objektů. Pro porovnání bude uveden i pohled standardního objektového modelu.

### 2.3.1 Objekty




Obrázek 2.1: Objekt s atributy

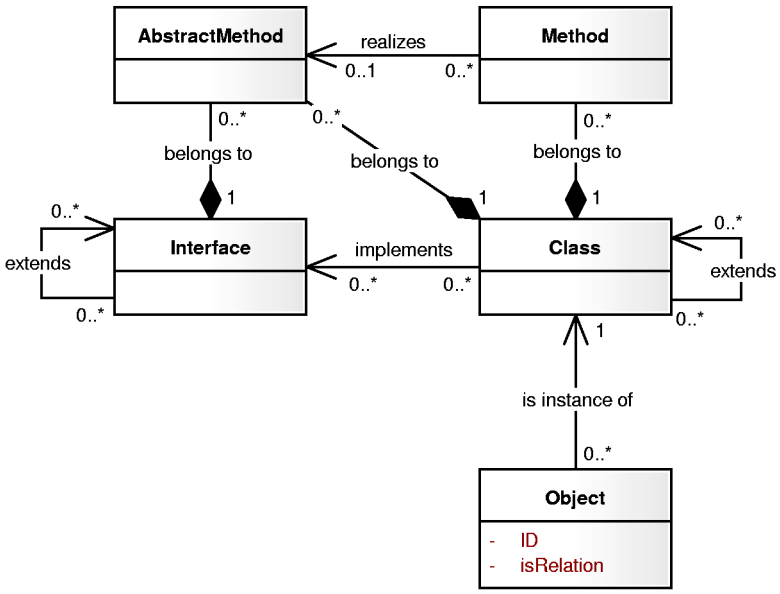
Základním prvkem je objekt (Object), který reprezentuje nějaký prvek modelované reality. Objekt obsahuje několik atributů (Attribute), které reprezentují jeho datový obsah. Každý atribut má své jméno a svůj typ, který určuje jakých hodnot může atribut nabývat. Aby k jednotlivým hodnotám atributů šlo jednoduše přistupovat, může objekt obsahovat jediný atribut s daným jménem, a tedy hodnota atributu je přístupná jen pomocí tohoto jména. Hodnotami atomických atributů mohou být postupně celé číslo, desetinné číslo, textový řetězec, časový okamžik a posloupnost bitů reprezentující například video (Integer, Decimal, String, Time a Binary). Hodnotou atributů typu odkaz (Reference) je další objekt. V případě atributu typu množina odkazů (ReferenceSet) je hodnotou atributu množina dalších objektů, která může být i prázdná. Hodnota každého atributu se může ukládat buď pouze aktuální nebo je možné ukládat hodnoty, které jsou platné jen v určitém časovém úseku (TimeValidity).

Jelikož je potřeba umět od sebe rozpoznávat jednotlivé objekty a navíc je potřeba se na objekty umět odkazovat, tak každý objekt obsahuje atribut ID určený k jeho jednoznačné identifikaci. Hodnota tohoto atributu je objektu automaticky přidělena při jeho vytvoření a už není nikdy změněna. Tady je důležité si uvědomit, že i když dva objekty obsahují stejné atributy, které mají všechny stejnou hodnotu, jedná se o rozdílné objekty.

Jediné rozdíly tohoto pohledu na objekty oproti standardnímu objektovému modelu je možnost ukládat hodnoty atributů s časovou platností a nemožnost, aby objekt obsahoval více atributů se stejným názvem (což je ale spíše nedostatek některých realizací vícenásobné dědičnosti, kde k této situaci může dojít v případě dědění z více tříd, které daný atribut obsahují).

### 2.3.2 Metody

Nejdříve si ukažme, jak vypadá vztah objektů a metod, které lze na objektech volat, ve standardním objektovém modelu.




Obrázek 2.2: Metody objektu ve standardním objektovém modelu

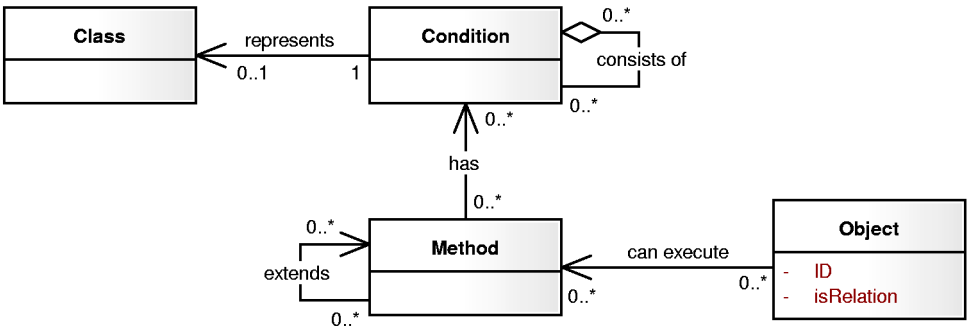
Na obrázku 2.2 můžeme vidět, že každý objekt má přiřazenou právě jednu třídu (vztah is instance of). Říkáme, že objekt je instancí této třídy. Jednou ze základních vlastností standardního objektového modelu je vlastnost, které se říká dědičnost. Můžeme pro třídu určit množinu jejích předků (vztah extends). V Javě musíme vždy určit právě jednoho předka, ale v C++ je možné zvolit i více předků. Tyto předky daná třída nějakým způsobem rozšiřuje. Potomek může přistupovat k datům a metodám jejich předků, pakliže to nemá od předků explicitně zakázáno.

Každá metoda (Method) je explicitně spojena se třídou (vztah belongs to). V levé polovině obrázku je zakreslen přístup abstrakce metod. V praxi se ukazuje, že je výhodné mít možnost uvádět jen hlavičky metod bez implementace. Kompilátorem je pak zaručeno, že tyto metody budou v implementaci třídy přístupné. U tříd to jsou abstraktní metody (Abstract method), které se odkazují na danou třídu (vztah belongs to). Nelze pak vytvářet instance tříd, které

nedefinují implementaci všech vyžadovaných abstraktních metod. Úplná abstrakce metod je realizována rozhraním (Interface). V tomto případě se abstraktní metoda odkazuje na příslušné rozhraní (vztah belongs to). Rozhraní neumožňuje definici jakýchkoliv atributů ani žádnou implementaci metod. Každé rozhraní může rozšiřovat libovolný počet jiných rozhraní (vztah extends). Rozhraní vlastně akorát říká, jaké implementace metod musí výsledná implementace třídy obsahovat.

Třída může implementovat libovolný počet rozhraní (vztah implements). Posledním vztahem zakresleným na obrázku je vztah mezi abstraktní metodou a metodou (vztah realizes). Každá implementace metody může buď překrývat metodu definovanou na předkovi nebo realizuje metodu rozhraní a nebo realizuje abstraktní metodu předka. Může nastat i více z těchto možností najednou. Samozřejmě poslední možností je, že metoda nemá žádný vztah s okolním prostředím, což je ale spíše přístup procedurálního způsobu programování.

Teď se pojďme pro porovnání podívat, jakým způsobem řeší vztah metod a objektů model rozvolněných objektů.




Obrázek 2.3: Metody objektu v modelu rozvolněných objektů

Na obrázku 2.3 je názorně vidět flexibilní přístup rozvolněných objektů ke spouštění metod na objektech. Na každém objektu je možné zavolat (vztah can execute) všechny metody, které mají ve své definici (vztah has) uvedeny podmínky, které objekt splňuje. Je důležité si uvědomit, že podmínky se týkají nejen požadavků na objekty, což jsou ty podmínky, které reprezentují jednotlivé třídy (vztah represents), ale také se požadavky týkají jednotlivých parametrů metod, které naopak se na třídy neodkazují. V definici každé podmínky se může použít libovolné množství jiných podmínek (vztah consists of).

Nejsložitějším vztahem rozvolněných objektů na tomto obrázku je vztah mezi jednotlivými metodami (vztah extends). Ve standardním objektovém modelu je možné překrývat metody, tedy potomek definuje metodu se stejným názvem i stejnými parametry. Jelikož ale rozvolněné objekty nemají třídu určenou explicitně a určuje se až na základě aktuálních hodnot atributů objektu, muselo by se explicitně určit pořadí, ve kterém se mají metody hledat.

Řešení navrhované ve článku [1] předpokládá, že každá metoda má jednoznačný název a dále navrhuje zavedení zástupných metod. Může pak existovat více metod se stejným názvem i stejnými parametry a zástupná metoda pak podle aktuálně předaných parametrů z nich vybere tu nejvhodnější, například tu, která požadovaný algoritmus provádí na daných datech nejrychleji. Problémem je, že je nutné tyto zástupné metody explicitně psát, což výrazně omezuje možnosti oproti standardnímu objektovému modelu, kde se metoda ze správné třídy zavolá sama. Zástupné metody mají samozřejmě tu výhodu, že mají řádově větší možnosti při výběru optimální metody. Mohou totiž metodu vybírat na základě aktuálně

předaných parametrů, kdežto ve standardním objektovém modelu se metoda vybírá podle příslušnosti objektu do třídy.

Zkusme se proto zamyslet, co by v modelu rozvolněných objektů znamenalo navrhnout obecný postup pro výběr té správné přetížené metody. Můžeme předpokládat, že podmínky na objekty mají tvar popsaný v části 2.4.2, která se věnuje popisu podmínek metod. Tedy podmínky jsou vytvářeny skládáním jiných podmínek pomocí logických operátorů and, or a not. Logická spojka and vlastně říká, že k původní podmínce přidáváme další požadavky, tedy vlastně z třídy definované původní podmínkou dědíme. Naopak logická spojka or říká, že původné podmínky na paramtery zmírňujeme (protože podmínka může být vyhodnocena jako pravdivá i když je splněna jen nově přidaná podmínka), tedy vlastně definujeme předka. Podmínka not nám definuje úplně novou třídu, která s předchozí podmínkou vůbec logicky nesouvisí. Můžeme si tedy na základě definice podmínek sami vytvořit hierarchii dědičnosti. Tento přístup nás sice nezbaví povinosti určovat v jakém pořadí se metody mají hledat, ale výrazně omezí logiku, která bude požadovanou metodu hledat.

V případě, že voláme metodu, která má na objekt definovanou podmínku A, pak můžeme použít následující postup: ze všech metod, které mají stejné parametry a stejné jméno, vybereme ty, které navíc v sobě obsahují podmínku A bez negace. Můžeme předpokládat (například proto, že takto se bude definovat implicitní hierarchie metod), že jsou to podmínky typu (A && B) nebo (A || B). Jelikož většinou je potřeba volat nejspecifičtější podmínku, tak podmínky typu (A || B) budeme ignorovat, protože ještě oslabují naše současné požadavky na objekt. Začneme tedy procházet podmínky typu (A && B), které představují všechny naše přímé potomky. První podmínku, která je splněná, začneme prohledávat úplně stejným způsobem jako podmínku A, budeme tedy hledat přímé potomky podmínky (A && B), které jsou typu (A && B && C). Opět při první splněné podmínce, začneme prohledávat všechny přímé potomky. V okamžiku, kdy jsme prošli všechny potomky podmínky a žádná z podmínek není splněná, jsme nalezli nejspecifičtější metodu, kterou tedy zavoláme. Samozřejmě za běhu aplikace jsou tyto seznamy k prohledání vytvořeny při spuštění, a je tedy spouštění metody zpomalováno akorát vyhodnocováním jednotlivých podmínek.

Pointa je v tom, že je potřeba vždy určit jen pořadí všech podmínek na stejné úrovni dědičnosti. Kdyby bylo explicitní určení pořadí i přesto složité, může se změřit, kolikrát je která podmínka splněna a pořadí automaticky určit tak, aby bylo vyhodnocováno například nejméně podmínek nebo aby celkový čas vyhodnocování podmínek byl co nejmenší.

Samozřejmě v případě, že nám tento systém spouštění metod nevyhovuje, můžeme si napsat svojí zástupnou metodu, která přesně určí, v jakém pořadí se metody budou prohledávat.

Metody objektů jsou sice nejdůležitějším druhem metod, nicméně přesto je potřeba, aby model rozvolněných objektů podporoval více druhů metod.

- běžné funkce - jsou proveditelné bez vazby na konkrétní objekt a mohou vracet atomické hodnoty, reference a seznamy nebo množiny referencí
- metody - jsou proveditelné pouze v kontextu konkrétního objektu a mohou vracet stejné hodnoty jako běžné funkce. Je možné volat metody i na množiny nebo seznamy objektů, a pak se metoda zavolá na všech objektech, které vyhovují podmínkám metod
- triggery - jsou speciálním případem metod. Jejich spuštění je automatické v okamžiku, kdy je splněna podmínka proveditelnosti v definici funkce
- agregační funkce - je proveditelná na množině objektů a mohou vracet stejné hodnoty jako běžné funkce

### 2.3.3 Vztahy mezi objekty

Je potřeba zaznamenávat nejen informace o vlastních objektech, ale i o vztazích mezi jednotlivými objekty. Tyto vztahy je možné realizovat buď pomocí odkazu jednoho objektu na jeden (pro binární vztahy) nebo více dalších objektů (v případě vztahů s větší aritou) a nebo je možné vztah realizovat pomocí vztahového objektu. Tento vztahový objekt se odkazuje na objekty, které ve vztahu vystupují, ale navíc může obsahovat další atributy (například u vztahu vyučující-předmět-učebna, který říká že daný vyučující učí daný předmět v dané učebně, je možné mít atribut čas, který říká, kdy výuka předmětu začíná). Tyto vztahové atributy by v případě přímého odkazování objektů mezi sebou mohly být matoucí. Muselo by se totiž vědět, které atributy se týkají vlastního objektu a které představují informace o realizovaném vztahu. Zda je nebo není objekt určen k reprezentaci vztahu mezi několika objekty, je dáno atributem isRelation objektu Object, jak je znázorněno na obrázku 2.1.

V pohledu na vztahy mezi objekty se model rozvolněných objektů od standardního objektového modelu neliší vůbec. Jenom je potřeba u realizace vztahů pomocí rozvolněných objektů dávat pozor, aby v případech vztahů s větší aritou, tyto vztahy neobsahovaly podmnožinu atributů, která je sama o sobě vztahem. Například u vztahu vyučující-předmět-učebna, není možné realizovat samostatný vztah vyučující-předmět, vyučující-učebna ani předmět-učebna, protože tyto vztahy už jsou obsažené vztahem vyučující-předmět-učebna. Řešení je samozřejmě jednoduché - stačí pro tyto požadované vztahy použít jiných atributů.

## 2.4 Podmínky metod

V modelu rozvolněných objektů je součástí definice metody zápis podmínek nejen na předávané parametry metody, ale i na objekt, na kterém může být metoda volána. Tyto podmínky se dají realizovat různými způsoby. V této části budou popsány dva z možných přístupů k této problematice. Podmínky by měli být co nejrychleji vyhodnotitelné, jelikož se budou typicky ověřovat před každým voláním metody.

### 2.4.1 Omezení dané množinou atributů

V tomto případě je možné zavolat metody jen za předpokladu, že daný objekt obsahuje všechny atributy, které má uvedené v podmínce. Tedy podmínka je libovolná podmnožina atributů, která vlastně odpovídá implicitní třídě. Přímý předek libovolné třídy obsahuje vždy o jeden atribut méně. Obecně předek třídy obsahuje vlastní podmnožinu atributů dané třídy. Zřejmě platí vztah, že pokud jeden objekt obsahuje podmnožinu atributů druhého objektu, tak i množina metod, které na prvním objektu jdou zavolat, je podmnožinou metod, které jdou zavolat na druhém objektu. Tedy, když je jeden objekt potomkem druhého objektu, tak může potomek volat všechny metody, které může volat jeho předek.

Uvažme aplikaci Evidence přestupků řidičů. Rozdíl mezi aktivním a pasivním řidičem se dá podmínkou vyjádřit nejlépe tak, že aktivní řidič má součet bodů menší než 12, tedy hodnota jednoho z jeho atributů je menší než 12, kdežto pasivní má splněnou negaci této podmínky. V případě, že součet není nikde uložený, ale musí se nějakým způsobem počítat, by podmínka byla ještě mnohem složitější. Dále uvažme aplikaci Úložiště výsledků testů. V této aplikaci se vyskytuje větší množství číselníků jako je například případ užití, testovací případ nebo požadavek. Tyto číselníky většinou obsahují stejné atributy jméno, kód a popis. Abychom od sebe rozlišili jednotlivé číselníky, tak je pořeba vytvořit nový atribut Type, po kterém budeme

v podmínce vyžadovat, aby měl konkrétní hodnotu. Pokud bychom chtěli tyto číselníky rozlišovat pomocí zde navrhovaného způsobu, tak bychom museli vytvořit atribut pro každý typ číselníku. Na těchto příkladech je vidět, že tato reprezentace podmínek je nedostačující a proto je potřeba vyjadřovací sílu podmínek zvýšit.

### 2.4.2 Omezení dané libovolnou podmínkou

Problémy, které vznikaly při definici podmínek jakožto podmnožin atributů, lze vyřešit použitím následujícího způsobu definice podmínek. Podmínka může vyžadovat nejen to, že atribut musí být definován (Age≠null) nebo zkráceně (Age), ale může požadovat i nedefinovanost atributu (Age==null), nebo omezení jeho hodnoty (Age==29), (Age≠29), (Age>29) nebo (Age<29). Pomocí logických spojek and(&&), or(||) a not(!) se mohou tyto podmínky ještě dále skládat, takže může vzniknout například podmínka pro číselník testovacích případů (Name && Code && Description && Type==TestCase). Pro přehlednost mohou mít podmínky své jméno. V libovolné podmínce se lze odkazovat na libovolnou jinou pojmenovanou podmínku. Tedy za předpokladu, že máme podmínku pro řidiče Driver, tak podmínka pro aktivního řidiče je (Driver && Points<12). Pokud jedna podmínka implikuje druhou podmínku, tak je třída reprezentovaná první podmínkou podtřídou třídy reprezentované druhou podmínkou.

## 2.5 Přístupová práva

Dalším požadavkem na aplikace, které používá více uživatelů, je existence přístupových práv. Někteří uživatelé potřebují mít přístup k více informacím a operacím než ostatní uživatelé, kteří naopak k těmto informacím či operacím přístup mít nesmějí. Ukazuje se, že navrhnout systém práv systému, který používá několik tisíc lidí, je vcelku složité. Díky tak vysokému počtu uživatelů není možné explicitně vyjmenovávat jednotlivá práva jednotlivých uživatelů. Většinou databázová vrstva neposkytuje dostatečné prostředky k užitečné definici práv, tak se používají práva až na aplikační úrovni. Dalším prakticky nutným požadavkem na práva je existence implicitních práv, takže například učitel může editovat sylabus předmětu, který učí, bez toho, aby v systému měl někde toto právo explicitně uvedeno. Nicméně tato implicitní práva se stejně nakonec ukládají do databáze, aby mohla být kontrolována.

V praxi se často používají systémy práv, které umožňují definovat různá práva pro určitý subjekt (kdo má mít právo) a objekt (na jaký objekt by měl mít právo). Konkrétní subjekt má tedy konkrétní právo na konkrétní objekt. Jelikož mohou mít aplikace různé požadavky na systém práv, tak by rozvolněné objekty měly jen umožnit se na práva ptát a to při spouštění jednotlivých metod nebo přímo jako součást dotazů na databázi. Jednotlivé aplikace by si pak vybraly, jakým způsobem chtějí práva využívat. Mělo by být možné práva nastavovat pro jednotlivé atributy jednotlivých objektů a to zvlášť pro to, jestli je možné atribut číst, zapisovat, přidat nebo odebrat. Jelikož jednotlivé metody obsahují podmínky na atributy, které se v metodě budou používat, tak v případě, že součástí podmínek je i informace, zda se jednotlivé atributy budou přidávat, ubírat, modifikovat nebo jen číst, je možné automaticky práva při volání metod kontrolovat. Navíc by musela být vždy přístupná informace, pro koho se práva kontrolují. Právo by mohlo být buď přímo na spuštění dané metody, nebo by se v definici metody určilo, jaká práva se musí ověřit, aby šla metoda spustit.

## 2.6 Dotazy

Jednou z nejpoužívanějších funkcí při práci s rozvolněnými objekty je vyhledávání objektů, které splňují určitá kritéria. Dotaz je boolská formule, která obsahuje seznam atributů, kde pro každý z nich lze volitelně navíc určit omezení na hodnoty, na přístupová práva nebo čas, kdy má hodnota atributu platit.

Jelikož dotazy i podmínky metod mají stejnou syntaxi, je možné je kdykoliv v programu zaměňovat. V článku [1] je navrženo, aby každý dotaz vracel množinu objektů. Když se podíváme, jaké jsou v Javě požadavky na množiny[4], bude lepší pro vyšší srozumitelnost v následujícím textu rozlišovat mezi pojmy množina a seznam. Množina může obsahuje každý prvek maximálně jednou, kdežto seznam má definované pořadí a může obsahovat jeden prvek vícekrát. Z praktických důvodů je bez újmy na obecnosti lepší, aby dotazy vracely seznamy, jelikož vždy při práci s objekty bude toto řešení dostačující. Práce se seznamy implementovanými jako spojové nebo rozšiřitelné pole je řádově rychlejší, protože se nemusí kontrolovat přítomnost objektu v množině a navíc tím získáme automaticky možnost výsledek dotazu setřídit, což by na množině nešlo a musela by se převést na seznam. Sice na množinách lze provádět efektivně operace průnik, sjednocení nebo rozdíl, ale po setřídění seznamu lze tyto operace provádět ještě rychleji. Seznam také lépe vystihuje typickou práci s výsledkem dotazu - průchod všemi prvky a s každým prvkem možnost něco vykonat. Každou referenci lze považovat za dotaz vracející jeden objekt. Dotaz se může pokládat nad databází nebo nad jiným seznamem objektů, což může být i výsledek jiného dotazu.

## 2.7 Manipulace s objekty

### 2.7.1 Vytváření, aktualizace a smazání

Vytváření i smazání objektů bude prováděno explicitně. Podobně bude realizována i změna, přidání a odebrání jednotlivých atributů objektů. Dále se bude požadovat, aby bylo možné objekty kopírovat třemi způsoby:

- minimalistická kopie - kopírují se pouze hodnoty atributů atomických datových typů
- standardní kopie - kopírují se hodnoty atributů atomických datových typů a reference se kopírují beze změn
- maximalistická kopie - kopírují se i všechny odkazy objektem odkazované. Zde je potřeba dávat pozor, co to vlastně znamená. V praxi často nastane případ, že objekt odkazuje na další objekt, ale ten je jednou potřeba zkopírovat a jednou převzít už existující. Například když se kopíruje dokument, který se skládá z listů, tak tam je většinou potřeba kopírovat dokument i jednotlivé listy. Za ten stejný dokument je někdo kompetentní, ale v případě kopie dokumentu se kompetentní role nesmí vytvářet nová, ale musí se převzít už existující. Ve standardním objektovém modelu by se v této situaci dokument odkazoval na roli, která by se tedy kopírovala, ale neměla by, a listy by se odkazovaly na dokument, tedy by se nekopírovaly, ale měly by se kopírovat. Navíc v současné reprezentaci odkazů je problém, že se vždy odkazuje dokument na jeho listy nebo listy na dokument, takže kdyby se automaticky procházely odkazy jedním směrem, tak občas by se potřebné objekty vykopírovaly a občas by díky opačnému směru odkazu

[4] http://java.sun.com/javase/6/docs/api/java/util/Set.html

zůstaly nenalezeny. Proto je vhodné podporu tohoto druhu kopírování nechat na uživateli, ať si sám vybere co všechno se má zkopírovat a co převzít. Jedno z řešení je kopírovat pouze to co si uživatel předtím načte pomocí dotazu z databáze, případně mu poskytnout nějaký nástroj, kterým by si mohl dodatečně zvolit, které objekty se mají převzít a které kopírovat.

### 2.7.2 Množiny a seznamy

Častý požadavek při práci s objekty je ukládání těchto objektů do seznamů a množin, tak jak byli popsány v části 2.6. U množin i seznamů rozvolněných objektů je potřeba umět procházet těmito strukturami, abychom byli schopní zavolat metodu na každém objektu z dané struktury. Navíc je potřeba umět zjistit, zda daný prvek je v množině či seznamu obsažen. Musí být možné tyto struktury mezi sebou navzájem převádět. U množin navíc předpokládáme podporu pro operace průnik, sjednocení a rozdíl.

### 2.7.3 Vlákna

Jelikož k objektům v paměti může přistupovat současně více vláken, bude potřeba navrhnout podporu pro paralelní přístup více vláken, aby v okamžiku, kdy s objektem už jedno vlákno pracuje, tak ostatní vlákna měla přístup k danému objektu odepřen.

### 2.7.4 Persistence

Pro zajištění konzistence je potřeba nabídnout podporu transakcí. V případě, že by měly aplikace přistupovat do více databází najednou, je potřeba použít distribuované transakce.

Většina frameworků poskytující persitenci objektů nabízí nějakou formu cache, která umožňuje nenačítat potřebná data z databáze, ale z paměti, kde se ukládají dříve použité objekty. Realizace bývá mapou, kde klíčem je id objektu a hodnotou daný objekt se všemi načtenými atributy. Problém u rozvolněných objektů je ten, že většinou se nebude pracovat se všemi atributy objektů, a tak by většinou docházelo k situaci, že i když je objekt v cachi obsažen, tak neobsahuje všechny potřebné atributy. Proto existence výhradní cache nebude vyžadována, ale je možné, že architektura realizace rozvolněných objektů bude schopná nějakou cache nabídnout, jelikož v sobě uložené objekty bude stejně obsahovat. V případě nutnosti je ale vždy možné si nějakou vlastní cache na aplikační úrovni vytvořit.

Protože k datům bude přistupovat více uživatelů současně, je potřeba nabídnout nějakou podporu ochrany dat, aby si připojení uživatelé vzájemně neškodili. Současné relační databáze nabízí většinou pro tyto účely podporu pro různé úrovně izolace transakcí, které jsou definované normou, takže převzetí této podpory bude určitě vyžadováno. Úrovně izolace transakcí přímo definují k jakým všem problémům může při přístupu více uživatelů docházet.

Dále současné relační databáze podporují různé formy zámků. Zámky lze získávat buď sdílené nebo exklusivní a zamykat lze mimo jiné buď jednotlivé záznamy nebo celé tabulky. Jelikož rozvolněné objekty mají jednotlivé atributy uloženy v různých tabulkách, je velmi výhodné nabídnout podporu pro dva druhy zámků:

- sdílené zamykání celých objektů - je potřeba pro vytvoření kopie, kdy ostatní uživatelé nesmí měnit hodnoty atributů právě se kopírujícího objektu, ale mohou je číst
- exklusivní zamykání jednotlivých atributů objektů - hodí se v případě, že potřebujeme zapsat hodnotu atributu, která byla získána na základě předtím načtené hodnoty,

a zároveň nechceme, aby ostatní uživatelé mohli načtenou hodnotu změnit. Jelikož se zamyká každý atribut zvlášť, je velká pravděpodobnost, že ostatní uživatelé budou měnit jiné než zamčené atributy a nebude tedy docházet k tak častému čekání na uvolnění zámků

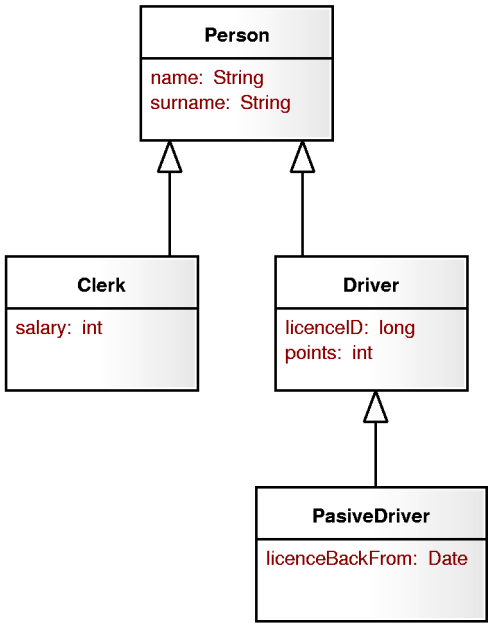
Pokud se budou atributy objektů zamykat vždy ve stejném pořadí, tak se zaručí, že nikdy nedojde k deadlocku. Samozřejmě v případě, že je zamykáno více atributů najednou, tak i jednotlivé atributy objektů se musí zamykat v pořadí podle id objektu, kterému atribut patří, jinak by k deadlocku mohlo docházet.

Ne všechny hodnoty atributy objektů je potřeba ukládat do databáze. Proto je potřeba přidat podporu pro transientní atributy, jejichž hodnota se nikdy do databáze ukládat nebude.

### 2.7.5 Systémový katalog

Pokud poběží více aplikací nad stejnou databází, tak se může stát, že některá aplikace používá atributy, které ostatní aplikace neznají. V tomto případě by nebylo možné vytvořit kopii objektu, protože by aplikace neznala všechny potřebné atributy. Řešením je existence systémového katalogu, který by popisoval všechny existující atributy. Realizaci katalogu by bylo nejvhodnější provést pomocí rozvolněných objektů, protože pak by systémový katalog mohl popisovat i atributy, ze kterých se sám skládá.

## 2.8 Datový model




Obrázek 2.4: Ukázkový diagram tříd

Na obrázku 2.4 je znázorněna jednoduchá hierchie tříd z ukázkové aplikace Evidence přestupků řidičů, na které si ukážeme, jak lze takováto data uložit do relační databáze. PasiveDriver představuje řidiče, kterému byl odebrán řidičský průkaz.

| Class | Name | Surname | Salary | LicenceID | Points | LicenceBackFrom |
|---|---|---|---|---|---|---|
| **Person** | Michal | Lulka | | | | |
| **Clerk** | Radek | Klakson | 25000 | | | |
| **Driver** | Marek | Dub | | 345897 | 11 | |
| **PasiveDriver** | Karel | Buchta | | 465879 | 12 | 1.10.2010 |

Obrázek 2.5: Seznam objektů k uložení do relační databáze

Na obrázku 2.5 je seznam objektů, které chceme uložit do relační databáze. V levém sloupečku je uvedena třída objektu a ve sloupečcích vpravo jsou uvedeny hodnoty známých atributů. K ukládání hierarchie tříd do relační databáze se používá několik přístupů. Každý z těchto způsobů je optimalizovaný na jiné druhy dotazů. Největší rozdíl je pak v tom, jestli vyhledáváme objekty jen z konkrétních tříd, nebo chceme vyhledávat objekty ze zvolených tříd včetně jejich podtříd. Další rozdíl je v tom, zda povolíme do databáze ukládat null hodnoty.

**Person**

| ID | Name | Surname |
|---|---|---|
| 1 | Michal | Lulka |

**Clerk**

| ID | Name | Surname | Salary |
|---|---|---|---|
| 1 | Radek | Klakson | 25000 |

**Driver**

| ID | Name | Surname | LicenceID | Points |
|---|---|---|---|---|
| 1 | Marek | Dub | 345897 | 11 |

**PasiveDriver**

| ID | Name | Surname | LicenceID | Points | LicenceBackFrom |
|---|---|---|---|---|---|
| 1 | Karel | Buchta | 465879 | 12 | 1.10.2010 |

Obrázek 2.6: Mapování hierarchie tříd do relační databáze pomocí horizontálního mapování

Na obrázku 2.6 je ukázka uložení požadovaných dat pomocí horizontálního mapování. Objekty z každé třídy jsou uloženy do samostatné tabulky včetně všech zděděných atributů. Vkládání objektů je velmi rychlé, neboť se přistupuje pouze do jedné tabulky. Dotazy na objekty ze třídy včetně podtříd jsou ale nutně vyhodnocovány přes spojení se všemi potomky dané třídy. Kdybychom chtěli použít toto mapování pro rozvolněné objekty, tak potřebujeme pro každou podmnožinu atributů vytvořit zvláštní tabulku (abychom mohli uložit všechny atributy, které objekt obsahuje), tedy pro k atributů potřebujeme $2^k$ tabulek. Při přidání nebo odebrání atributu jednoduše přesuneme objekt z jedné tabulky do druhé. Problém ale nastává pří polymorfních dotazech, kdy musíme prohledávat všech $2^{\text{k-počet atributů definovaných třídou}}$ potomků třídy.

**Person**

| ID | Name | Surname |
|---|---|---|
| 1 | Michal | Lulka |
| 2 | Radek | Klakson |
| 3 | Marek | Dub |
| 4 | Karel | Buchta |

**Clerk**

| ID | Salary |
|---|---|
| 2 | 25000 |

**Driver**

| ID | LicenceID | Points |
|---|---|---|
| 3 | 345897 | 11 |
| 4 | 465879 | 12 |

**PasiveDriver**

| ID | LicenceBackFrom |
|---|---|
| 4 | 1.10.2010 |

Obrázek 2.7: Mapování hierarchie tříd do relační databáze pomocí vertikálního mapování

Na obrázku 2.7 je vidět, jak vertikální mapování uloží výše zmíněné objekty do databáze. Atributy objektů z každé třídy jsou uloženy do samostatné tabulky zvlášť od zděděných atributů. Vkládání objektů vyvolá řetězec vložení, neboť je potřeba uložit atributy do všech předků třídy objektu, který se vkládá. Dotazy na objekty ze třídy včetně podtříd ale přistupují pouze do jediné tabulky. Pro ukládání rozvolněných objektů podobně jako pro horizontální mapování potřebujeme pro každou podmnožinu atributů vytvořit zvláštní tabulku. Tabulka s jediným atributem obsahuje id objektu a jeho hodnotu. Ostatni tabulky reprezentují třídy, které všechny své atributy zdědily z těchto tabulek, tedy obsahují jen id objektu. Každá třída má $2^{\text{počet atributů definovaných třídou}}$-1 přímých předků. Při přidání atributu musíme přidat id objektu do všech odpovídajících tabulek, kterých je ale $2^{\text{počet atributů definovaných třídou}}$.

**Person**

| ID | Class | Name | Surname | Salary | Licence ID | Points | LicenceBackFrom |
|---|---|---|---|---|---|---|---|
| 1 | Person | Michal | Lulka | null | null | null | null |
| 2 | Clerk | Radek | Klakson | 25000 | null | null | null |
| 3 | Driver | Marek | Dub | null | 345897 | 11 | null |
| 4 | PasiveDriver | Karel | Buchta | null | 465879 | 12 | 1.10.2010 |

Obrázek 2.8: Mapování hierarchie tříd do relační databáze pomocí filtrovaného mapování

Posledním často používaným mapováním je filtrované mapování, jehož způsob uložení požadovaných objektů je vidět na obrázku výše. Toto mapovámí používá pouze jednu tabulku pro všechny objekty. Výhodou jsou rychlé polymorfní dotazy i vkládání bez řetězových reakcí nebo joinů. Hlavní nevýhodou je, že většina hodnot je null a nutnost přidat další sloupec, který zaznamenává přesnou třídu objektu. Dále v případě, že pridáváme nový atribut, ja potřeba provést nastavení hodnoty null pro všechny existující objekty. Pro rozvolněné objekty by toto mapování šlo použít, ale díky požadavku, aby rozvolněné objekty uměly ukládat hodnoty atributů s časovou platností raději použijeme speciální mapování. Při změně hodnoty jediného atributu by totiž bylo potřeba kopírovat celý řádek tabulky.

**Name**

| ID | Value |
|---|---|
| 1 | Michal |
| 2 | Radek |
| 3 | Marek |
| 4 | Karel |

**Surname**

| ID | Value |
|---|---|
| 1 | Lulka |
| 2 | Klakson |
| 3 | Dub |
| 4 | Buchta |

**Salary**

| ID | Value |
|---|---|
| 2 | 25000 |

**LicenceID**

| ID | Value |
|---|---|
| 3 | 345897 |
| 4 | 465879 |

**Points**

| ID | Points |
|---|---|
| 3 | 11 |
| 4 | 12 |

**LicenceBackFrom**

| ID | Value |
|---|---|
| 4 | 1.10.2010 |

Obrázek 2.9: Mapování hierarchie tříd do relační databáze pomocí rozvolněných objektů

Na obrázku 2.9 je vidět, jak mapování rozvolněných objektů po sloupcích uloží požadované objekty do relační databáze. Toto mapování vytvoří pro každý atribut jednu tabulku, která vždy bude obsahovat dva sloupce. První sloupec je vždy id objektu, kterému atribut patří. Druhý sloupec obsahuje hodnotu daného atributu. V případě atributu s časovou platností by tabulka navíc obsahovala sloupce s id řádku, časem od kdy a časem do kdy je daná hodnota atributu platná.

Vhodné by bylo i vytvoření jedné tabulky, která by obsahovala jen ID všech objektů. Na tyto by se šlo odkazovat v integritních omezeních a například mazání objektu by tak stačilo udělat vždy jen z této tabulky a zbytek by databáze odstranila sama. Problémem by byl určitě přístup k více atributům objektu, jelikož standardním způsobem nejde vytvořit index přes více tabulek. Kdyby aplikace vyžadovaly načítat objekty se všemi atributy, které objekty obsahují, bylo by rozumné vytvořit atribut bitového pole, ve kterém by byly zaevidovány všechny definované atributy objektu. Sice by to zpomalovalo všechno vkládání objektů, ale zase by šlo načítat objekty včetně všech jejich atributů. Tento atribut by mohl nahradit tabulku s ID všech objektů, protože hodnotu tohoto atributu by měly automaticky nastavenu všechny objekty. Většinou se ale přistupuje jen k několika málo atributům objektu, takže je možné, že některé dotazy budou vyhodnocovány rychleji, než dotazy nad méně tabulkami s více sloupci, kde nepotřebné atributy zpomalují načítání potřebných atributů.

## 2.9 Shrnutí požadavků

Nakonec ještě přehledně na jednom místě sepišme seznam všech požadavků, které vyplývají z předchozího textu. Požadavky jsou z větší části seřazeny podle důležitosti.

- Srovnatelné možnosti a výkon se standardním objektovým modelem
- Jednoduchá manipulace s objekty, které mají rozdílné množiny definovaných atributů. Možnost ukládat objekty do seznamů a množin. Podpora pro současný přístup k objektům z více vláken
- Podmínky, které musí být splněny, aby bylo možné metodu zavolat, jsou součástí definice této metody. Je možné zavolat metodu na všech objektech, které v danou chvíli splňují podmínky metody. Podmínky metod se mohou skládat z dalších existujících podmínek. Je možné metodu nevolat a jen ověřit, zda s předanými parametry ji lze zavolat. Podpora pro skupiny podmínek, aby šlo při spouštění aplikace zvolit, jaké všechny skupiny podmínek se mají kontrolovat

- Existence zástupných metod, které konkrétní implementaci metody zvolí až při volání metody v závislosti na aktuálně předaných parametrech
- Podpora pro časovou platnost hodnot atributů jednotlivých objektů
- Podpora pro přístupová práva uživatelů k jednotlivým metodám zavolaných nad určitým objektem nebo přímo k jednotlivým atributům objektů
- Možnost automatického zavolání metody po splnění podmínky
- Podpora pro persistenci objektů, transientní atributy, transakce, úrovně izolací transakcí, sdílené zámky pro objekty a exklusivní zámky pro atributy objektů
- Podpora pro objektově orientované dotazy a možnost manipulace se všemi objekty uloženými v databázi najednou.
- Existence systémového katalogu
- Možnost snadné realizace nad libovolnou relační databází

# Kapitola 3

# Realizace rozvolněných objektů v Javě

V této kapitole bude popsána realizace konceptů rozvolněných objektů v programovacím jazyku Java. Bude zde ukázáno, jakým způsobem je vhodné reprezentovat rozvolněné objekty včetně způsobu jejich uložení do seznamů a množin. Dále zde bude vyřešena otázka persistence rozvolněných objektů se speciální podporou pro transakce a zamykání rozvolněných objektů v databázi. Součástí této kapitoly je i popis systémového katalogu. Na konci kapitoly budou zmíněné méně důležité implementační záležitosti jako je podpora pro načítání parametrů, logování, jmenné konvence, struktura projektů a popis použitých knihoven. Dále cílem této kapitoly bude nalezení odpovědí na následující otázky:

- Jaký je nejvhodnější zápis pro omezení parametrů jednotlivých metod a pro omezení objektů, na kterých mají být metody volány.
- Jak optimálně zařídit, aby se tato omezení kontrolovala za běhu aplikace.
- Jakým způsobem pro rozvolněné objekty, včetně jejich seznamů a množin, jednoduše poskytnout možnost volat metody, které splňují všechny omezení a mohou být tedy bezpečně volány.
- Jak nejlépe vyhledávat v databázi rozvolněné objekty, které mají požadované vlastnosti.

## 3.1 Rozvolněný objekt

Jeden z prvních, ale zároveň jeden z nejdůležitějších problémů, které je potřeba vyřešit, je reprezentace vlastních rozvolněných objektů v Javě. Nejvíce nás budou zajímat rychlost prováděných operací a paměť, kterou objekty zabírají. V následujícím textu budeme rozlišovat mezi primitivními datovými typy a objektovými typy. Primitivní datové typy přímo obsahují data, takže jsou optimální z hlediska paměťových nároků. Objektové typy fungují tak, že do proměné je ukládán jen odkaz na vlastní objekt. V Javě má tento odkaz vždy 4B. Tedy bez použití cache už vlastní použití objektových typů znamená 4B navíc díky nutnosti se na daný objekt odkazovat. V Javě existuje jeden předek všech tříd, ze kterého všechny ostatní třídy dědí základní funkcionalitu. Touto třídou je třída Object.

V případě, že sada atributů je definována třídou objektu, je přirozené přímo v definici třídy tyto parametry vyjmenovat. Zkusme se podívat, jaké jsou paměťové nároky na instance tříd, které přímo v definici obsahují vyjmenované všechny atributy, jež je možné do třídy uložit.

| Popis třídy objektu | Zabraná paměť | Čas vytvoření instance | Čas vytvoření instance do pole s velikostí 1000000 |
|---|---|---|---|
| **Object** | 8B | 12ns | 22ns |
| **Třída bez atributů** | 8B | 13ns | 22ns |
| **Třída obsahující int** | 16B | 18ns | 25ns |
| **Třída obsahující String** | 16B | 19ns | 25ns |
| **Třída obsahující int a String** | 16B | 18ns | 25ns |
| **Třída obsahující dvakrát int** | 16B | 19ns | 26ns |
| **Třída obsahující dvakrát String** | 16B | 18ns | 24ns |
| **Třída obsahující třikrát int** | 24B | 25ns | 33ns |
| **Třída obsahující čtyřikrát int** | 24B | 26ns | 32ns |
| **Třída obsahující pětkrát int** | 32B | 35ns | 40ns |

Obrázek 3.1: Vytvoření instancí tříd s pevně danými atributy

Na obrázku 3.1 jsou znázorněny paměťové nároky a rychlost tvorby instancí tříd s pevně daným výčtem atributů. V prvním sloupci jsou popsány různé třídy objektů převážně rozdělené podle počtu a typu atributů, které obsahují. Primitivní datový typ reprezentuje datový typ int a objektový typ je reprezentován typem String. Ve druhém sloupci je uvedeno kolik paměti zabírá vytvoření objektu dané třídy. Ve třetím sloupci jsou uvedeny časy potřebné k vytvoření objektů. V posledním sloupci se k času vytvoření ještě přičítá doba uložení do pole, což simuluje běžný požadavek na trvalé ponechání objektu v paměti. Čas potřebný k tomuto uložení je u větších objektů zanedbatelný, takže dále budeme předpokládat jen vytváření trvalých objektů. Musíme ovšem dávat pozor, aby nedocházelo k příliš velké alokaci nové paměti, aby tím nebyly ovlivněny výsledky testů. Časy jsou vždy průměr z pěti spuštění, přičemž v každém spuštění se vytvořilo milión objektů. Vytváření objektů bylo prováděno s verzí Javy 1.6.0_13 na procesoru AMD Athlon 64 3200+ taktovaném na 2GHz[1]. Přesto že výsledky testů jsou měřeny v ns, je přesnost měření velmi dobrá - v převážné většině případů +/- 2ns.

Z výsledků je patrné, že z hlediska rychlosti tvorby objektů záleží pouze na počtu atributů (které se zatím v tomto testu neinicializují požadovanými hodnotami, tedy není započítána rychlost tvorby ani velikost objektu typu String, ale počítá se jen velikost odkazu na něj). Přesněji řečeno záleží pouze na velikosti paměti, kterou objekty zabírají. Zajímavé jsou paměťové nároky jednotlivých tříd. Pro úplnost uveďme, že datový typ int sám o sobě zabírá 4B. Třída, která neobsahuje žádné atributy, zabere 8B. Vždy, když se přidají do třídy nějaké atributy, tak se navíc alokuje pro každý objekt takové množství bytů, aby se všechny přidané atributy do alokované paměti vešly a zároveň aby byl počet přidaných bytů celým násobkem 8. Takže pro třídu s jediným atributem zabírajícím 4B potřebujeme 16B a pro třídu se třemi atributy, kde každý z nich zabírá 4B, potřebujeme celkem 24B.

[1] Pro podrobnosti testování se lze podívat do projektu 21_SRO do package test.sro.core.object, kde jsou testy v souborech končících řetězcem Test.

Je vidět, že objektové zabalení dat je oproti primitivním datovým typům paměťově velmi náročné. Například pro uložení jediné hodnoty datového typu int, který obsahuje jen 4B dat, potřebujeme 20B. 16B je pro objekt, ve kterém je uložená hodnota typu int, a 4B potřebujeme na odkaz na tento objekt. Na druhou stranu do zmíněných 16B objektu se vejde ještě další atribut, který zabírá maximálně 4B. S narůstajícím počtem atributů jsou ale tyto paměťové požadavky z hlediska celkové velikosti objektu čím dál více přijatelnější.

Jelikož výsledky pro typy String i int jsou velmi podobné, budeme v následujícím textu uvažovat už jen datový typ int. Vytvořené objekty, které budou rozvolněné objekty v sobě obsahovat jakožto hodnoty atributů, se musí vytvářet stejným způsobem i ve standardním objektovém modelu, takže jejich velikost i rychlost vytváření můžeme zatím ignorovat.

| **Popis třídy objektu** | **Čas získání hodnoty atributu** | **Čas nastavení hodnoty atributu** | **Čas vytvoření včetně inicializace v konstruktoru** | **Čas vytvoření a nastavení všech hodnot atributů** |
|---|---|---|---|---|
| **Třída obsahující public final int** | 10ns | n/a | 25ns | n/a |
| **Třída obsahující public int** | 11ns | 13ns | 25ns | 26ns |
| **Třída obsahující int** | 11ns | 17ns | 25ns | 32ns |
| **Třída obsahující dvakrát int** | 12ns | 16ns | 26ns | 44ns |
| **Třída obsahující čtyřikrát int** | 15ns | 19ns | 27ns | 66ns |
| **Třída obsahující osmkrát int** | 20ns | 29ns | 51ns | 98ns |
| **Třída obsahující šestnáctkrát int** | 26ns | 55ns | 83ns | 178ns |
| **Třída obsahující třicetdvakrát int** | 25ns | 52ns | 196ns | 402ns |
| **Třída obsahující třicetdvakrát int synchronizovaná kritická sekce** | 89ns | 91ns | 189ns | 417ns |
| **Třída obsahující třicetdvakrát int synchronizovaná metoda** | 92ns | 95ns | 194ns | 428ns |

Obrázek 3.2: Čtení a modifikace instancí tříd s pevně danými atributy

Na obrázku 3.2 jsou znázorněny výsledky testů nastavování a čtení hodnot atributů objektů. Tentokrát se jedná o průměry z pěti spuštění, kde každá operace proběhla stotisíckrát. Abychom zvýšili přesnost měření, bylo potřeba snížit počet provádění operací, jinak by se měřilo spíše jakou rychlostí umí Java přidělovat paměť. To je dáno tím, že při těchto testech používáme řádově větší objekty než v předchozím případě, takže se pamět spotřebovává rychleji. Při vytvoření většího počtu objektů je ale vždy nutno počítat s tím, že časy se budou díky paměťovým požadavkům objektů výrazně zvyšovat. Ve druhém a třetím sloupci se můžeme převědčit, že když máme pevně dané atributy třídy, tak při jejich čtení nebo nastavování výrazně nezáleží na jejich počtu. V prvních dvou řádcích jsou ještě zvlášť otestovány případy, kdy jsou atributy třídy veřejné (modifikátor final znamená, že hodnota se po nastavení nesmí měnit) a přistupuje se přímo k nim. V ostatních případech byly atributy

nastavovány a čteny jen pomocí metod.

Ve čtvrtém a pátém sloupci můžeme porovnat dva způsoby inicializace objektů. Ve čtvrtém sloupci jsou konstruktoru předány všechny hodnoty atributů, kdežto v pátém sloupci se použije bezparametrický konstruktor a následně se zavolají všechny metody, které jednotlivé atributy nastavují. Se zvyšujícím se počtem atributů se výrazně zvyšuje i doba potřebná k vytvoření inicializovaného objektu druhým zmíněným způsobem.

V posledních dvou řádcích můžeme vidět, jaká je rychlost požadovaných operací v případě, že budeme přistupovat k datům exklusivně. Trošku lepších výsledků dosáhlo použití sekce synchronized jen kolem kritické části kódu (což je způsob, který se doporučuje všude v literatuře a sami autoři Javy ho používají ve svých knihovnách), nicméně i použití klíčového slova synchronized na celou metodu má srovnatelný výkon. Každopádně při exklusivním přístupu se rychlost nastavování a čtení atributů výrazně zmenšila.

Zkusme navrhnout rozvolněné objekty tak, abychom se co nejvíce přiblížili paměťovým nárokům a rychlosti operací tříd s pevně danými atributy. Kdybychom chtěli použít stejným způsobem třídy s pevně vyjmenovanými atributy, tak v případě nastavení nového atributu musíme vytvořit novou instanci. Problémem je však získání třídy této instance. Potřebovali bychom totiž mít k dispozici pro každou kombinaci definovaných atributů jednu třídu, tedy celkem $2^{\text{počet všech atributů}}$ tříd. Navíc nalezení nebo vytvoření té správné třídy a vytvoření nové instance se stavem té současné by nebylo časově zanedbatelné. Také vytvářet novou instanci po zavolání metody na nastavení hodnoty atributu by nebylo moc uživatelsky přívětivé.

Budeme proto muset použít nějaký další objekt, do kterého potřebné hodnoty atributů uložíme. Z hlediska rychlosti je nejvýhodnější použít mapu, kde klíčem je atribut a hodnotou hodnota daného atributu. Z hlediska paměťových nároků je nejlepším řešením použít pole.

| **Počet atributů třídy typu int** | **Čas získání hodnoty atributu** | **Čas nastavení hodnoty nového atributu** | **Čas nastavení hodnoty známého atributu** | **Čas vytvoření včetně inicializace v konstruktoru** | **Čas vytvoření a nastavení všech hodnot atributů** |
|---|---|---|---|---|---|
| **1** | 30ns/30ns | 106ns/106ns | 57ns/57ns | 30ns/58ns | 156ns |
| **2** | 36ns/37ns | 107ns/152ns | 56ns/60ns | 29ns/71ns | 296ns |
| **4** | 41ns/54ns | 107ns/181ns | 59ns/61ns | 31ns/89ns | 623ns |
| **8** | 53ns/65ns | 108ns/218ns | 68ns/75ns | 30ns/133ns | 1416ns |
| **16** | 80ns/95ns | 106ns/322ns | 92ns/107ns | 32ns/230ns | 4527ns |
| **32** | 104ns/188ns | 109ns/650ns | 120ns/208ns | 30ns/502ns | 11818ns |
| **32 Integer** | 147ns/190ns | 115ns/641ns | 162ns/201ns | 32ns/677ns | 13060ns |

Obrázek 3.3: Čtení a modifikace instancí tříd s atributy v poli

Na obrázku 3.3 jsou uvedeny rychlosti operací ve třídě, která ukládá hodnoty atributů do pole. Ve třídě existují dvě pole. V prvním jsou uloženy názvy atributů a ve druhém jsou uloženy hodnoty atributů. Pole s hodnotami atributů ukládá datový typ int a v posledním řádku můžeme tento přístup porovnat s testem, kde se používalo pole s objektovým typem Integer. Většina testů

je provedena pro nejlepší případ (první hodnota v buňce) a pro nejhorší případ (druhá hodnota v buňce).

Ve druhém sloupci je čas načítání hodnot atributů. První hodnota je přístup k prvnímu indexu pole a druhá hodnota měří rychlost přístupu k poslednímu indexu pole. Tyto časy jsou velmi dobré pro nízké počty atributů, s vzrůstajícím počtem atributů se rychlost přístupu k atributům samozřejmě snižuje. Zajímavé je, že se zvyšující se velikostí pole se zvyšuje doba potřebná k přístupu k jeho prvnímu prvku.

Nastavování hodnot atributů je měřeno dokonce čtyřmi způsoby. Musíme rozlišit, zda atribut je nebo není obsažen v poli. V případě, že není, obě datová pole objektu se zkopírují do nově vytvořených polí. Opět je zřejmé, že s větším polem se operace budou provádět pomaleji. Nejhorší případ u nastavování nového atributu znamená, že máme definovaných n-1 atributů a musíme nastavit hodnotu atributu n. Nejdříve se prohledá n-1 názvů atributů, pak se zjistím že atribut není nastaven, následně se zkopírují dvě pole velikosti n-1 a teprve pak je nový prvek vložen. Toto je největší slabina tohoto řešení. Výsledky tohoto testu jsou znázorněny ve druhé hodnotě ve třetím sloupci. Kdybychom netrvali na minimálních paměťových nárocích, tak můžeme použít rozšiřitelné pole se zhruba stejným výkonem, ale některé indexy pole mohou zůstat nevyužity. Ušetříme tím ale většinu kopírování polí v případě, že nastavujeme hodnotu zatím nenastaveného atributu.

Vytváření objektů bylo testováno třemi způsoby. V prvních dvou případech se vytvořil objekt a do konstruktoru se mu zadaly názvy a hodnoty nastavených parametrů. První případ předpokládá, že tyto hodnoty už známe předem, a druhý jejich vytvoření započitává do doby vytvoření objektu. V praxi většinou bude nastávat ten horší případ. Poslední varianta vytváření objektů je vytvoření objektu bez nastavených atributů a postupné zavolání metod, které nastaví hodnoty všech známých atributů. Tento druh inicializace dopadl výrazně nejhůře, v porovnání s předchozími způsoby je zhruba dvacetkrát pomalejší.

Instance této konkrétní testované realizace třídy s polem zabírala po vytvoření 48B (16B objekt a dvě pole - pole spotřebuje 12B plus objem svého obsahu). Za každé další dva definované atributy vždy potřebná paměť vzrostla o 16B. Celkové paměťové požadavky byly tedy zhruba dvakrát větší než byl datový obsah třídy. Častým dotazem na databázi je vrácení objektů, které mají všechny definované stejné atributy. Pak by bylo možné pole s názvy atributů sdílet.

Nejpoužívanější datové typy v praxi jsou určitě int a String. V tomto testu bylo použito pole typu int, ale budeme potřebovat umět ukládat i objektové typy. Kdybychom měli jedno pole objektů, tak máme sice optimální řešení pro řetězce, ale čísla bychom museli ukládat do wrapperu, který zabírá 16B. Jedno z řešení je vytvořit jedno speciální pole pro čísla. Tady je problém ten, že prázdné pole v Javě zabírá 16B. Výhodou by bylo naopak rychlejší hledání čísel, jelikož by se hledala jen ve svém menším poli. Druhým řešením pro ukládání čísel je uživatelem definovaná cache. V případě, že bude používat více čísel, zvětší si cache, když čísla bude používat méně, nevadí, že budou zabírat více paměti. Každé z těchto řešení se hodí pro jiné situace, ale žádné z nich není obecně lepší.

Řešení s nejrychlejším získáváním a nastavováním hodnot atributů by mělo být pomocí asociativního pole (neboli mapy) implementovaného pomocí třídy HashMap. Otázka je, jak bude toto řešení vhodné pro objekty s málo nastavenými atributy. Tato realizace mapy je pomocí procesu zvaného hašování. Pro vkládaný nebo hledaný atribut se na základě jeho hodnoty spočítá hašovací kód. Na jeho základě se určí index v poli, od kterého se bude atribut hledat. Výhodou je, že se nemusí prohledávat celé pole, ale většinou stačí prohledat jeho zanedbatelnou část.

| Počet atributů třídy typu int | Čas získání hodnoty atributu | Čas nastavení hodnoty nového atributu | Čas nastavení hodnoty známého atributu | Čas vytvoření včetně inicializace v konstruktoru | Čas vytvoření a nastavení všech hodnot atributů |
|---|---|---|---|---|---|
| **1** | 72ns | 176ns | 95ns | 264ns | 276ns |
| **2** | 100ns | 196ns | 133ns | 366ns | 470ns |
| **4** | 180ns | 258ns | 189ns | 582ns | 682ns |
| **8** | 241ns | 302ns | 275ns | 1000ns | 1147ns |
| **16** | 253ns | 276ns | 309ns | 5239ns | 6329ns |
| **32** | 330ns | 309ns | 382ns | 11370ns | 13329ns |

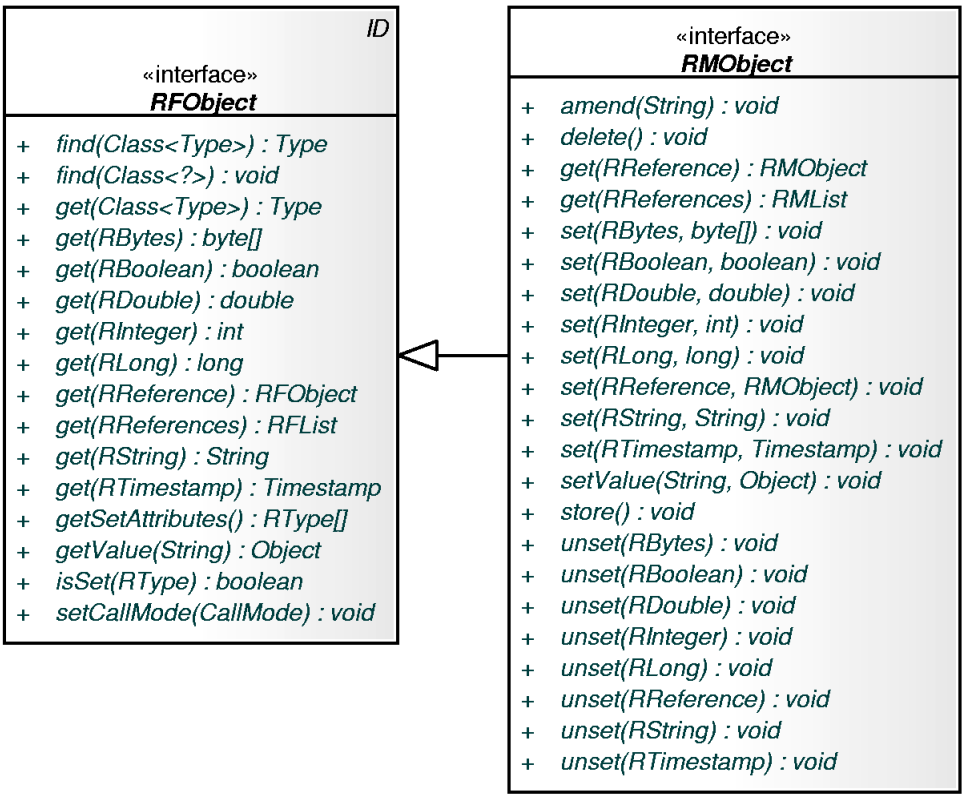
Obrázek 3.4: Čtení a modifikace instancí tříd s atributy v mapě

Na obrázku 3.4 se můžeme přesvědčit, že použití HashMapy pro naše účely je velmi nevýhodné. Pro malé počty atributů je řešení s polem o dost výkonější. Navíc objekt HashMapy po vytvoření zabírá 80B a za každou vloženou dvojici si vytvoří interní obalující objekt, který má dalších 24B.

Jelikož budeme chtít rozvolněné objekty ukládat do databáze, je potřeba si také pamatovat informace o tom, který atribut máme nově nastaven či změněn. Půjde pak snadno zjistit, jaké změny je potřeba zpropagovat do databáze.

Jednou z doporučovaných praktik objektově orientovaného programování je vytváření neměnitelných tříd. Neměnitelná třída je taková, jejíž instance nelze měnit. Všechny informace obsažené v dané isntanci jsou od okamžiku vytvoření pevně dány. Tyto třídy je jednodušší navrhnout a používají se snáze. Nám by takové třídy vyhovovaly z toho důvodu, že chceme kontrolovat, zda objekty splňují nějaké podmínky. Když objekt nemůže změnit nijak svůj stav, je potřeba podmínku otestovat jen jednou a pak už je splněna navždy. Další výhodou je jejich snadné kopírování. Jelikož jejich stav nelze změnit, je možné při kopírování přímo předat reference na všechny atributy objektu.

Teď už známe všechny pořadavky na rozvolněné objekty a možnosti, které pro rozvolněné objekty v Javě máme, pojďme se tedy podívat na navrhované řešení, které je znázorněno na obrázku 3.5.




Obrázek 3.5: Reprezentace rozvolněného objektu

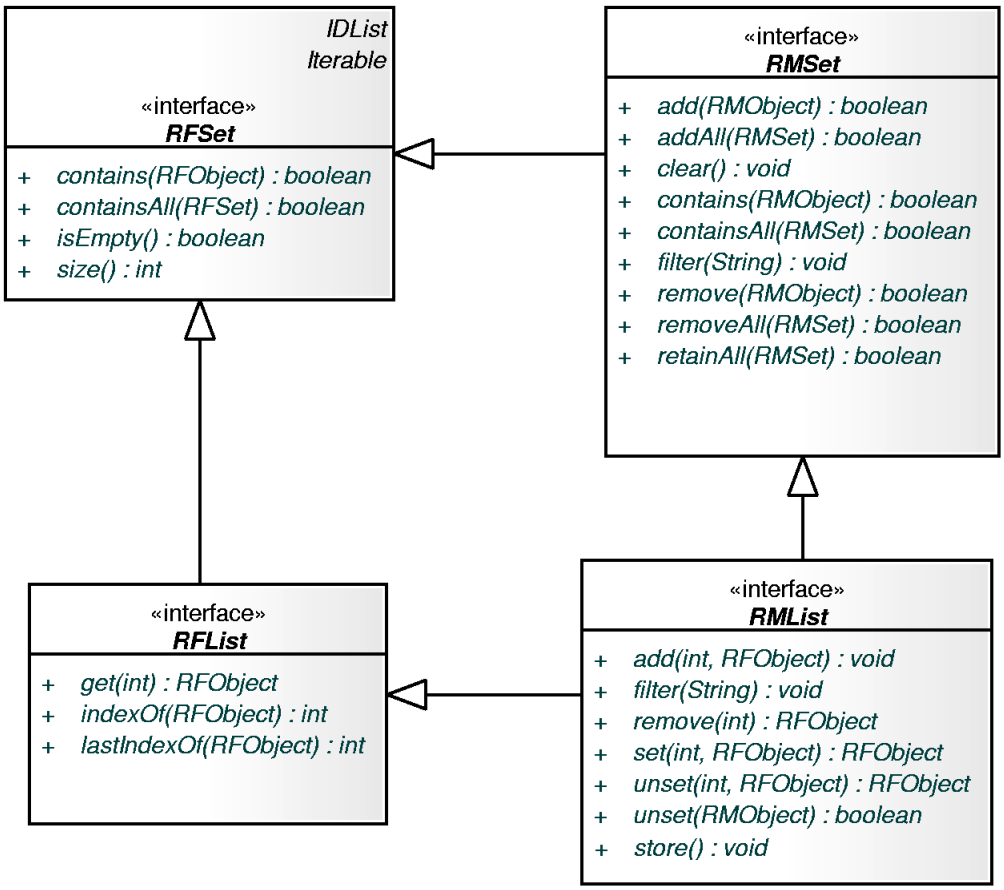
Jelikož uživatelé většinu času při používání aplikací stráví jen čtením různých informací a uživatelů, kteří data zapisují, je jen minimum, je rozumné vytvořit dvě reprezentace rozvolněných objektů. RFObject (zkratka za relaxed final object) reprezentuje neměnitelný objekt a RMObject (zkratka za relaxed mutable object) reprezentuje měnitelný objekt.

Použitím RFObjectu nepotřebujeme vůbec řešit změnu jeho stavu, takže všechny operace na něm mohou probíhat rychleji (hlavně díky tomu, že objekt přesně ví, jaké má nastavené atributy - měnitelné objekty si to musí vždy při každé operaci kontrolovat) a můžeme ušetřit paměť, kterou bychom jinak potřebovali na uložení stavu atributů. RFObject je zabezpečený z hlediska vláken - každé vlákno může pracovat jen s aktuálním stavem. Všechny implementace RMObjectu budou vláknově nezabezpečené a poskytneme wrapper, který překryje všechny metody předaného měnitelného objektu s použitím klíčového slova synchronized. Tím si uživatel může vybrat, jestli zvolí rychlou implementaci měnitelného objektu nebo potřebuje použít vláknově zabezpečenou, která bude k dispozici pro libovolnou měnitelnou implementaci. Jak jsme viděli dříve, na uchování atributů bylo nejlepší použít pole. Když se bude vytvářet konkrétní neměnitelný objekt, může se na základě jeho aktuálních atributů zvolit optimální implementace (například s polem typu int ve třídě navíc).

Obě rozhraní obsahují přetížené metody get,set a unset. Zajímavé na nich je to, že typy jejich parametrů jsou definované speciálně pro rozvolněné objekty. Dosáhneme tím řádově větší kontroly kompilátoru než je v běžných řešeních, které předávají název atributu jako ničím nekontrolovaný řetězec (bude vysvětleno v části 3.3). Dalším zajímavým faktem jsou návratové hodnoty funkcí get - jsou jimi totiž ve všech možných případech primitivní datové typy. Nebudeme tím vyžadovat po jednotlivých implementacích, aby používaly wrappery primitivních datových typů a možná tím ušetříme paměť nebo zrychlíme přístup k atributům.

Metoda amend donahraje objektu požadované atributy z databáze. Atributy, které už objekt obsahuje, zůstanou nezměněny. Metoda delete odstraní objekt z databázi. Metoda store uloží objekt včetně všech odkazovaných objektů do databáze. Vždy když se volá store, tak se postupně v dávkách ukládají jednotlivé atributy. Metoda unset označí atribut za odstraněný a při dalším volání store ho odstraní z databáze.

## 3.2 Množiny a seznamy objektů




Obrázek 3.6: Reprezentace množiny rozvolněných objektů

Množiny a seznamy jsou stejně jako rozvolněné objekty rozdělené na neměnné a modifikovatelné. Rozdíl mezi metodou unset a remove je ten, že remove odstraní objekt ze seznamu, kdežto unset jen označí objekt za odstraněný a při dalším volání store odstraní odkaz na něj z databázi (objekt ale zůstává vždy zachován, odstrňuje se jen odkaz). Metoda filter na rozhraní množiny nebo seznamu zanechá jen ty objekty, které podmínku předanou jako parametr splňují.

Nejzajímavější je ale realizace volání metod. Pomocí volání metody find explicitně zažádáme, aby se pro aktuální stav objektu našla správná implementace. Tato je navíc uložena do cache objektu. Jakého typu je předaný parametr, takový typ dostaneme v návratové hodnotě. Pomocí metody get se nejdříve podíváme, zda není implementace z předchozích volání známá, a když není, tak se ta správná nalezne. Každopádně po zavolání metod find nebo

get dostaneme implementaci metody, kterou můžeme standardním způsobem zavolat. V některých případech realizace metod lze jako parametr funkcí get a find předat nejen rozhraní, ale i přímo třídu a je nám vrácen nejspecifičtější potomek.

Metoda setCallMode nastaví způsob následujícího volání metody. Můžeme si vybrat, jestli se má volat včetně vyhodnocení podmínek, nebo jen vyhodnotit podmínky nebo jen volat metodu. V případě zavolání ověření podmínky je potřeba pak zavolat nějakou další metodu, která vrátí informaci, zda předchozí ověřování podmínek proběhlo v pořádku.

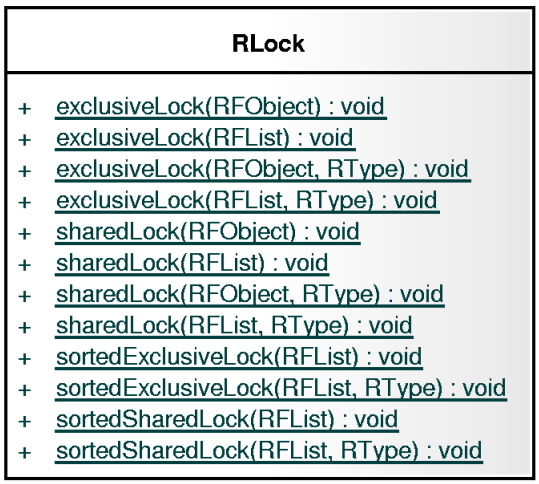
## 3.3 Persistence objektů

```
public enum String implements RString {
  @Info("State description")
  StateDescription
}
```

Obrázek 3.7: Definice atributu

Na obrázku 3.7 je ukázka definice atributu. Je definován jako prvek výčtového typu a jeho datový typ je mu určen rozhraním, které implementuje. Tento způsob definice atributů je velmi bezpečný, jelikož pak při přístupu k atributům objektu je tento typ kontrolován kompilátorem. Pomocí anotace musíme zapsat popis tohoto atibutu. Můžeme zadat další parametry jako je požadavek na vytvoření indexu nebo požadavek na jednoznačnost hodnoty atributu v databázi. Dále lze nastavit jednotku atributu. Jediný způsob, jak vytvořit atribut bude tato definice. Potom bude možné zavolat metodu, která projde classpath (kterou ale půjde omezit, kde všude se má prohledávat), a pro nově vytvořené atributy vytvoři tabulku v databázi a zaregistruje se v systémovém katalogu. Když neuvedeme u atributu anotaci Info, musíme uvést anotaci Transient, která reprezentuje tranzientní atribut, tedy jeho hodnota se neukládá do databáze. Když neuvedeme žádnou anotaci u definice atributu, tak je metodou vyhozena vyjímka.

Bude podporováno několik druhů transakcí. Při volání začátku transakce je pak možné zvolit, jak se má systém chovat v případě, že už jedna transakce existuje. Můžeme si vybrat mezi vyhozením vyjímky, převzetím existující transakce, nebo vytvořením nové transakce. Posledním módem je vytváření nové transakce jen v případě, že je to potřeba. Při volání začátku transakce je možné zvolit úroveň izolace transakce a zda má být transakce read only. Většina databázových systému toto neumožňují měnit v průběhu transakce. Tento poslední mód by umožňoval se v tomto případě rozhodnout pro konkrétní databázi, zda je potřeba vytvořit novou transakci.




Obrázek 3.8: Podpora zamykání

Na obrázku 3.8 je vidět rozhraní pro zamykání objektů. Bude možné zamykat sdíleně nebo exklusivě jednotlivé atributy, nebo celé objekty. V případě zamykání více objektů najednou si můžeme vybrat, zda se objekty musí zamykat v pořadí podle svých id.

## 3.4 Metody s podmínkami

První otázkou je, jakým způsobem by bylo vhodné zapisovat omezení metod na parametry a objekty, na kterých mohou být spouštěny. Máme následující možnosti.

- komentáře v kódu - jejich hlavní nevýhoda je, že při kompilaci se ztratí
- volání speciálních metod, které podmínky zkontrolují - výhodou je, že jsou kontrolovány kompilátorem, ale zásadní nedostatek tohoto řešení je nutnost psát a volat ověřovací metody
- ve speciálním souboru - nevýhodou je oddělenost definice podmínky od metody. Výhoda je naopak jednoduché vynucení kontroly takových podmínek, protože například projekt AspectJ[2] dokáže zařídit, aby definovaná metoda byla zavolána před nebo po zavolání určených metod
- anotace - tato speciální konstrukce jazyka Java představuje kompilátorem kontrolované komentáře. Lze přesně určit, kde všude v kódu lze tyto komentáře psát

Druhým problémem je vybrání způsobu jak zaručit, aby se zapsané podmínky kontrolovaly za běhu aplikace. Opět máme na výběr z několika možností.

- preprocesor - převede zdrojový kód na požadovaný upravený zdrojový kód. Nevýhodou je, že je potřeba vždy udělat tento mezikrok
- vlastní kompilátor - převede zdrojový kód nebo bytecode na požadovaný bytecode. Problém je závislost na tomto kompilátoru

[2] http://www.eclipse.org/aspectj/

- instrumentace bytecodu - Java umožňuje při nahrávání tříd do paměti provést úpravu této právě načítané třídy

- aspectj - používá instrumentaci bytecodu pro úpravu tříd, je výrazným zjednodušením použití vlastní instrumentace bytecodu, ale umožňuje pouze zavolat metodu při splnění určité podmínky (například, že je metoda označena nějakou anotací), ale neumožní už dané třídě bytecode dál modifikovat (vlastní instrumentace bytecodu by mohla načíst podmínky a přímo do tříd, kde by se měli podmínky kontrolovat, by mohla volání konkrétních podmínek přidat)

Posledním problémem je vybrání způsobu jak objektům zpřístupnit volání metod, které jsou definované ve více třídách (aby objekty měli možnost zavolat libovolnou metodu, pro kterou spňují podmínku). Opět se nabízí několik možností.

- reflexe - je možné programově volat libovolnou metodu, takže můžeme poskytnout rozhraní, které bude umožňovat zavolat libovolnou metodu. Problém je, že parametry by šly předávat jen typu Object a tudíž by nebyla prováděna dostatečná typová kontrola. Navíc použití reflexe výrazně zpomaluje volání metod

- proxy - Java umožňuje za běhu vygenerovat implementaci pro sadu zadaných rozhraní. Omezením je, že parametry musí být rozhraní a třídy se nemůžou používat, To by nutně znamenalo pro každou metodu definovat i rozhraní. Vygenerování implementace za běhu aplikace je také časově velmi náročné

- vlastní řešení - mít speciální metodu, která by požadované implementace hledala. Hlavní výhoda tohoto řešení je možnost hledat potomky třídy, která by se předala jako parametr

Podmínky tříd se dědí a to včetně podmínek na parametry metod. V navrhovaném modelu se vždy rozlišují dvě kategorie podmínek.

- požadavky na rozhraní nebo třídě - specifikují požadavky na všechny metody rozhraní nebo třídy. Tyto podmínky mají formát specifikovaný v části 2.4.2

- požadavky přímo na parametry metod - reprezentují požadavky na aktuálně předaná data. Mohou mít tvar jakéhokoliv validního Javovského příkazu (například volání libovolné funkce)

První řešení pro typově bezpečné objekty je generování objektů implementujících více rozhraní pomocí java proxy objektů. Na obrázku 3.9 je standardní definice metody, kde podmínky na metodě se mohou definovat jak na rozhrani, tak pak speciálnější podmínky na implementaci, které budou rozšiřovat podmínky rozhrani. Na obrázku 3.10 je ukázka implementace proxy objektu, který může reagovat na volané metody objektů a kontrolovat tak podmínky metod a hledat nejvhodnější implementace rozhraní podle současných parametrů objektu. Na obrázku 3.11 je vidět použití proxy objektů. Jelikož Java kontroluje přetypování objektů za běhu, tak pakliže chce uživatel přetypovávat objekt na jiný inteface než zadal při tvorbě objektu, tak je vyhozena vyjímka. Jediné omezení je, že by se rozvolněné objekty musely vytvářet přes tovární metodu, tj nemohly by se vytvářet přes konstruktor, ale to nám v tomhle případě vůbec nevadí, jelikož budou existovat jen dvě třídy pro rozvolněný objekt.

```
public interface Foo {
    Object bar(Object obj) throws BazException;
}

public class FooImpl implements Foo {
    Object bar(Object obj) throws BazException {
        // ...
    }
}
```

Obrázek 3.9: Definice metody

```
public class DebugProxy implements java.lang.reflect.InvocationHandler {

    private Object obj;

    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new DebugProxy(obj));
    }

    private DebugProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        Object result;
        try {
            System.out.println("before method " + m.getName());
            result = m.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: " +
                                       e.getMessage());
        } finally {
            System.out.println("after method " + m.getName());
        }
        return result;
    }
}
```

Obrázek 3.10: Implementace proxy objektu

```
Foo foo = (Foo) DebugProxy.newInstance(new FooImpl());
    foo.bar(null);
```

Obrázek 3.11: Použití proxy objektu

Druhým mnohem složitějším řešením je použití instumentace bytecodu a volání metod find a get, tak jak byli popsány v části 3.2.

V definici metod by vyžadovala buď implementaci metody pro předání rozvolněného objektu, nebo v případě řešení přidání ověřování podmínek metodám pomocí instrumentae bytecodu by bylo možné dokonce metody rozpoznávat podle toho, zda obsahují jako svůj člen

rozvolněný objekt. Každá metoda je tedy povinná zvnějšku přijmout stav nad kterým může být zavolána.

## 3.5 Dotazy

Optimální řešení je odstínit vývojáře od relační databáze a poskytnout mu jen objektově orientované dotazy. Proto dotazy budou řešené pomocí metod, které vrací buď RFList (v případě, že už objekty nebudeme chtít měnit) nebo RMList. Tyto metody mají parametr podmínku, která by se mohla skládat z dalších podmínek.

## 3.6 Systémový katalog

```
public class SystemCatalog {
  public enum String implements RString {
      @Info(value="Name of attribute",unique=true)
    RAttribute,
      @Info("Type of attribute")
    RType,
      @Info("Unit of attribute (e.g. ms,s,CZK,USD)")
    RUnit,
      @Info(value="Description of attribute",unique=true)
    RDescription
  }
  public enum Integer implements RInteger {
      @Info(value="Index of attribute for RSetAttributes",unique=true)
    RSetIndex
  }
  public enum Boolean implements RBoolean {
      @Info("Information if values of attribute have to be unique")
    RUnique,
    @Info(value="Information if index is created for attribute")
    RIndex
  }
  public enum Bytes implements RBytes {
      @Info("Bit array of set attributes of relaxed object")
    RSetAttributes,
  }
}
```

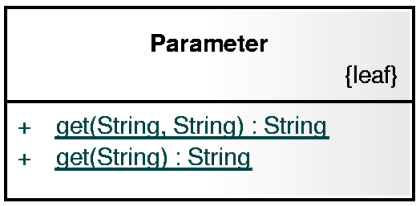
Obrázek 3.12: Definice systémového katalogu

Na obrázku 3.12 si můžeme prohlédnout definici systémového katalogu. Zajímavé je, že je úplně přesně shodná s definicí ostatních atributů. Díky povinosti komentovat definici atributů si můžeme popis jednotlivých atributů přečíst přímo v jeho definici.

Jediné atributy, které je potřeba popsat jsou RSetIndex a RSetAttributes. Hodnotou RSetAttributes je bitové pole, které eviduje nastavené atributy pro každý objekt. Jelikož Oracle nepodporuje žádné bitové typy, tak je potřeba reprezentovat toto pole jako pole bytu. Zajímavé na tom je, že toto pole může mít menší velikost než je počet existujících atributů. Neexistující index znamená, že atribut není nastaven. Je možné za běhu aplikace přiřadit jednotlivým

atributům jiné indexy v tomto poli, aby se tak dosáhlo minimálního součtu všech délek bitových polí.

## 3.7 Konfigurace




Obrázek 3.13: Podpora pro načítání parametrů

Byla vytvořena speciální podpora pro načítání parametrů. Nejprve se načte jeden výchozí soubor, v němž je parametr cmn.core.parameter.config.paths, který definuje posloupnost cest, ze kterých se přečtou parametry z properties souborů. Pointa je v tom, že v různých cestách můžou být definovány stejné parametry a jde tak změnou jediného parametru například změnit databázi z oracle na postgres nebo přidat speciální konfiguraci pro testy. Při načítání se do názvu parametru ještě přidá celá adresářová struktura souboru, ve kterém je parametr definován, aby se minimalizovala šance, že víc částí aplikace bude používat stejný název parametru.

Pro vyšší bezpečnost je možné parametry jen číst a to pomocí dvou metod, kde jedna umožňuje získat hodnotu parametru a druhá si může definovat defaultní hodnotu. V případě, že první metoda přistupuje k neexistujícímu parametru, je vyhozena vyjímka. Druhá metoda v tomto případě zaloguje varování, že se přistupuje k neexistujícímu parametru.

## 3.8 Logování

Ke konfiguraci je potřeba použít parametr cmn.core.logger.config.file, kde se nastaví cesta ke standardnímu properties souboru log4j. Ve frameworku se používají tyto úrovně logování:

- info - logování běžných událostí
- severe - došlo ke kritické chybě
- warning - něco není v pořádku, ale aplikace bude pravděpodobně fungovat správně
- fine - určené pro logování ladících informací

## 3.9 Jmenné konvence

Všechny důležité rozhraní a třídy byli pojmenovány s počátečním písmenem R, které znamená zkratku za relaxed. Umožňuje to rychlé vyhledávání požadovaných objektů. Vývojové prostředí totiž po napsání prefixu názvu automaticky doplní chybějící písmena, nebo v případě nejednoznačného prefixu dá uživateli vybrat, co požaduje doplnit.

## 3.10 Použité knihovny

- Asm (asm-2.0.RC1.jar) - umožňuje instrumentaci bytecodu, tedy je možné načítané třídy modifikovat
- JEXL (commons-jexl-1.1.jar) - umožňuje zadat knihovně řetězec s přikazem v Javě a pak ho vyhodnotit
- JFreeChart (jfreechart-1.0.0.jar) - umožňuje generovat grafy
- JUnit (junit-4.5.jar) - podpora pro unit testy
- TestNG (testng-jdk.jar) - podpora pro unit testy, jde o rozšíření JUnit
- Postgres (postgresql.jar) - umožňuje připojení do postres databáze
- OJDBC (ojdbc14.jar) - umožňuje připojení do databáze oracle
- Cloning (cloning-1.4.jar) - umožňuje udělat hlubokou kopii jakéhokoliv objektu

# Kapitola 4

# Ukázková aplikace: Úložiště výsledků testů

V této kapitole bude navržena aplikace Úložiště výsledků testů standardním způsobem a poté s pomocí rozvolněných objektů. Dále v této kapitole bude provedeno porovnání tvorby a výkonu obou verzí aplikace. Autor této diplomové práce měl možnost pracovat několik let jako vývojář nástrojů pro automatizované testy ve společnosti Unicorn a tato aplikace by výrazně pomohla k efektivnějšímu rozšiřování vyvíjených nástrojů určených k automatizovanému testování. Všechny požadavky na tuto aplikaci vznikly z reálných potřeb vzniklých při zavádění testovacího procesu na projektu Unicorn Universe ve společnosti Unicorn.

## 4.1 Motivace

Vývoj software je sada aktivit, jejichž výsledkem je softwarový produkt. Za úspěšný lze považovat takový vývoj software, který vytvoří softwarový produkt s požadovanou funkčností v dostatečné kvalitě za domluvenou cenu a ve stanovený termín[1]. Uvedená kritéria úspěchu spolu velmi úzce souvisí, neboť jsou-li například kladeny vyšší nároky na kvalitu nebo funkčnost, vzrostou automaticky náklady nebo čas potřebný k vývoji softwarového produktu. Mezi významné faktory, které ovlivňují úspěšnost vývoje, patří zkušenost vývojového týmu, vhodnost zvolené metodiky vývoje a užitečnost použitých nástrojů.

Jedním z největších rizik vývoje software je existence chyb, které nutí provádět nenaplánované zásahy do již existujících částí softwarového produktu. Složitost současně vytvářených softwarových produktů bohužel v podstatě zaručuje existenci těchto chyb nezávisle na tom, jak dobrý vývojový tým i proces vývoje jsou. Tuto skutečnost si většina firem začíná uvědomovat a proces testování, který se právě hledáním chyb zabývá, se stává neoddělitelnou součástí vývoje software. Testování ovlivňuje všechna výše zmíněná kritéria úspěšného vývoje software. Náklady na odstranění chyby i čas potřebný k jejímu odstranění lze výrazně snížit jejím včasným nalezením. Testování slouží k ověření, že softwarový produkt obsahuje všechnu požadovanou funkčnost a je nejdůležitější aktivitou, kterou lze zaručit kvalitu výsledného softwarového produktu. Testování objevuje chyby, které by se mohly projevit při práci s nasazeným softwarovým produktem. V případě, kdy stačí chybu opravit, se zvýší náklady na vývoj softwarového produktu. V nejhorším případě ale mohou chyby v nasazeném softwarovém produktu znamenat i ztrátu zákazníka.

Náklady na testování a odstraňování chyb představují nezanedbatelnou část celkových nákladů celého vývoje software. Proto se může v praxi použít přístup, kdy si zákazník sám

[1] Podle této metriky se hodnotí úspěšnost projektů ve firmě Unicorn

přesně určí, co musí bezpodmínečně nutně fungovat a jaké chyby je možné tolerovat. U každé nalezené chyby se lze rozhodnout, zda je potřeba jí do výsledného softwarového produktu opravit, vydat opravu později nebo dokonce chybu úplně ignorovat. Oprava chyby může ohrozit termín vydání softwarového produktu. Na druhou stranu chyba nemusí být pro zákazníka natolik závažná, že možnost používat dříve nové funkčnosti softwarového produktu může být pro zákazníka mnohem výhodnější než trvat na okamžité opravě. V případě, že zákazník na opravě chyby netrvá, je možné ušetřit náklady a čas. Z tohoto je patrné, že i s chybami je potřeba umět efektivně pracovat.

Hlavním úkolem testování je ověření, že softwarový produkt splňuje všechny požadavky na něj kladené. Složitost dnešních softwarových produktů v podstatě znemožňuje nesystematický přístup, jelikož otestování jedné verze softwarového produktu by pro jednoho člověka mohlo znamenat několik týdnů práce. Díky tomu je způsob testování, kdy by všichni testeří používaly softwarový produkt stejným způsobem a snažili se najít co nejvíce chyb, v současné době nedostačující. Testování se musí rozdělit mezi více testerů a každý z nich má za úkol otestovat podle předem daných scénářů pouze část softwarového produktu. Výsledky testů představují důležitý zdroj informací pro rozhodování o dalším vývoji softwarového produktu a proto jsou požadavky na otestování aktuální verze softwarového produktu čím dál častější a je tedy potřeba proces testování provádět co nejefektivněji. Efektivitu testování lze výrazně zvýšit používáním automatizovaných testů, které dokáží ověřit řadu vlastností softwarového produktu bez přítomnosti lidského faktoru. Automatizované testy nemohou bohužel manuální testy kompletně nahradit, ale pouze je doplňují. Existují totiž situace, kdy buď nelze automaticky ověřit požadovanou vlastnost softwarového produktu nebo by toto ověření bylo finančně nebo časově řádově náročnější než v případě manuálních testů. Příkladem takovéto vlastnosti může být vzhled softwarového produktu. Naopak automatizované testy dokáží otestovat vlastnosti softwarového produktu, které manuálními testy nasimulovat nejdou. Jedná se převážně o simulaci práce více uživatelů s přesným časováním jednotlivých aktivit. Pro nejefektivnější otestování softwarového produktu je tedy potřeba používat automatizované testy společně s manuálními testy.

Snahou testování je nalézt co největší množství chyb v co nejkratším čase. Existuje několik základních kategorií testů, které se specializují na konkrétní druhy chyb. K nejpoužívanější kategorii testů patří unit testy, které testují funkčnost jednotlivých metod a tříd. Jedná se o vývojářské testy, kde autor testu přesně ví, jak metoda a třída fungují, a kde se ověřuje, zda daná metoda a třída dělá to, co se od nich očekává. Dalším příkladem kategorie testů jsou funkční testy. Zde se jedná o uživatelské testy, kde autor testu neví nic o vnitřní struktuře softwarového produktu a snaží se ověřit, zda softwarový produkt dělá to, co potřebuje uživatel. V tomto případě test přistupuje k softwarovému produktu přes stejné rozhraní jako uživatel. Tyto dvě kategorie testů ověřují funkčnost softwarového produktu. Existuje mnoho dalších kategorií testů, které ověřují kvalitu zdrojového kódu, dobu odezvy pod očekávanou zátěží, zabezpečení, rozhraní softwarového produktu, dokumentaci a spoustu dalších vlastností softwarového produktu. Pro ověření těchto vlastností většinou existují speciální nástroje nebo lze jejich ověřování provádět manuálně.

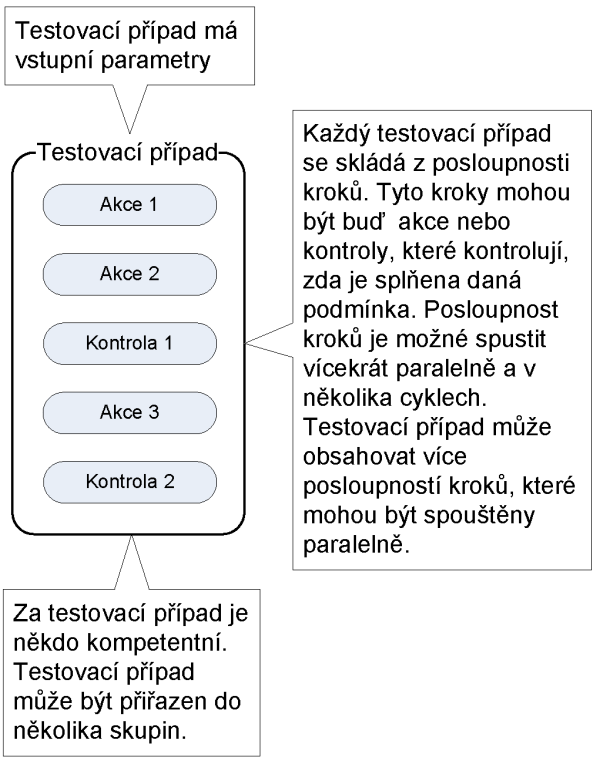
Při zavádění procesu testování se může objevit velké množství problémů. Většina dostupných nástrojů určených pro testování používá svou vlastní terminologii pro danou kategorii testů. Vlastní struktura testů a měřené statistiky jsou v těchto nástrojích různé. Formát reportů s výsledky testů je různý a obsahuje výsledky jen jednoho spuštění testů. Nelze tedy porovnávat více spuštění testů mezi sebou, což je v praxi jeden z nejčastějších požadavků. Reporty jsou typicky speciální pro daný nástroj a ve většině případů nejsou snadno rozšiřitelné. V případě potřeby přidání nových měřených statistik nebo potřeby přidání nové metriky do

reportu je potřeba tyto úpravy dělat v každém nástroji zvlášť a většinou v každém nástroji jiným způsobem. Navíc nutnost analyzovat výsledky vždy v jiném formátu reportu může být celkem matoucí a vést i k nespravné interpretaci výsledků nebo nenalezení potřebných informací, které mohou být v daném reportu obsažené na jiném místě, než kde jsou hledány.

Hlavním cílem je vytvoření úložiště výsledků testů, které bude společné pro všechny kategorie testů. Dále bude navržena podpora pro generování reportů s možností porovnávat mezi sebou jednotlivé spuštění testů. Při integraci nového testovacího nástroje bude tedy potřeba jen rozšířit nástroj tak, aby byl schopen ukládat do úložiště informace o struktuře testu a o výsledcích jednotlivých spuštění testů. Potřebná práce s tvorbou a rozšiřováním reportů pak bude probíhat stejným způsobem a ve stejných technologiích. Navíc ve většině případů reporty budou sdílet už jednou použité části reportů, takže práce s reporty by měla být minimální. Součástí návrhu úložiště bude i návrh společné terminologie a společné struktury testů s jejich výsledky pro všechny kategorie testů, což by mělo zpřehlednit čitelnost reportů a zjednodušit práci s jejich tvorbou a rozšiřováním. Každé spuštění testů bude mít možnost ukládat jiné statistiky. Například v jednom spuštění testů se přesně monitoruje kompenenta, ale monitorování natolik zpomaluje softwarový produkt, že po otestování komponenty se opět měří jen původní statistiky. Dalším příkladem je přidání podpory pro nové statistiky do testovacího nástroje, takže staré spuštění testů obsahují jiné statistiky než nové spuštění testů. Hlavní výhodou úložiště výsledků testů je jednoduchá tvorba a rozšiřitelnost reportů s výsledky testů, kde každé spuštění testů může obsahovat různé statistiky. Navíc v reportech je možné porovnávat mezi sebou výsledky testů z více spuštění. Model rozvolněných objektů je právě optimální pro práci s takto různorodými daty. Přidáním dat týkajících se výsledků testů je automaticky možné generovat všechny předpřipravené reporty.

## 4.2 Obecná struktura výsledků testů

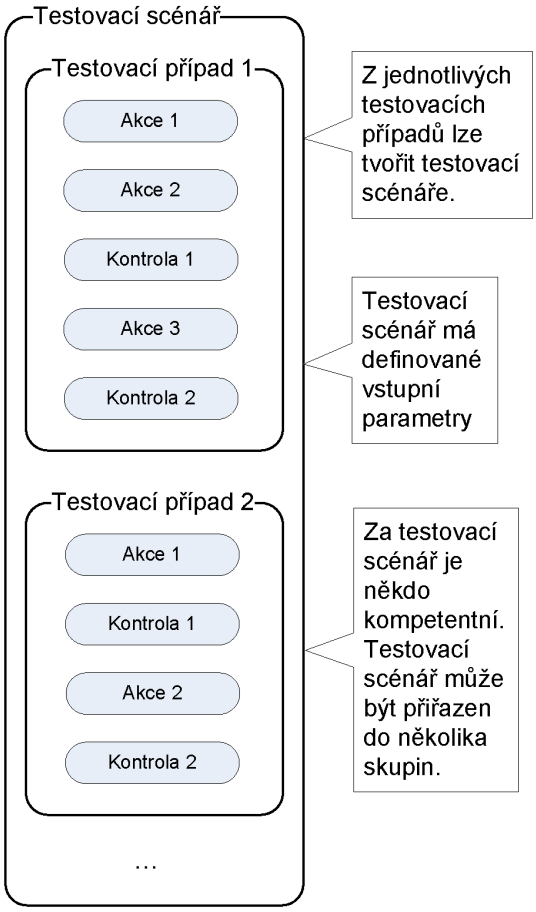
Nejdříve budeme muset vymyslet takový úhel pohledu na testy a jejich výsledky, abychom mohli veškeré informace, které se testů i jejich výsledků týkají, uložit do společného datového modelu. Pak jednotlivé části reportu půjde použít ve všech spuštěních testů, která budou obsahovat požadovaná data.




Obrázek 4.1: Testovací případ

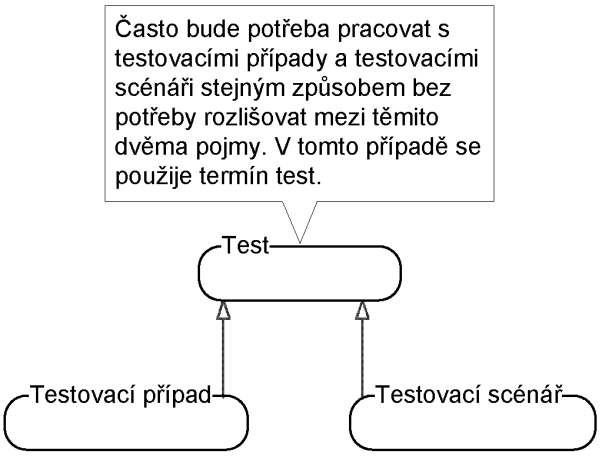
Základním objektem používaným při testování je testovací případ. Pro většinu testovacích frameworků, které podporují funkční testování, je výsledek testovacího případu již dále nedělitelný. Jednoduše nás zajímá, zda požadovaná funkcionalita je v pořádku. Příkladem takového frameworku v Javě je JUnit[2] nebo TestNG[3]. V některých případech ale může být potřeba znát detailnější strukturu testovacího případu. Každý testovací případ se vždy skládá z několika akcí (například volání metody nebo poslání požadavku na http server), pak se většinou ještě explicitně kontroluje, zda je splněna nějaká podmínka (typicky test, zda daná metoda vrací očekávaný výsledek). V některých případech jen zavoláme metodu a v případě, že zavolání metody nezpůsobí žádnou chybu, považujeme testovací případ za úspěšný. Takto vypadá většina funkčních testů. Zátěžové testy vypadají podobně, jen se tato posloupnost akcí a kontrol pouští opakovaně v několika paralelně běžících vláknech. Občas požadujeme, aby v rámci jednoho testovacího případu běželo více takových posloupností akcí a kontrol paralelně (například ve druhém vláknu měříme co dělá první vlákno, případně častým požadavkem je, že jedno vlákno čte data z aplikace a další je zapisuje, aby se simuloval přesný poměr uživatelů, kteří v aplikaci jen čtou a uživatelů, kteří i zapisují). Testy můžou mít vstupní parametry (zavolání metody pro různé vstupy nebo přihláěení do systému pod konkrétním uživatelem). Za každý testovací případ je někdo kompetentní, aby v případě, že testovací případ nefunguje, mohla dotyčná osoba být automaticky informována a požádána o nápravu. Skupiny testovacích případů jsou potřeba například pro označení nedokončených testů, smoke testů, testů testujících základní toky a podobně.

[2]http://www.junit.org/
[3]http://testng.org



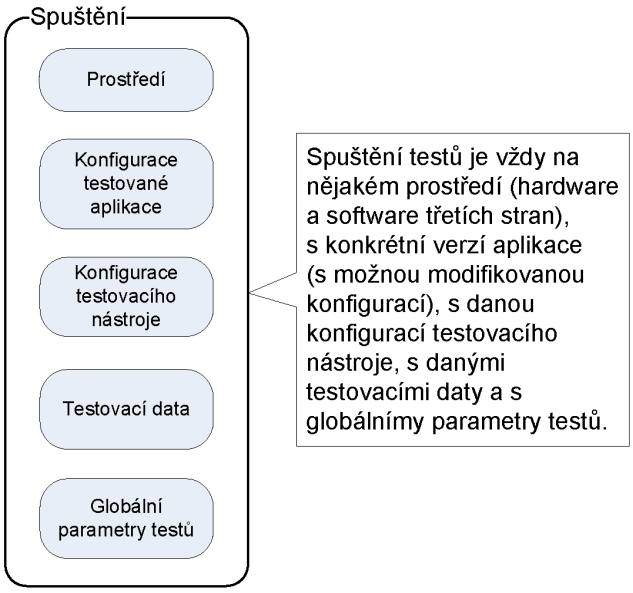

Obrázek 4.2: Testovací scénář

Z testovacích případů lze skládat testovací scénáře. Můžeme tak použít opakovaně definici testovacího případu. Například testovací případ může být přihlášení do aplikace, odhlášení z ní a vyhledání nějaké informace. Z těchto testovacích připadů lze sestavit testovací scénář, který můžeme spouštět pro různé počty paralelně pracujících uživatelů a v různých počtech cyklů, tedy také testovací scénáře mají své vlastní parametry stejně jako testovací případy. Také přiřazení kompetencí a skupin je stejné jako u testovacího případu.



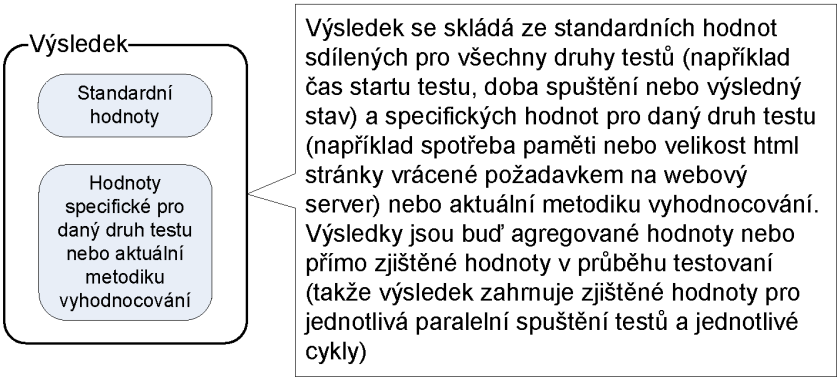

Obrázek 4.3: Test

Zobecněním testovacího případu a testovacího scénáře je pojem test. Testem budeme rozumět nějakou testovací aktivitu, která může mít nějaký výsledek.



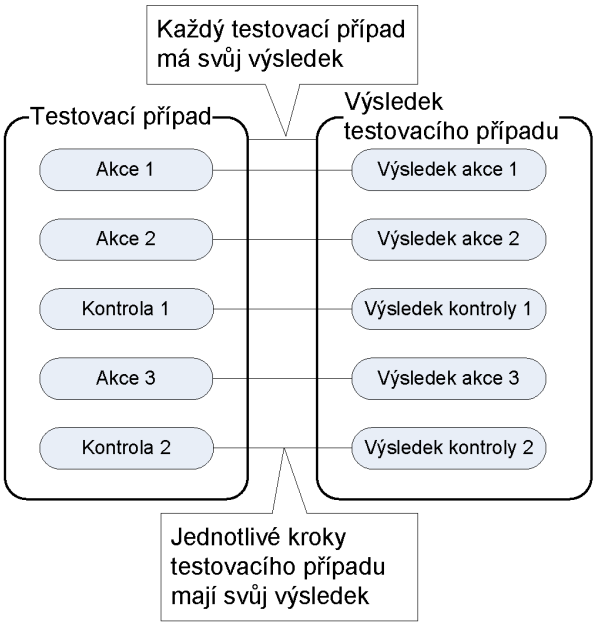

Obrázek 4.4: Spuštění

Výsledky testů se vždy týkají konkrétního spuštění. Je potřeba (například kvůli opakovanému spuštění testu) ukládat pro každé spuštění informace týkající se použitého software a hardware. Mezi takové informace patří například velikost paměti serveru, verze Javy, použitá sada testovacích dat nebo třeba speciální konfigurace aplikace.

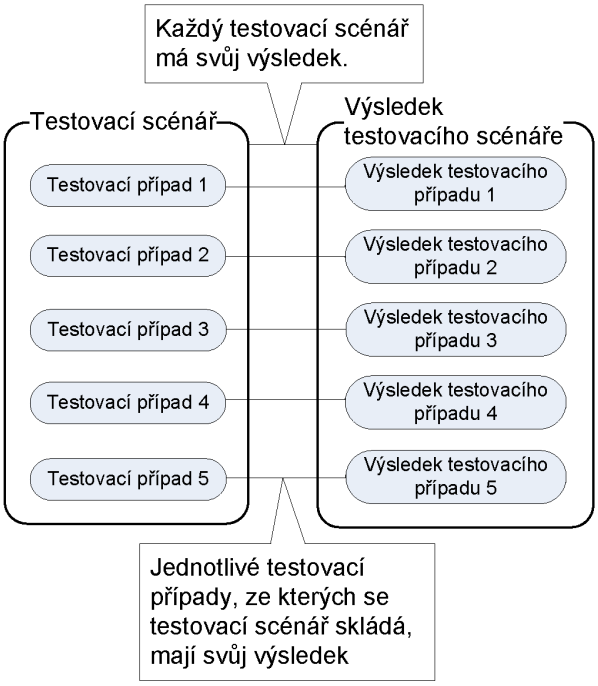



Obrázek 4.5: Výsledek

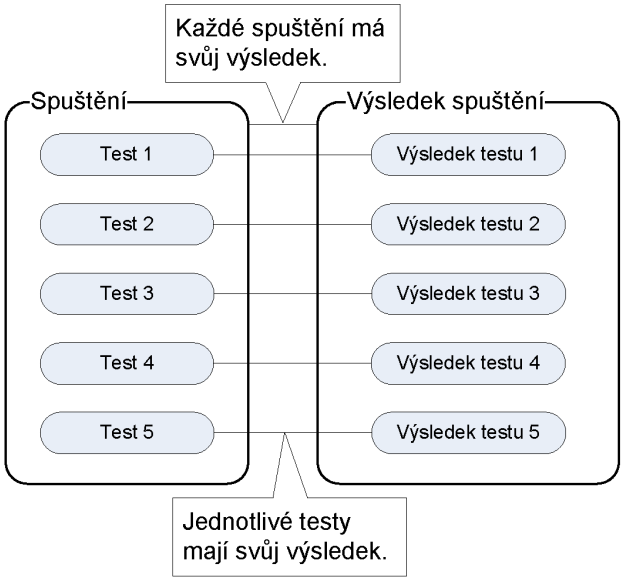
Co nás bude nejvíce na testech zajímat, jsou výsledky jejich spuštění. Většinou se jedná o informaci, zda daná funkcionalita funguje, případně jak rychle. Jelikož je ale možné v průběhu běhu testu získávat různé další informace týkající se testu (například spotřeba paměti aplikačního serveru pod zátěží nebo velikost přenosu dat po síti), je potřeba ukládat i tyto informace. Proto budeme rozlišovat hodnoty výsledků na standardní, které se vyskytují ve většině výsledků testů a specifické, které je buď potřeba ukládat jen občas nebo každý druh testu má tyto informace rozdílné. Na obrázcích 4.6, 4.7 a 4.8 můžeme vidět, že výsledky je potřeba ukládat pro testovací případy, testovací scénáře i spuštění testu. Tím je dán kontext, ke kterému uložená hodnota patří. Například zatížení procesoru aplikačního serveru je výsledek spuštění, ale doba odezvy jednotlivého http požadavku je výsledek testovacího případu.




Obrázek 4.6: Výsledek testovacího případu



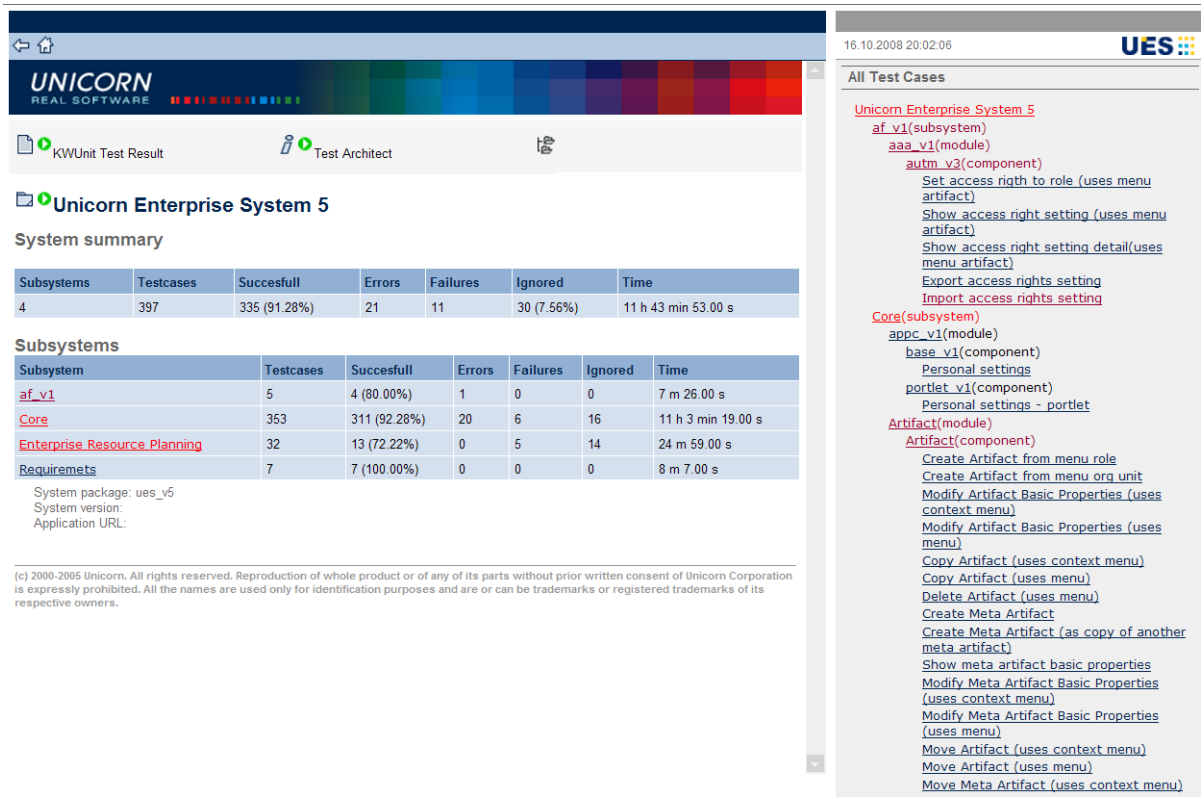

Obrázek 4.7: Výsledek testovacího scénáře




Obrázek 4.8: Výsledek spuštění

## 4.3 Požadavky na reporty

Podívejme se na ukázkové reporty, které bude potřeba generovat. Názvosloví a některé reporty byly se svolením firmy Unicorn převzaty z projektu Unicorn Universe.




Obrázek 4.9: Přehledový report funkčních webových testů

V pravé části obrázku 4.9 je seznam testovacích případů obsažených v daném spuštění testů. Jsou to odkazy na detaily jednotlivých testovacích případů. Vlevo jsou zobrazeny informace o struktuře projektů, které byly testovány. Vidíme zde celkové úspěšnosti jednotlivých subsystémů a můžeme si zobrazit detail libovolného z nich. Dále jsou zde počty úspěšných testů, počty nespuštěných testů a počty spuštěných testů, které skončily s chybou. Failure znaméná, že test dělal nějakou cílenou kontrolu podmínky a ta není splněna, a error znamená, že v průběhu testu nastala neočekávaná chyba.

**Show access right setting (uses menu artifact)**

**Testcase summary**

| Test Scenarios | Succesfull | Errors | Failures | Time |
|---|---|---|---|---|
| 5 | 5 (100.00%) | 0 | 0 | 50.58 s |

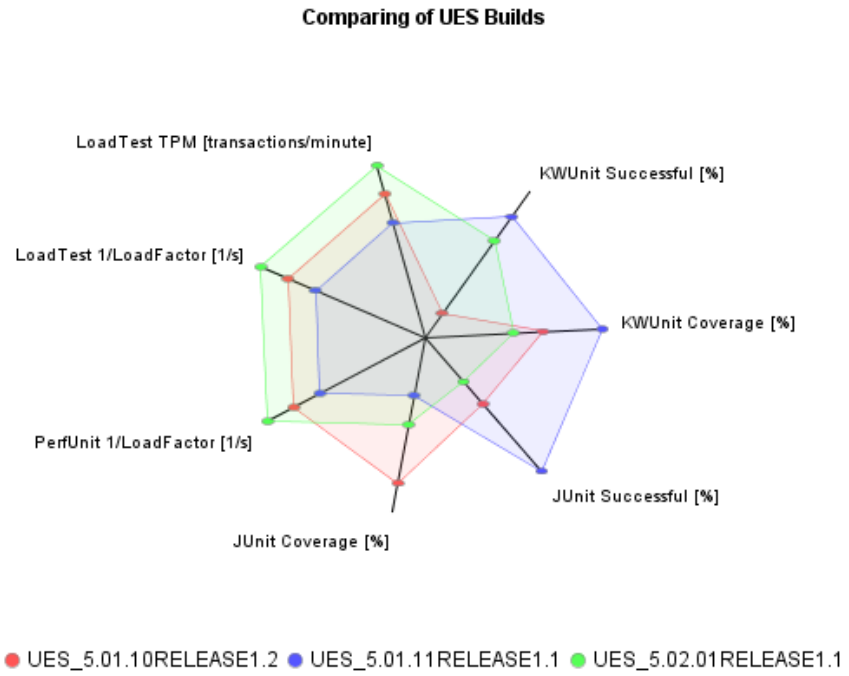
Class: cz.unicorn.ues_v5.af_v1.aaa_v1.autm_v3.kwunit.C10H102MATest
Competency: UES.CDP.SA
Start at: 16.10.2008 17:20:16

**Scenarios**

| TestScenario | Status | Time |
|---|---|---|
| Login | Succesfull | 5.53 s |
| testC101000MU | Succesfull | 18.86 s |
| testC10H101MA | Succesfull | 17.97 s |
| testC10H102MA | Succesfull | 8.23 s |
| Fake Logout | Succesfull | 0.00 s |

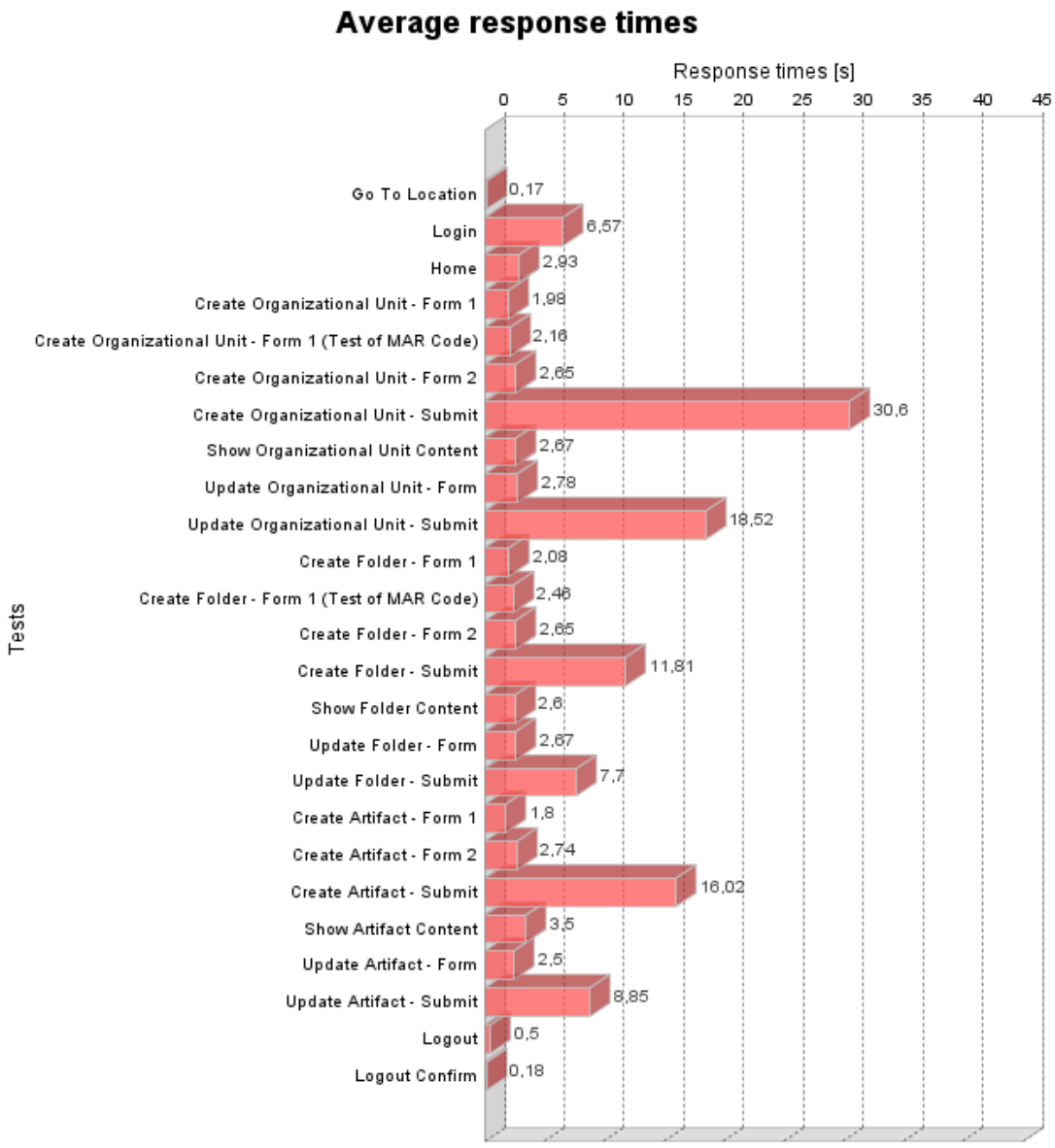
Obrázek 4.10: Detail funkčního webového testu

Na obrázku 4.10 můžeme vidět detail testovacího scénáře. Je tu vidět čas spuštění, kdo je za test kompetentní a seznam testovacích případů, ze kterých se testovací scénář skládá. Každý testovací případ se odkazuje do spodní části reportu, kde jsou zobrazeny obrazovky získané v průběhu spouštění testu a další informace týkající se průběhu testu.



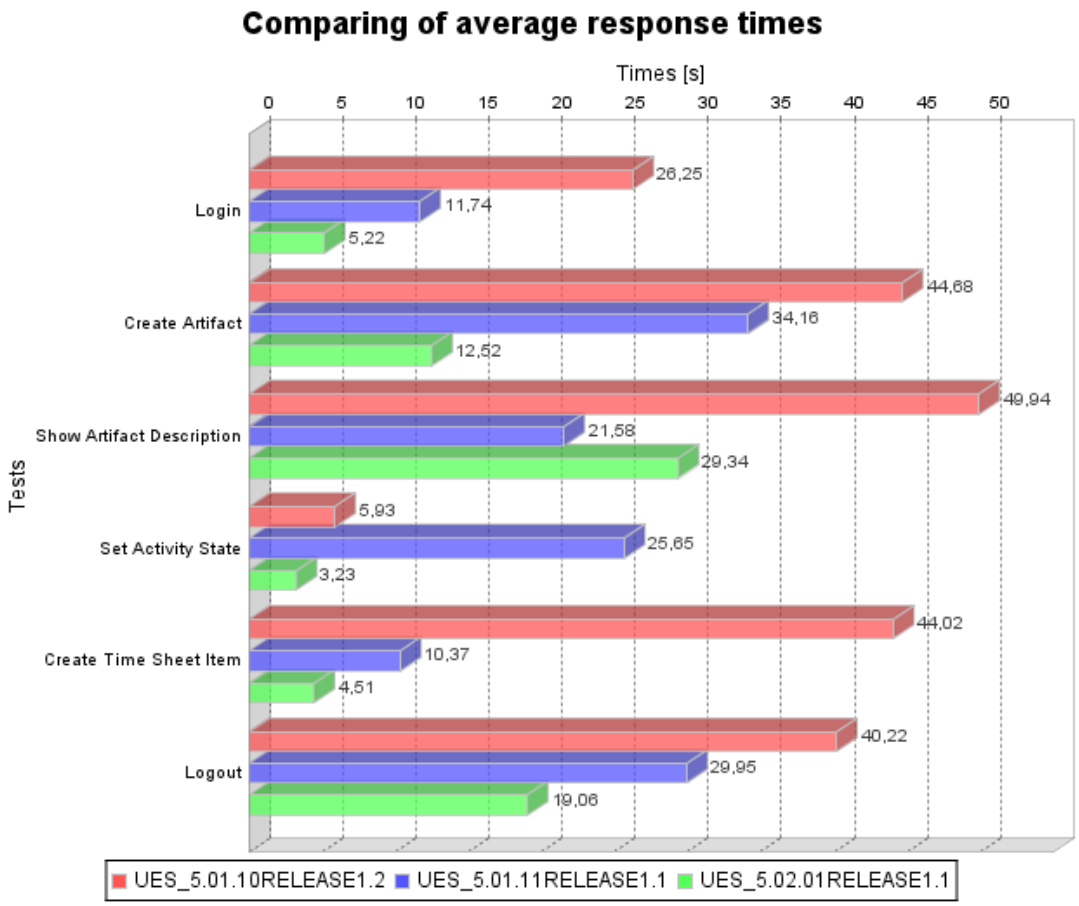

Obrázek 4.11: Přehledové porovnání několika verzí aplikace

Na obrázku 4.11 si můžeme prohlédnout jednoduché porovnání kvality více verzí testovaného produktu. Každá osa reprezentuje kvalitu z určitého úhlu pohledu (například výkon nebo pokrytí testy). Spočítáním plochy můžeme získat určitý pohled na kvalitu dané verze.



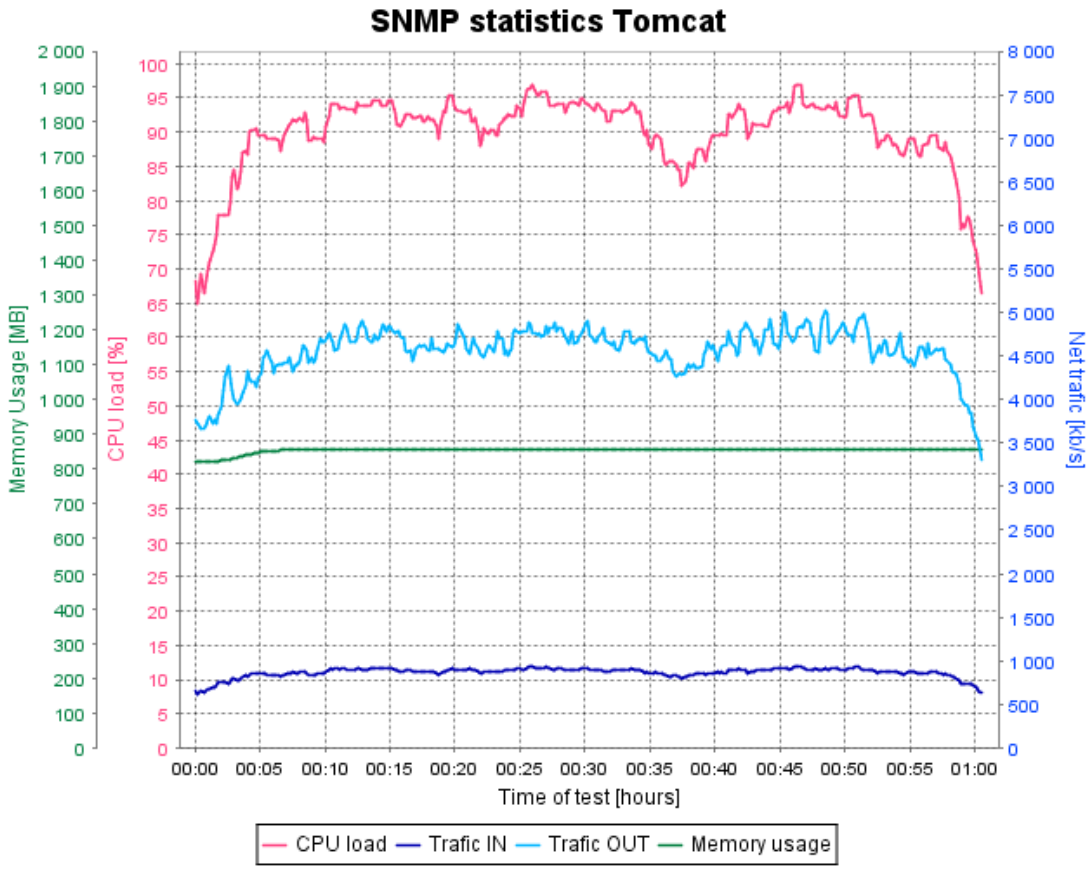

Obrázek 4.12: Doby odezvy na jednotlivé http requesty

Na obrázku 4.12 je zobrazen testovací scénář. Jsou tu znázorněny průměrné doby odezvy http serveru na požadavky, které posílají jednotlivé testovací případy.




Obrázek 4.13: Doby odezvy na jednotlivé http requesty pro více buildů

Na obrázku 4.13 jsou zobrazeny průměrné doby odezvy http serveru na požadavky, které posílají jednotlivé testovací případy. Ale na rozdíl od obrázku 4.12 porovnáváme více spuštění daného testovacího scénáře.

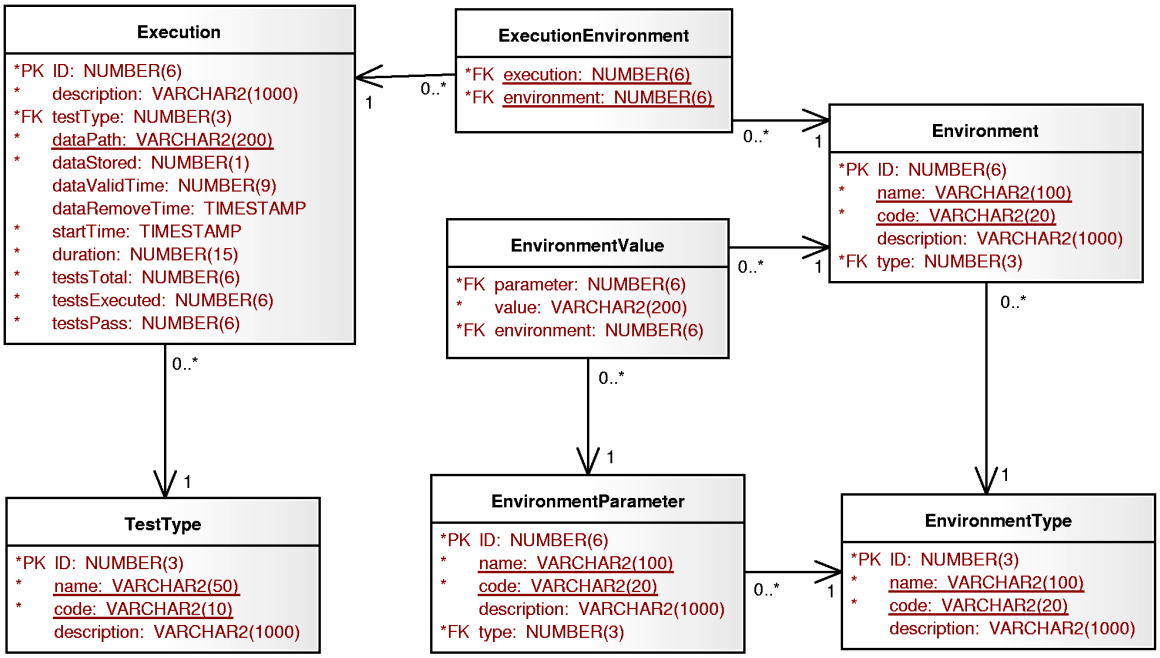



Obrázek 4.14: Zatížení serveru v průběhu testu

Na obrázku 4.14 můžeme vidět spotřebu zdrojů jednoho ze serverů, na kterém běží testy. Je zde vidět zatížení procesoru, spotřeba paměti a velikost síťového provozu.

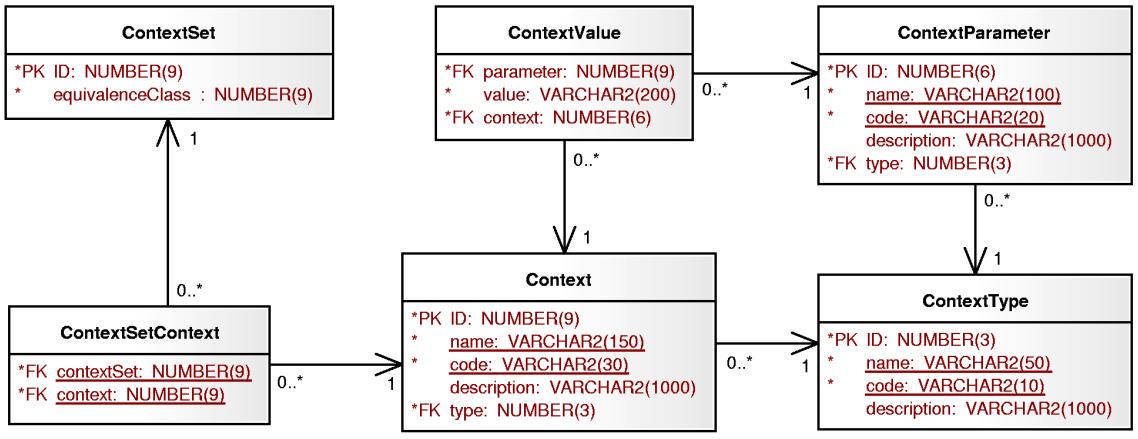
## 4.4 Datový model

Nejdříve bude popsán návrh datového modelu, který bude použit pro verzi aplikace vytvářenou standardním způsobem. V tomto návrhu uvidíme, že využití objektů s různými množinami definovaných atributů je v této aplikaci vyžadováno několikrát.




Obrázek 4.15: Spuštění testů

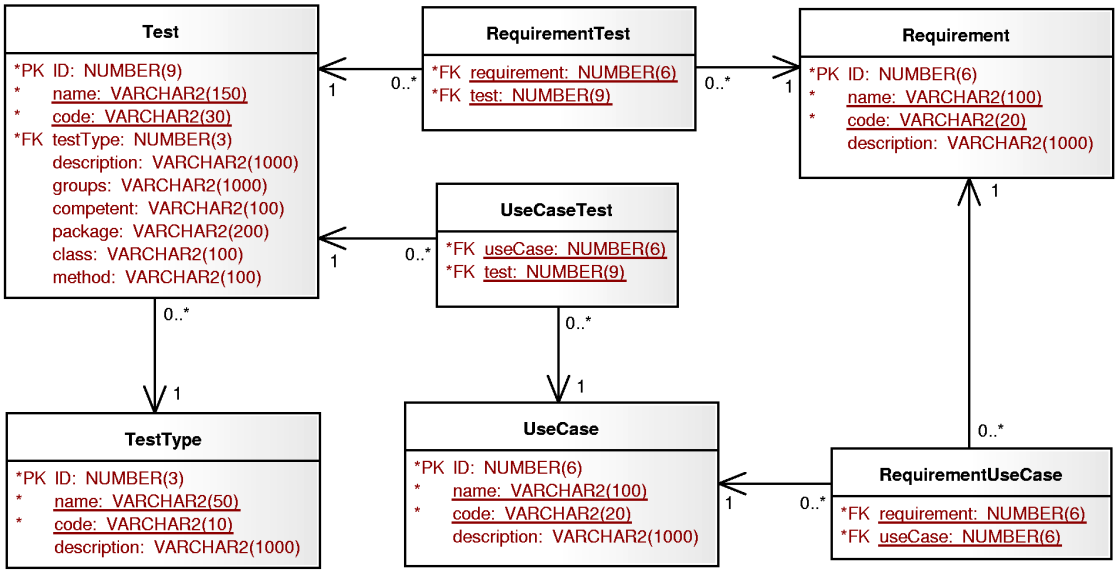
Na obrázku 4.15 je vidět spuštění testů. Každé spuštění (Execution) se týká nějakého typu testů, tedy například webových funkčních nebo zátěžových testů. Každé spuštění běží s nějakou konfigurací (ExecutionEnvironment), což je například verze buildu, konfigurace buildu nebo konfigurace prostředí, na kterém běží testy. Tyto parametry prostředí jsou pro každý test jiné, ale mění se málo často, proto jsou navrženy tak flexibilně, jelikož se vždy při ukádání spuštění testů ověří, zda už stené prostředí není v databázi jednou uloženo.




Obrázek 4.16: Kontexty testů

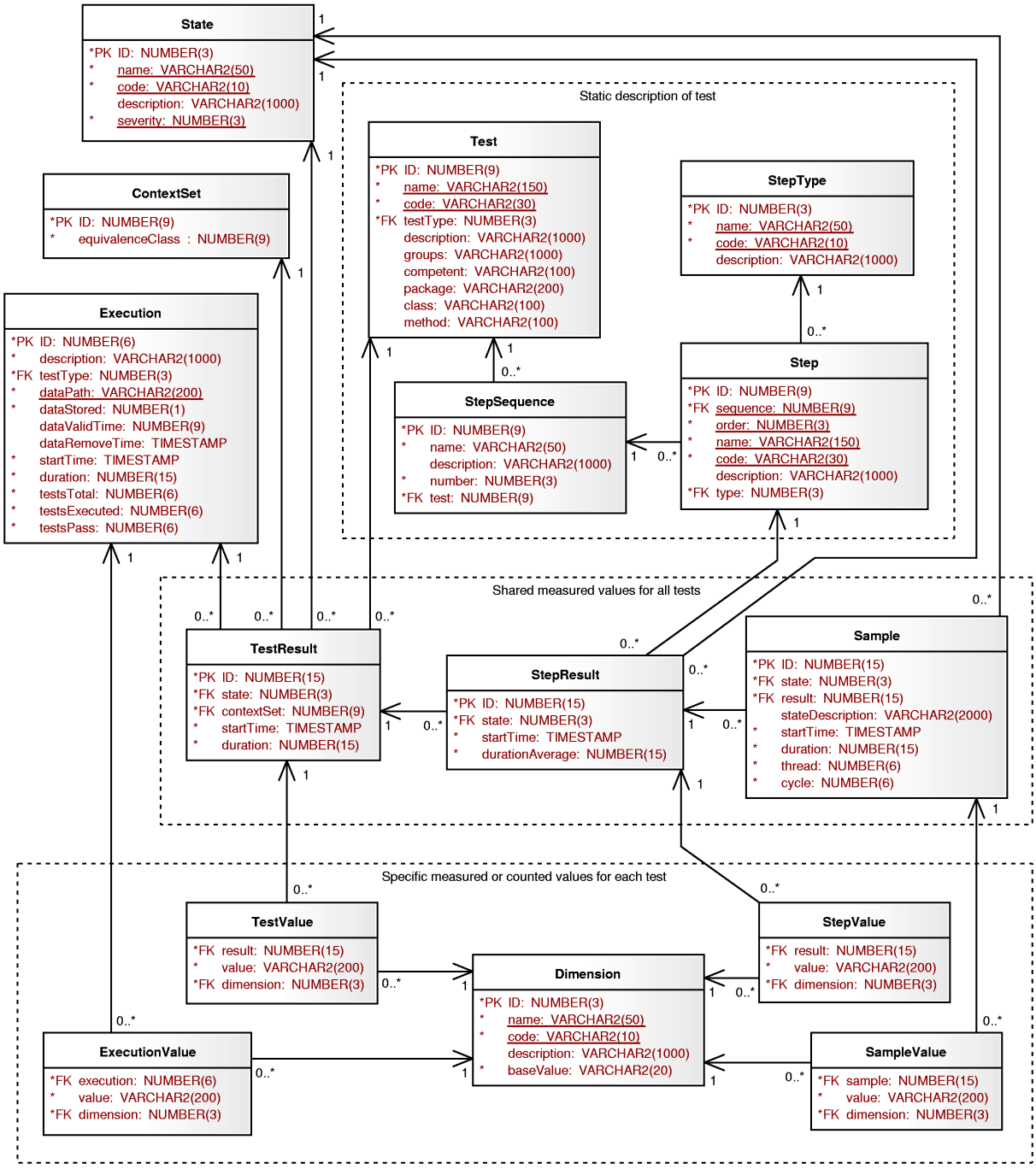
Na obrázku 4.16 je znázorněno, že každý test může být spouštěn s nějakými parametry, což je například verze testovacích dat v databázi nebo uživatel, pod kterým se test přihlašuje do

aplikace. Koncept kontextu testu je podobný jako prostředí u spuštění testu, tedy málokdy se mění hodnoty kontextu testů a již uložené objekty se tedy používají opakovaně.




Obrázek 4.17: Zařazení testů

Na obrázku 4.17 můžeme vidět možnost zařazovat testy k jednotlivým případům užití nebo požadavkům, jsou-li v aplikaci evidovány.



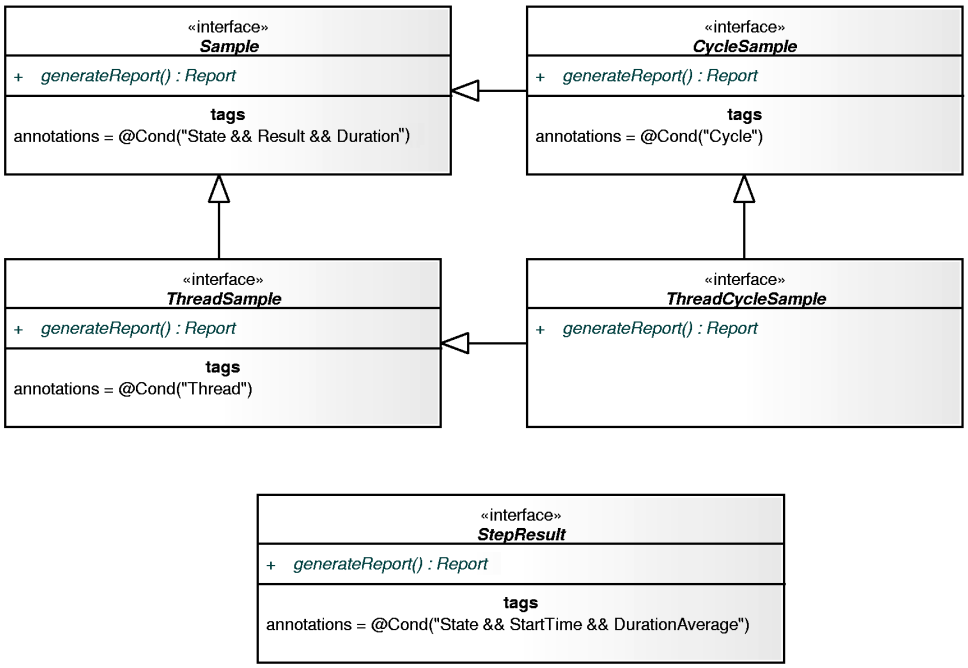

Obrázek 4.18: Výsledky testů

Na obrázku 4.18 je znázorněna nejsložitější část datového modelu, která obsahuje reprezentaci výsledků testů. Každé spuštění může mít měřené nějaké specifické parametry, které se mohou uložit do tabulky ExecutionValue. Dimension reprezentuje typ ukládaných informací. V tomto modelu lze znázornit veškeré potřebné situace. Například každý test (Test) se skládá z několika paralelně běžících posloupností (StepSequence). Ty se skládají z několika kroků (Step) s definovaným typem (StepType). Každý krok testu má výsledek (StepResult), který může obsahovat své specifické hodnoty (StepValue) a který také může mít více vlastních

výsledků měření (Sample), například když test běží ve více cyklech nebo vláknech. Každý takovýto výsledek měření (Sample) může obsahovat ještě svoje specifické naměřené nebo spočítané hodnoty (SampleValue). Každý test má pak celkový výsledek (Test result). Ten může obsahovat specifické hodnoty (TestValue). Všechny výsledky mají svůj stav (State). Je vidět, že rozvolněné objekty se zde dají použít na mnoha místech, vždy když je potřeba ukládat hodnoty atributů, které jsou v každém typu testu nebo spuštění jiné. Například se jedna o popis prostředí, kde testy běžely nebo o popis parametrů spouštěných testů, ale hlavně se jedná o jednotlivé výsledky testů. Při generování reportu je buď přistupováno ke konkrétním atributům nebo je potřeba přistupovat ke všem uloženým atributům daného objektu a ze všech takto načtených informací se může vygenerovat kompletní report s výsledky testů.

## 4.5 Porovnání obou verzí aplikace

Mohlo by se na první pohled zdát, že jsme zapoměli vytvořit datový model pro verzi aplikace realizovanou pomocí rozvolněných objektů. Mapování do relační databáze je ale dáno, takže jedinou informaci, kterou potřebujeme znát, jsou jednotlivé atributy, které se mohou do relační databáze ukládat. Je tedy potřeba jen vytvořit potřebné diagramy tříd s definicí potřebných metod včetně podmínek a díky tomu je dán i datový model.




Obrázek 4.19: Metody generující reporty s výsledky testů

Podívejme se na obrázek 4.19, kde jsou znázorněny metody generující reporty v modelu rozvolněných objektů. Máme zde rozhraní StepResult, které bude mít na starosti tvorbu reportů týkající se jednotlivých kroků testu. Aby šlo vytvořit report, musí mít rozhraní StepResult k dispozici atributy State, StartTime a DurationAverage (případně může požadovat méně atributů a další může požadavat až pro spuštění jednotlivých metod), tedy podmínka rozhraní StepResult je (State && StartTime && DurationAverage).

Mnohem zajímavější je ale případ Sample, který zapouzdřuje generování reportu pro jednotlivé naměřené vzorky (Sample). Každý vzorek se totiž může týkat buď konkrétního

vlákna, konkrétního cyklu, obojího nebo se netýká ani vlákna ani cyklu (test byl spuštěn v jediném vlákně a jediném cyklu). Podle požadovaných atributů máme definovanou hierarchii rozhraní. Sample vyžaduje atributy State, Result a Duration. V případě, že zavoláme metodu generateReport na jakémkoliv vzorku (tedy je splněna podmínka (State && Result && Duration)), zkontrolujeme, zda se nemá zavolat metoda na nějakém potomkovi. V tomto příkladu nezáleží v jakém pořadí budeme potomky prohledávat. V případě, že je definován atribut Thread, se ověří, zda není definován i atribut Cycle a zavolá se ta správná metoda. V případě, že atribut Thread definován není, ověří se atribut Cycle a opět se zavolá ta správná metoda. Když vzorek nemá definován ani atribut Thread ani atribut Cycle, tak se zavolá metoda přímo na Sample. Je vidět, že všechny varianty vedou nejkratší možnou cestou k nalezení té správné metody.

Je potřeba si uvědomit, že všechna rozhraní dědí všechny požadavky od svých rodičů (podmínky na rozhraní, ale i podmínky jednotlivých metod), tedy rozhraní ThreadCycleSample i když explicitně neobsahuje žádnou podmínku, tak pro náležení rozvolněného objektu do třídy ThreadCycleSample je potřeba splnit podmínku (State && Result && Duration && Thread && Cycle).

Pro úplnost dodejme, že tento diagram byl vygenerován automaticky ze zdrojového kódu programem Enterprise Architect[4]. Díky tomu, že jsme podmínky definovali jako anotace, jsou tyto podmínky součástí vygenerovaného modelu. Je možné také nakreslit příslušný diagram tříd a z něj nechat vygenerovat zdrojový kód.

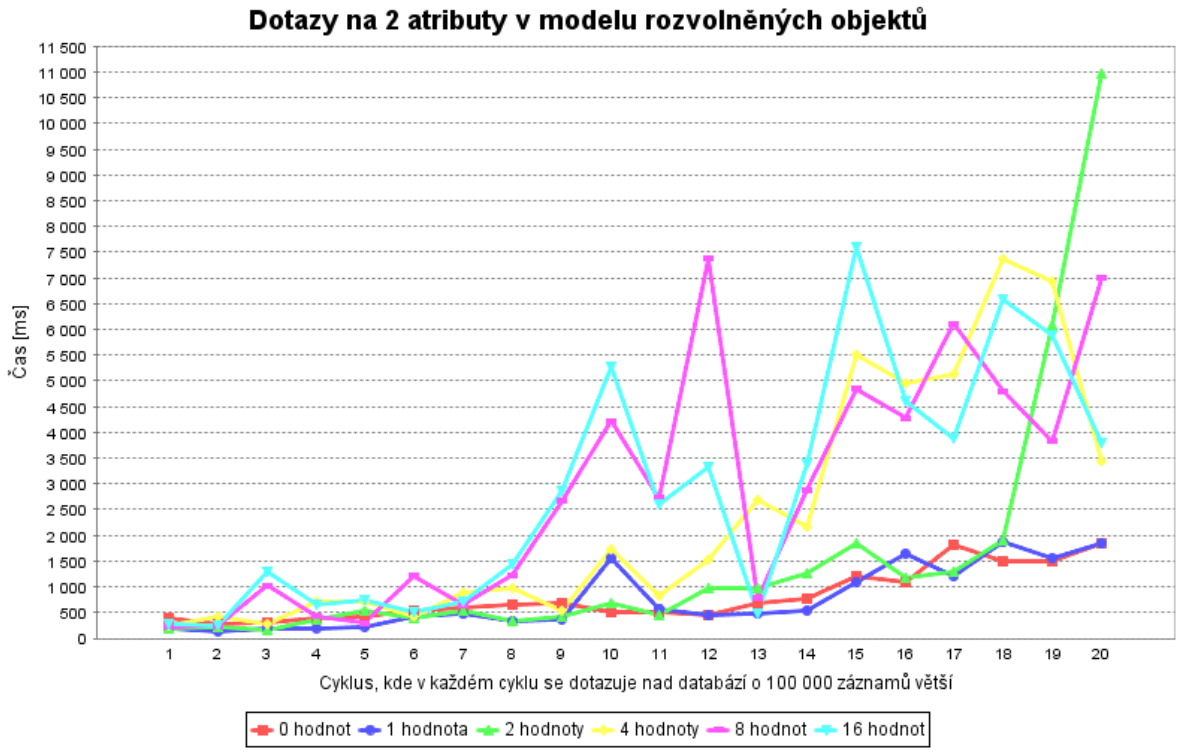
Pokud porovnáme tvorbu návrhu obou verzí aplikací, tak se můžeme převědčit, že návrh pomocí rozvolněných objektů je jednodušší, protože nemusí explicitně z diagramu tříd ještě vytvářet datový model. Na druhou stranu rozvolněné objekty zase musí při tvorbě diagramu tříd explicitně vyjmenovávat všechny své požadavky.

Na obrázcích 4.20 a 4.21 je porovnán výkon rozvolněných objektů a standardního objektového modelu. Ukážeme si výsledky testu, na kterém se rozdíly nejvíc projevily[5]. Jedná se o dotaz na tři atributy, kde hodnotu jednoho z nich přímo zvolíme. Tedy chceme načíst dva nové atributy. Výsledek dotazu je 5000 záznamů, což reprezentuje počet záznamů reálného zátěžového testu. V obou případech je na známému atributu vytvořen index. Standardní model přistupuje k datům v podstatě v konstantním čase vzhledem k počtu záznamů uložených v databázi, ale v modelu rozvolněných objektů čas přístupu k datům s rostoucí velikostí databáze výrazně vzrůstá. Dotazy byly kladeny nad rozšiřující se databází. Před každým novým testem bylo vloženo 100000 objektů. Dotazy byly pokládány jen na data, která nebyla použity v žádném z předchozích dotazů (aby databáze nemohla použít cache). V grafu je zobrazen vždy průmě dotazů nad stejně velikou databází. Barvy v grafu rozjišují počty objektů, které se vkládaly navíc do databáze za každý vložený objekt.

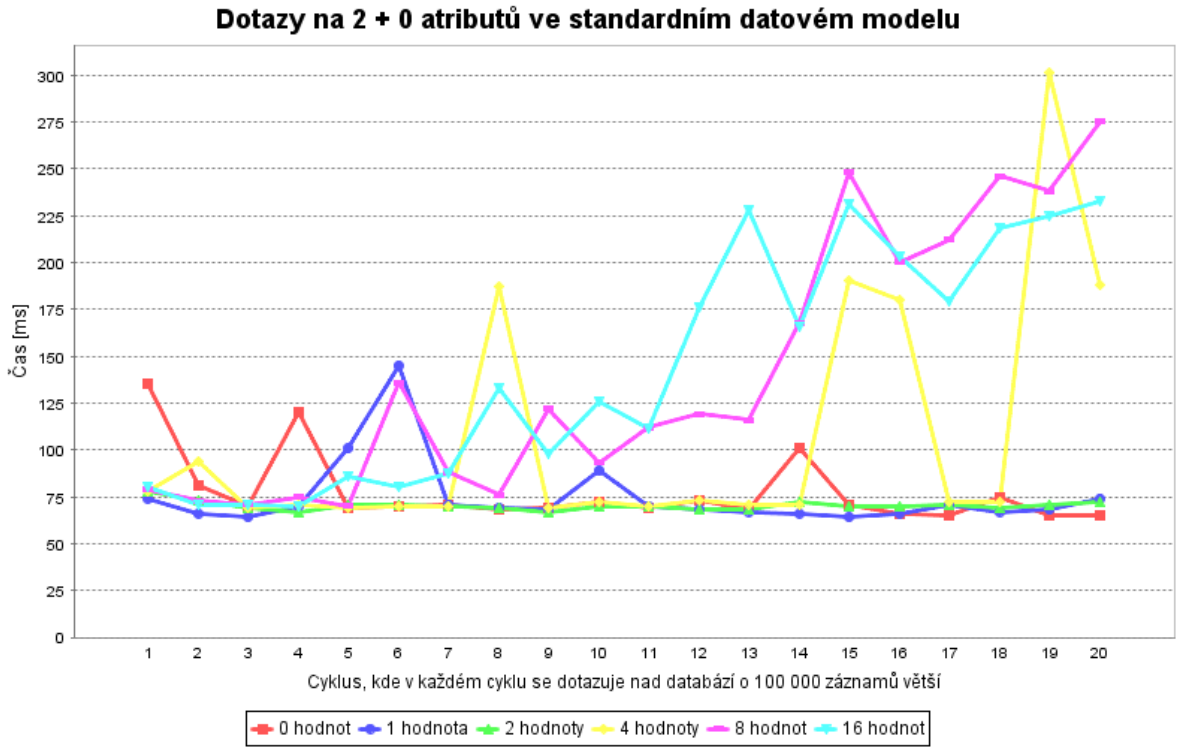
Vkládání objektů do databáze bylo u modelu rozvolněných objektů několikrát pomalejší, jelikož se objekty musí vkládat do několika tabulek. Počet vložení do databáze byl v testech za jednotku času zhruba konstantní, takže lze docela přesně odhadnout, jak se vkládání rozvolněných objektů bude chovat.

[4] http://www.sparxsystems.com.au/

[5] Všechny výsledky testů včetně popisu testu je možné nalézt na přiloženém cd




Obrázek 4.20: Dotazy na 2 atributy v modelu rozvolněných objektů




Obrázek 4.21: Dotazy na 2 atributy ze stejné tabulky ve standardním objektovém modelu

# Kapitola 5

# Možnosti rozšíření rozvolněných objektů

## 5.1 Neimplementované vlastnosti

Implementace byla provedena pro části modelu, které ukázková aplikace nutně potřebovala pro svoje spuštění a takovým způsobem, aby bylo možné výsledný framework simulovat. Nebyla implementována podpora pro dotazy, časová podpora a podpora pro triggery. Testování dotazů bylo prováděno spouštěním příkazů, které by se po rozpársování podmínek vytvořily. Vyhodnocování podmínek bylo implementováno formou prototypu, kde se na jednoduchých přikladech zkoumalo, jaký způsob je nejvhodnější.

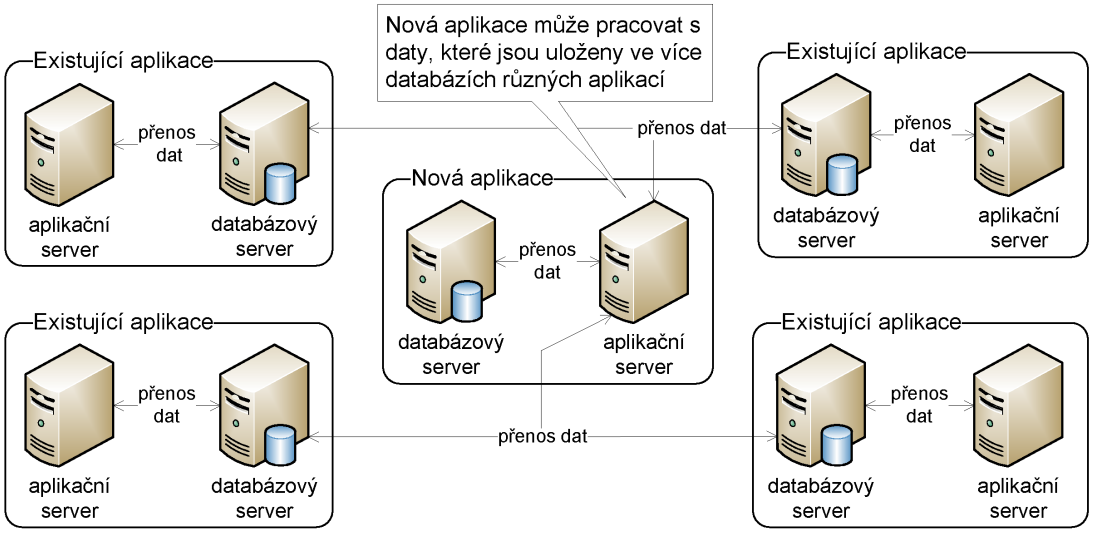
## 5.2 Nově navrhované vlastnosti

V průběhu tvorby této diplomové práce bylo objeveno několik nedostatků navrhovného modelu. Většina z nich je zmíněna v závěrečném shrnutí v kapitole 6. My si ukážeme návrhy dvou nejdůležitějších řešení, které by nás zbavily všech zásadních zjištěných problémů. Uložení jednotlivých atributů do vlastních sloupců v relační databázi vyžaduje časté použití spojení v dotazech. Abychom se tomuto vyhnuli, je možné uvažovat ne o definici jednotlivých atributů, ale o množinách atributů, které by se společně ukládaly do stejné tabulky. V ukázkové aplikaci by toto řešení bylo plně dostačující, jelikož u některých objektů požadujeme, aby obsahovaly dané atributy (například u všech vztahů s větší aritou je potřeba definovat všechny atributy) a naopak je výhodou, že databáze neumožní uložit objekty s nedefinovanými hodnotami požadovaných atributů.

Dalším problémem je fakt, že všechny objekty sdílí stejné atributy. Například v tabulce se seznamem id všech objektů jsou uloženy i id objektů systémového katalogu. To může komplikovat zapisování podmínek metod nebo dotazů, jelikož může být potřeba tyto objekty do výsledků dotazů nezahrnovat. Navíc se často stává, že informace obsažené v atributu, které by ve standardním objektovém modelu byly uloženy ve dvou tabulkách, tak v modelu rozvolněných objektů jsou v jediné tabulce. Například atribut Name a Code se dá předpokládat u velkého množství tabulek. Potom například přístup ke konkrétnímu jménu daného číselníku je zpomalován všemi ostatními existujícími číselníky. Proto by bylo vhodné zavést například namespacy nebo implicitní typy. Každá hodnota by se ukládala do tabulek definovaných pro daný typ a tvořila by tak logicky oddělenou množinu objektů (například zvlášť zvířata, lidi, budovy nebo číselníky). Ve většině případů se pracuje jen v rámci jednoho typu, ale v případě potřeby by bylo možné přistupovat k datům více typů zároveň. Také by se zrychlil přístup k požadovaným datům, jelikož by se data rozdělily do více tabulek.

## 5.3 Integrační platforma

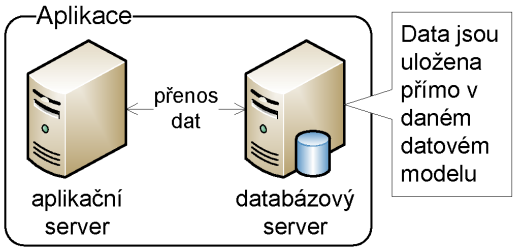
Potřeba pracovat s daty získanými z různých zdrojů je čím dál více důležitější pro většinu firem. Může se jednat například o sloučení více firem, kde každá z firem používala k uložení svých dat jiný způsob, ale ve výsledku je potřeba pracovat se všemi daty stejným způsobem. Dalším příkladem může být výběr dodavatelů pro konkrétní položky objednávky. Kdyby existovala možnost vybírat ze všech dodavatelů současně podle různých kritérií, tak realizovat například nejlevnější nebo nejrychlejší objednávku by bylo velmi jednoduché. Kdyby navíc existovala možnost si jednotným způsobem přímo u vybraných dodavatelů objednávku vytvořit, ušetřilo by to firmám spoustu času a tím i peněz. Základní myšlenkou je tedy vytvořit integrační platformu, která bude umožňovat aplikacím sdílet mezi sebou svá data. Existujícím aplikacím poskytne možnost nasdílet svá data a nově vytvářeným aplikacím poskytne možnost tyto data používat.



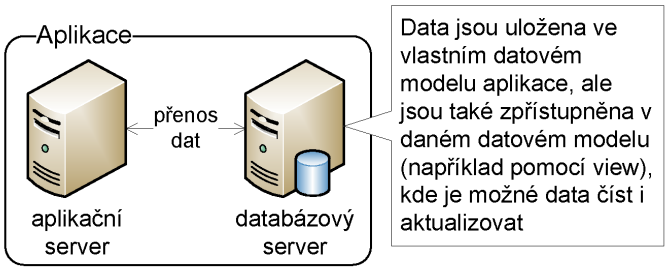

Obrázek 5.1: Spolupráce více aplikací, které používají několik databází

### 5.3.1 Sdílený datový model

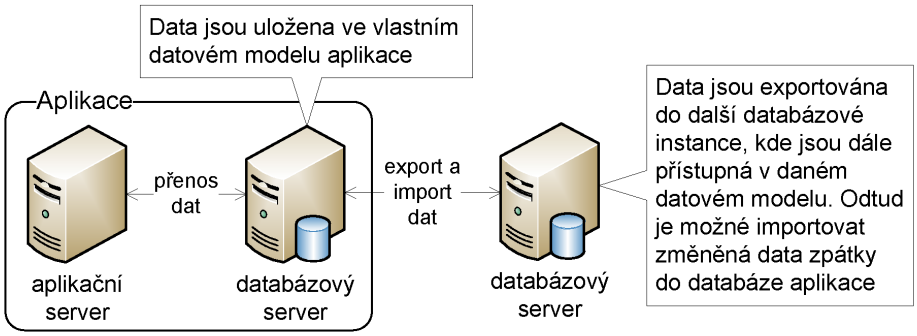
Pro jednoduchost tvorby nových aplikací pracujících s více databázemi se použije pro všechny aplikace společný datový model, který byl představen v této diplomové práci. Kdyby totiž každá aplikace mohla mít teoreticky svůj vlastní datový model, bylo by pak téměř nereálné psát aplikace nad více datovými modely, které by mohly i přibývat za běhu aplikace. Existuje několik možností, jak s tímto sdíleným datovým modelem pracovat.




Obrázek 5.2: Uložení dat ve sdíleném datovém modelu




Obrázek 5.3: Zpřístupnění dat ve sdíleném datovém modelu



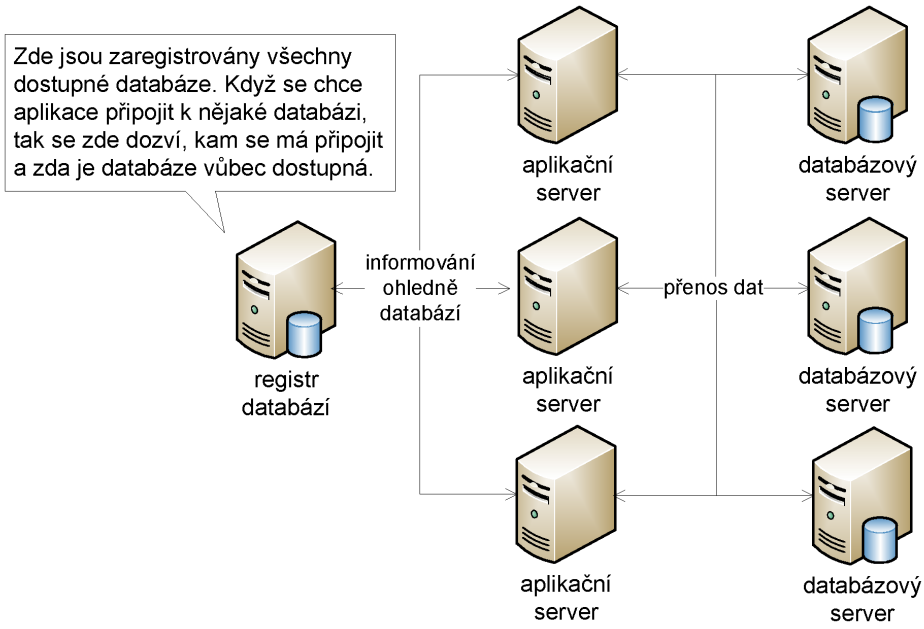

Obrázek 5.4: Export dat do sdíleného datového modelu

Ve všech třech případech je možné nastavit speciálně možnost číst data a zapisovat data. Lze tedy například nasdílet data jen pro čtení nebo jen pro zápis. První dvě varianty obsahují vždy aktuální data, ale poslední varianta obsahuje data platná v době exportu dat z aplikace. Poslední varianta je výhodná například z důvodu výkonu, aby zpřístupňování dat nezpomalovalo původní aplikaci.

Jelikož je potřeba pracovat s daty už existujících aplikací, tak některé databázové systémy už jsou dané. Proto je potřeba, aby sdílený datový model neobsahoval specifické požadavky konkrétního databázového systému. Nejlepší řešení je použít co nejrozšířenější normu, například SQL 92, aby bylo možné se připojovat například do oraclu i postgres najednou, bez toho, aby nově vytvářené aplikace poznaly jakýkoliv rozdíl v přístupu k požadovaným datům.

### 5.3.2 Registr databází

Nové aplikace, které budou sdílet svoje data, mohou samozřejmě přibývat za běhu ostatních aplikací. Proto je potřeba, aby existoval seznam všech dostupných databází, který musí být pořád dostupný všem aplikacím.




Obrázek 5.5: Registr databází

Bylo by možné zaregistrovat databázi s nějakým unikátním jménem, případně jí zrušit nebo se zeptat, zda dané jméno databáze už je registrované. Dále by bylo možné evidovat atributy patřící dané databázi. Mezi nastavované atributy by patřilo například informace o připojení k databázi, datum poslední aktualizace dat, informace, zda je systém momentálně dostupný nebo informace ohledně základních práv čtení a zápisu do dané databáze. Každá aplikace může mít nastaveny různá práva přístupu k datům. Samozřejmě lepším řešením by bylo poskytnout aplikacím naříklad lokální registry databází, aby v případě výpadku hlavního serveru, aplikace mohli přistupovat alespoň k lokálnímu.

# Kapitola 6

# Závěr

Cílem této práce bylo ověření konceptu rozvolněných objektů. Přesto, že došlo k implementaci jen nejdůležitějších částí modelu a ukázkové aplikace, tato práce splnila svůj účel.

Bylo navrženo řešení v Javě, jehož použití ukazuje výhodnost použití modelu rozvolněných objektů pro řešení určité sady problémů. Toto řešení obsahuje i polymorfismus, který v původním modelu rozvolněných objektů vůbec nebyl navržen. Na ukázkové aplikaci bylo zjištěno, jaké výhody a nevýhody tvorba aplikace s rozvolněnými objekty přináší. Díky rozšířenosti a propracovanosti Javy ve verzi 6.0 se ukázalo, že takto netypický koncept lze v Javě realizovat a využívat nejnovější technologické vymoženosti této platformy. V plánovaných rozšířeních Javy jsou další vlastnosti, které by tomuto konceptu výrazně pomohly. Například názvy parametrů metod se mají stát veřejnými a mají být v bytecodu k dispozici vývojářům, stejně jako je dnes k dispozici informace o typu parametrů metody. Tím by se výrazně zvýšila srozumitelnost zápisu podmínek, neboť metody po zveřejnění názvu svých parametrů by neměly možnost tyto názvy změnit a bylo by tedy možné se na ně s jistotou odvolávat.

Kompletní zavedení konceptů rozvolněných objektů do reálného vývoje aplikací je velmi časově náročný proces. V této práci byli analyzovány jen základní běžně rozšířené potřeby aplikací. Mnoho podobně atypických konceptů bylo zaváděno do vývoje několik desítek let. Například zmiňme v současné době nejrozšířenější aspektově orientované programovaní[1] nebo koncept naked objects[2], který se zaměřuje na myšlenku, že po definici doménových objektů lze většinu uživatelského rozhraní automaticky vygenerovat. Je zřejmé, že pro tvorbu reálných aplikací se koncept naked objects bude prosazovat ještě v řádu minimálně let (jestli se ovšem vůbec prosadí).

Jelikož rozvolněné objekty, tak jak jsou navrženy v této práci, definují aplikaci v podstatě jako množinu metod, které jen specifikují své požadavky na potřebná data, lze od sebe úplně oddělit aplikační a databázovou vrstvu. Lze si představit, že po napsání aplikační části, tedy sady metod, lze pomocí podmínek specifikovaných v metodách automaticky vygenerovat databázovou vrstvu. Díky tomu, že potřebujeme tyto pomínky za běhu aplikace kontrolovat, máme možnost si dělat statistiky o spouštěných metodách a jejich parametrech, tedy můžeme pro konkrétní nasazení aplikace vygenerovat optimální databázovou vrstvu. Největší rozdíl oproti současnému přistupu by ale byl ten, že databázová vrstva by se při vývoji aplikace už nemusela vůbec řešit.

[1] http://en.wikipedia.org/wiki/Aspect-oriented_programming
[2] http://www.nakedobjects.org/tutorial/developer-tutorial.shtml

Nakonec ještě shrňme hlavní výhody návrhu rozvolněných objektů, který byl proveden v této práci:

- možnost pracovat s různorodými daty s možností volat na nich všechny metody, které dává smysl volat
- zvýšená odolnost proti chybám díky ověřování podmínek metod za běhu aplikace
- transparentní persistence objektů s jednoduchou a bezpečnou definicí mapování objektů do relační databáze
- současný návrh využívá nejnovějších technologických vlastností Javy jako jsou Annotations, Annotation Processing a Java Instrumentation API, díky kterým je možné například napsat plugin do vývojové prostředí, který bude kontrolovat syntaxi podmínek metod ve zdrojovém kódu, nebo je možné převádět zdrojový kód metod na diagram tříd nebo naopak
- přidáním polymorfismu jsou možnosti rozvolněných objektů srovnatelné se standardním objektovým modelem

Bohužel seznamem výhod tato práce nekončí a musíme zmínit i nedostatky současného návrhu:

- největším problémem je vlastní uložení do databáze po jednotlivých sloupcích. V aplikaci se ukázalo, že často potřebujeme načítat několik atributů, přičemž omezení máme jen na jeden (například chceme vidět seznam testů s výsledky, které jsou obsaženy v daném spuštění). Toto řešení vyžaduje provést dvě spojení, přičemž každé spojení znamená výrazný pokles výkonu. Ve standardním mapování máme možnost mít jedinou tabulku se třemi atributy a jediným indexem a můžeme přistupovat k požadovaným záznamům v podstatě v konstantním čase v závislosti na velikosti tabulky. Se zvyšujícím se počtem načítaných atributů se řešení stává nepoužitelnější. Samozřejmě lze problém vyřešit pomocí materializovaného pohledu nebo pluginu do používané databáze, ale v prvním případě zpomalíme vkládání a ve druhém jsme závislí na konkrétní databázi. Nejlepším řešením se proto jeví kombinovat přístup, kdy ukládáme více atributů do stejné tabulky a jen atributy, které nejsou povinné, se ukládají do zvláštních tabulek. Aplikace úložiště výsledků testů by tím způsobem optimálně využila výhod obou přístupů a byla by ve všech operacích rychlejší. Toto je zásadní problém současného návrhu, všechny následující problémy už jsou jen drobné
- zpomalení a větší paměťové nároky na aplikační vrstvě díky nutnosti poskytovat možnost dodefinovávat hodnoty atributů. Opět, zkombinováním přístupu, že nějaké atributy jsou povinné a nějaké volitelné, se výrazně omezí tento problém
- v případě, že potřebujeme reprezentovat více číselníků, je potřeba jednotlivé číselníky v podmínkách rozlišovat zvláštním atributem (například TestType==JUnit). Toto řešení je velmi nepraktické, protože se musí načítat hodnota dalšího atributu a definice podmínek je nepřehlednější. Řešením je definovaní typů přímo v modelu rozvolněných objektů. Součástí definicí podmínek by bylo určení typu objektu, na který se ptáme. V případě neuvedení by se použil výchozí typ. Navíc by toto řešení mohlo výrazně zrychlit aplikaci tím, že by jednotlivé typy ukládalo do zvláštních tabulek a jednotlivé typy by tedy mohly používat různá mapování. V praxi by toto řešení bylo často postačující, jelikož z člověka se málokdy stává číselník nebo zřícenina

- dalším nedostatkem návrhu je nedostatečná podpora agregačních funkcí. V standardním SQL je možné přímo používat agregační funkce, ale v navrženém modelu je nutné výsledky dotazů dále explicitně zpracovávat, což není moc uživatelsky přívětivé
- posledním úplně drobným nedostkem, který je spíše novým požadavkem, je podpora agregačních funkcí s podporou času. Například chceme zobrazit průměrné zatížení procesoru v jednotlivých minutách/hodinách/dnech požadovaného intervalu

Přesto, že jsme narazili na několik nedostatků modelu, navržený model prokázal svou užitečnost při tvorbě aplikací.

# Kapitola 7

# Seznam literatury


[1] Michal Kopecký a Michal Žemlička, *Rozvolněné objekty*, In: Ježek, K. (Ed.): DATAKON 2004. FAV ZČU, Plzeň, 2004. ISBN 80-210-3516-1. pp. 243-252.

[2] M. Stonebraker, *C-Store: A Column-oriented DBMS*, In: Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005.

[3] J. Kulhánek a D. Obdržálek, *Generating and handling of differential data in datapile-oriented systems*, Proceedings of the 24th IASTED international conference on Database and applications, Innsbruck, Austria, 2006.

[4] Johannes Rieken, *Design By Contract for Java - Revised*, 2007, http://modernjass.sourceforge.net/docs/mastersthesis-johannes_rieken.pdf, Navštíveno 6.8.2009.

[5] Rudolf Pacinovský, *Návrhové vzory*, Computer Press, a.s., ISBN 978-80-251-1582-4, 2007.

[6] Joshua Bloch, *Java efektivně*, 57 zásah softwarového experta, Grada Publishing, ISBN 80-247-0416-1, 2002.

[7] Michal Žemlička, *ACASE: Programování s podrobnou specifikací rozhraní*, In: Sborník konference Objekty'99. Praha, 1999.