České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Ondřej Mirtes**

Studijní program: Otevřená informatika (magisterský)
Obor: Softwarové inženýrství

Název tématu: **Framework pro správu a vyhodnocení byznys pravidel s podporou transformace do různých platforem**

Pokyny pro vypracování:

Proveďte rešerši problematiky současného vývoje webových aplikací a přístupů k zachycení byznys pravidel.
Navrhněte a naimplementujte framework v PHP pro zápis a parsování pravidel ve vlastním doménově specifickém jazyku. Zaměřte se na automatickou transformaci pravidel do prostředí relační databáze pro eliminaci duplicit ve zdrojovém kódu aplikace: překlad do integritních omezení a SQL dotazů za pomoci Doctrine 2 ORM s možností manuální transformace u konkrétních pravidel.
Implementovaný framework bude podporovat statickou analýzu pravidel pro hledání chyb a generování statistik.
Navrhněte "best practices" pro psaní a organizaci většího počtu byznys pravidel, aby byla zajištěna snadná údržba a rozšiřitelnost systému, který je používá. Napište na implementovaný framework jednotkové testy a demonstrujte ho na ukázkové aplikaci o rozsahu alespoň pěti databázových tabulek.
Zhodnoťte možné oblasti použití frameworku a další cílová prostředí vhodná pro transformaci pravidel.

Seznam odborné literatury:

[1] Tomas Cerny, Michael J. Donahoo, and Eunjee Song. 2013. Towards effective adaptive user interfaces design. In Proceedings of the 2013 Research in Adaptive and Convergent Systems (RACS '13). ACM, New York, NY, USA, 373-380. DOI=10.1145/2513228.2513278 http://doi.acm.org/10.1145/2513228.2513278
[2] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song. 2013. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. In Applied Computing Review, Vol. 13, Issue 4, ACM, New York, NY, USA, 53-65. ISSN 1559-6915 http://www.sigapp.org/acr/Issues/V13.4/ACR-13-4-2013.pdf
[3] Cemus, Karel and Cerny, Tomas, 2014. Aspect-Driven Design of Information Systems. In SOFSEM 2014: Theory and Practice of Computer Science, LNCS, vol. 8327, 174-186, Springer International Publishing Switzerland 2014, isbn 978-3-319-04298-5

Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2014/2015

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 19. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Diplomová práce

Framework pro správu a vyhodnocení byznys pravidel s podporou transformace do různých platforem

Bc. Ondřej Mirtes

Vedoucí práce: Ing. Tomáš Černý

4. května 2014

Poděkování

Děkuji Ing. Tomáši Černému za vstřícné vedení práce a cenné rady.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 4. května 2014 .....

Odkaz na tuto práci

Mirtes, Ondřej. *Framework pro správu a vyhodnocení byznys pravidel s podporou transformace do různých platforem.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta elektrotechnická, 2014.

Abstract

The goal of this thesis is to implement a framework for managing and evaluating business rules for the purpose of their transformation to diverse platforms. Framework enables developers to avoid code duplication, makes maintenance easier and lowers error rate during development.

Amphibian framework is implemented in the PHP language. It supports evaluating of user-defined rules in PHP and transforming them to SQL in order to query data stored in relational databases. It uses Doctrine 2 ORM and similar language called DQL to achieve that.

Thesis also includes unit tests and a sample web application to showcase capabilities of the framework.

Keywords Business rules, framework, ORM, SQL.

Abstrakt

Předmětem práce je realizace frameworku pro správu a vyhodnocování byznys pravidel za účelem jejich transformace do různých platforem. Díky tomu framework umožňuje vyhnout se duplikaci kódu, usnadnit tak údržbu aplikací a snížit chybovost při vývoji.

Framework Amphibian je implementovaný v jazyce PHP, ve kterém podporuje i vyhodnocování zapsaných pravidel. Vedle toho umožňuje jejich transformaci do jazyka SQL, ve kterém aplikace komunikují s relačními databázemi. K tomu jako prostředníka využívá knihovnu Doctrine 2 ORM a její příbuzný jazyk DQL.

Součástí práce jsou jednotkové testy a ukázková webová aplikace demonstrující možnosti frameworku.

Klíčová slova Byznys pravidla, framework, ORM, SQL.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Úvod

V současnosti existuje mnoho metodik, principů a doporučení pro vývoj software. Tyto postupy si kladou za cíl usnadnit a zrychlit proces tvorby a zkvalitnit výsledný produkt. Práce softwarového inženýra spočívá především ve studiu, výběru, tvorbě a aplikaci těchto postupů takovým způsobem, aby byly splněny požadavky projektů a dovedeny k úspěšnému a včasnému dokončení. Jedná se o netriviální disciplínu. Vedle samotného softwarového inženýrství musí zapojení lidé rozumět i oborům, do kterých vytvářená aplikace zasahuje. Z těchto důvodů je až 59 % IT projektů neúspěšných z toho pohledu, že nebyl splněn alespoň jeden cíl projektu – časový, rozpočtový nebo kvalitativní [20].

Vývoj webových aplikací přináší další výzvy, protože je do jejich tvorby oproti jiným druhům aplikací zapojeno mnoho různorodých technologií a postupů, které je třeba znát a umět. Spadají do těchto kategorií:

- Aplikační technologie na straně serveru – např. PHP, Java, Python.
- Persistentní datové úložiště – relační či jiné databáze.
- Webový server a jeho konfigurace.
- Protokol síťové aplikační vrstvy – HTTP.
- Výstupní formáty dokumentů – HTML, XML, JSON.
- Definice vzhledu dokumentů – CSS.
- Skriptování na straně klienta (ve webovém prohlížeči) – JavaScript.

V takto fragmentovaném prostředí je aplikování některých principů ještě složitější, protože to nelze snadno provádět napříč jednotlivými kategoriemi.

Mezi ty nejdůležitější patří „Don't repeat yourself" známé také jako DRY [19]. Klade si za cíl usnadnit vývoj a údržbu software tím, že nařizuje, aby každá skutečnost/znalost/informace měla v systému právě jednu jednoznačnou a autoritativní reprezentaci.

Pokud bychom toto nedodrželi a měli stejnou informaci vyjádřenou na více než jednom místě, vystavujeme se riziku desynchronizace. Pokud přijde požadavek na úpravu funkcionality, která s touto informací souvisí, nic nás implicitně nenutí upravit obě místa najednou. Pokud jednu z reprezentací neupravíme vůbec nebo neodpovídajícím způsobem, v systému vznikne rozpor a ten může vést k nežádoucímu chování.

Duplikace může vzniknout nechtěně, ale i úmyslně. Zkopírování informace totiž zpočátku působí jako méně práce, než její přizpůsobení znovupoužitelnosti, komplikace ale nastanou při první úpravě.

Někdy se duplikaci nedá vyhnout z technických důvodů. Pokud vyvíjíme webovou aplikaci a potřebujeme tu samou logiku vykonat např. v serverové části a v relační databázi, nezbývá nám, než danou informaci zduplikovat, protože programování pro tyto dvě platformy je vzájemně nekompatibilní. Probíhá v odlišných jazycích. Problematiku a výzvy ve vývoji webových aplikací více rozebírám v kapitole 1.

Mým cílem je tuto bariéru mezi aplikací na serveru a relační databází vhodným způsobem odstranit. Za tímto účelem jsem vyvinul framework Amphibian[1], který umožňuje zapisovat výrazy, jejichž rozborem lze dosáhnout vyhodnocování jak v PHP, tak v relační databázi.

Kromě ověření veškeré funkcionality pomocí automatizovaných testů jsem vytvořil i ukázkovou aplikaci demonstrující možnosti frameworku na reálných příkladech. Popisuji ji v kapitole 11.

[1] *Amphibian* jako obojživelník. Nástroj, který dokáže pracovat ve dvou různých prostředích.

Část I

Rešerše

Kapitola 1

Problematika a výzvy současného vývoje webových aplikací

1.1 Škálování

Webové aplikace fungují na principu klient-server. Strana klienta je reprezentována webovým prohlížečem, který se pomocí protokolu HTTP dotazuje webového serveru na data. Webové služby jsou globálně dostupné a mohou mít obrovské množství uživatelů. Server je tedy možným terčem zátěže. Řešení tohoto problému se nazývá škálování a existují dva druhy – vertikální a horizontální. U vertikálního se kapacituje navyšuje tak, že na jediném serveru, kde aplikace běží, se přidávají další hardwarové prostředky – přidává se RAM, vyměňuje se CPU za lepší apod. Kvůli omezením hardwaru ale nelze takto růst donekonečna, navíc tento systém neposkytuje žádnou redundanci při výpadku. Horizontální škálování naopak nespočívá v růstu jednoho uzlu, ale v přidávání uzlů. Dnes je založení serveru díky virtualizaci otázka několika kliknutí (v případě poskytovatelů typu IaaS) nebo plně automatická (v případě PaaS). Aplikace musí být provozu na více serverech uzpůsobená, je třeba zajistit konzistentní pohled na rozštěpené datové úložiště pomocí vhodného přístupu k tzv. shardingu. Způsob, jakým rozdělit data na víc strojů tak, aby všechny stroje byly zatíženy rovnoměrně, data, která jsou často vybírána společně, byla na stejném stroji a aby byl umožněn další růst [31].

Problematika škálování je tedy dostatečně prozkoumána, popsána a v dnešním světě globálních služeb běžně aplikována.

1.2 Podpora různých prohlížečů

Dalším problémem, kterému vývojáři čelí, je podpora různých webových prohlížečů. Standard jazyka HTML je spravován a popsán konsorciem W3C [35]. Ne všechny prohížeče ho ale interpretují naprosto shodně a správně, navíc se musí potýkat s tím, že některé dokumenty ani nejsou validní podle standardu. V zájmu uživatelské přívětivosti si ale musí poradit s interpretací a vykreslováním i těchto dokumentů. Situaci na desktopu zjednodušuje klesající podíl zastaralých prohlížečů jako jsou Internet Explorer 6 a 7 [30]. Počet prohlížečů, ve kterých je nutné testovat, byl ale zvýšen rozmachem smartphonů a tabletů. S nimi přichází i větší rozmanitost ve velikostech displejů. Uspokojení potřeb uživatelů napříč různými zařízeními řeší populární přístup nazvaný Responsive Web Design [22]. Podporu napříč prohlížeči částečně ulehčují frameworky jako Bootstrap [1], ale testování a ladění kódu v různých prohlížečích se vyhnout nedá. Jelikož jde o těžko řešitelný problém (fragmentace trhu webových prohlížečů se nelze zbavit a monopol jednoho prohlížeče by zas představoval jiné problémy), ve své diplomové práci se budu zabývat jinou oblastí, jak ulehčit vývoj webových aplikací.

1.3 Reakce na změny

Pokud odhlédneme od těchto specifik webového vývoje zmíněných v této kapitole a v úvodu, jedná se o software jako každý jiný. Primárně v něm jde o dodávání hodnoty všem zainteresovaným stranám (stakeholderům).

Zdali bude projekt úspěšně dokončen se rozhoduje ve všech jeho fázích. Již při sběru požadavků a jejich analýze je třeba se neustále ujišťovat, zdali zákazníka správně chápeme a chystáme se mu dodat opravdu to, co potřebuje. Při samotném vývoji i přesto velice pravděpodobně nastane v požadavcích změna a je potřeba se na to připravit.

Důvody pro změny požadavků jsou různorodé. Může se jednat o externí faktory jako je situace na trhu nebo změny v zákonech; interní faktory kvůli tomu, jak se mění zákazníkův byznys nebo struktura firmy; technické důvody v momentě kdy zjistíme, že musíme zavést novou či jinou technologii pro splnění požadavků, protože ta původně vybraná se ukázala jako nevhodná; změny na základě získaných zkušeností – během vývoje se dozvídáme nové skutečnosti o doméně aplikace, na jejichž základě se můžeme rozhodnout invalidovat původní požadavky [21].

Proto je potřeba zajistit, aby software nejen splňoval požadavky a obsahoval co nejméně chyb, ale umožňoval svojí flexibilitou i snadnou údržbu a přidávání nové funkcionality.

1.4 Doménový model

Požadavky se nejsnáze promítají do tzv. doménového modelu [17]. Ten představuje množinu objektů, jejich atributů a vztahů mezi objekty mapujících data a procesy z problémové domény.

Úspěšný doménový model dodržuje principy objektově orientovaného návrhu pro zachování udržovatelnosti. Jejich účelem je podpořit znovupoužitelnost a rozšiřitelnost návrhu a jeho odolnost vůči chybám. Formuloval je Robert C. Martin [23]:

Princip jedné odpovědnosti
: *Třída má mít jedinou zodpovědnost.* Pokud při popisu toho, co má třída na starosti, použijeme spojku „a“, porušili jsme tento princip, který má vést ke znovupoužitelnosti a přehlednosti díky vysoké granularitě. Čím jednodušší třída, tím snáze se dá pochopit její účel a snáze se dá znovupoužívat v různých kontextech.

Princip otevřenosti a uzavřenosti
: *Softwarové entity (třídy, moduly, funkce atd.) by měly být otevřeny pro rozšiřování, ale uzavřeny pro modifikaci.* Při přidávání funkcionality by nemělo být nutné upravovat existující kód.

Substituční princip Liskovové
: *Objekty by měly být nahraditelné instancemi svých subtypů bez ovlivnění správnosti programu.* Častá chyba při používání dědičnosti. Potomek v hierarchii tříd nesmí rozbít funkcionalitu svého předka, protože ho lze vložit tam, kde je funkční předek očekáván (princip polymorfismu).

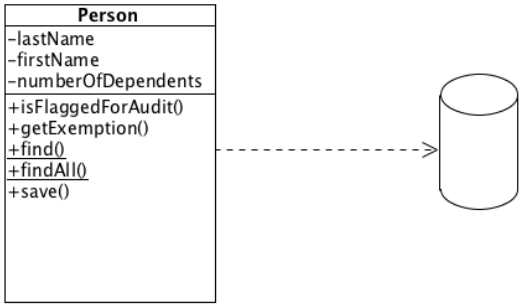
Princip oddělení rozhraní
: *Více specifických rozhraní je lepších, než jedno univerzální.* Klienti by neměli být závislí na metodách, které nepoužívají. Pokud se stává pravidlem, že třídy z nějakého rozhraní používají pouze podmnožinu metod, mělo by toto rozhraní být rozděleno.

Princip obrácení závislostí
: *Abstrakce by neměla záviset na konkrétní implementaci. Konkrétní implementace by měla záviset na abstrakci.* Usnadňuje zaměňování implementací a znovupoužitelnost.

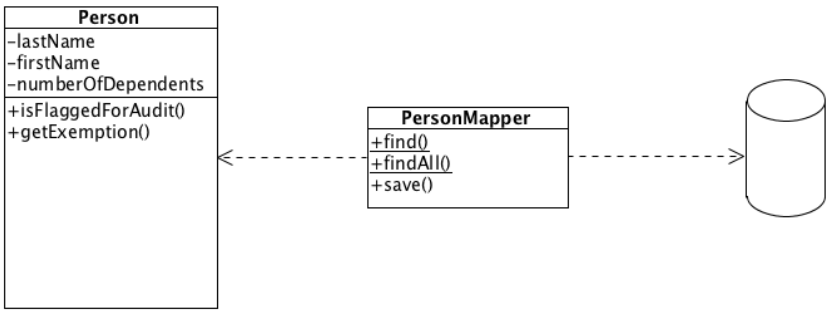
1.5 Komunikace s databází

Data v doménovém modelu zpravidla podléhají persistenci – uchování pro pozdější použití. Pro ukládání objektů do relačních databází existují dva přední návrhové vzory: Active Record a Data Mapper.

Obrázek 1.1: Active Record



Obrázek 1.2: Data Mapper



Active Record je reprezentovaný objektem, který představuje jeden řádek z databázové tabulky. Jednotlivé atributy objektu reprezentují jednotlivé sloupce tabulky. Objekt zapouzřuje přístup k databázi (výběr, ukládání, mazání) a doménovou logiku. Tím porušuje princip jedné odpovědnosti a způsobuje i další problémy [24].

Data Mapper poskytuje větší flexibilitu tím, že odděluje objektovou reprezentaci záznamu a databázové operace do různých tříd, potažmo vrstev. Doménové objekty tak ani neví o existenci persistence, tudíž mohou být znovupoužívány k různým účelům, případně k persistování různými způsoby. Data Mapper je využíván populárními ORM frameworky jako Doctrine a Hibernate.

Alternativou k ORM je reprezentace dat v aplikaci pomocí obecných schránek na data, např. asociativním polem. Výhodou tohoto přístupu je, že můžeme z databáze vybírat libovolné množiny sloupců bez nutnosti ke každé variantě psát třídu a nastavovat mapování. V rozhraních metod pak ale bez toho nelze vynutit konkrétní typ databázového záznamu nebo výsledku dotazu. Nemůžeme se spoléhat na uživatele metody, že předal takový argument, s jakým počítáme a vlastně kromě textového komentáře ani neexistuje způsob, jak tento předpoklad vyjádřit. V komplexnějších aplikacích tento způsob tedy nejde použít.

Namísto zavržení objektů v aplikaci bychom je mohli ukládat do vhodnějšího typu databáze než je relační. Nebylo by tak třeba vytvářet mapování

mezi objektovým a relačním světem, objekty by se ukládaly ve své původní struktuře. Objektové databáze nejsou bohužel zdaleka tak populární jako relační. Ačkoli usnadňují práci programátorům snadným ukládáním objektů, nejsou příliš kompatibilní se zbytkem světa, který uvažuje relačně, v tabulkách. Objektovým databázím chybí standard v podobě dotazovacího jazyka [32]. S tím souvisí i nedostatek široce použitelných nástrojů pro práci s daty mimo aplikaci. Relační data lze importovat do tabulkového procesoru jako je Microsoft Excel a pracovat s nimi bez znalosti programování. Pokud chcete pracovat s daty uloženými v objektové databázi, musíte být programátor. Další problém představují změny objektového schématu, např. přidání atributu. V přístupu do objektové databáze je potřeba zajistit zpětnou kompatibilitu (považovat nový atribut za nepovinný), nebo všechny uložené objekty upravovaného typu přeuložit, což bude při větším objemu dat časově náročné. Kvůli těmto problémům jsou tedy relační databáze stále nejpoužívanější, i přes práci s mapováním navíc.

1.6 Normalizované schéma databáze a duplicitní kód

Normalizace relační databáze je proces reorganizace tabulek a sloupců za účelem minimalizace redundance dat. Normální formy jsou soubor pravidel, jak normalizace dosáhnout [37]. Splnění vyšší formy předpokládá splnění i všech nižších forem.

První normální forma
: Každý atribut obsahuje pouze atomické hodnoty.

Druhá normální forma
: Každý neklíčový atribut je plně závislý na primárním klíči.

Třetí normální forma
: Všechny neklíčové atributy musí být vzájemně nezávislé.

Existují i další normální formy, ale splněním té třetí jsou obvykle splněny i ty další, což je činí, ironicky, redundantními.

Primární motivací pro normalizaci je zabránění aktualizačních anomálií. Stejně jako při eliminaci duplikace v kódu, tak i v databázi těžíme z toho, že je v ní každá informace uložena právě jednou. Pokud tuto informaci aktualizujeme, vystačíme si právě s jedním dotazem, nemohou vzniknout žádné inkonzistence. Další motivací je čistě praktická úspora diskového a paměťového prostoru.

Někdy ale může být opodstatněný i požadavek na denormalizaci. Pokud často spojuji data z různých tabulek pomocí klíčového slova `JOIN` a tyto tabulky obsahují velké množství dat, mohu těžit z denormalizace za účelem zvýšení výkonu.

Dalším důvodem může být účelná aktualizační anomálie pro zachování historických dat. Pokud máme kupříkladu fakturační systém, který obsahuje tabulku kontaktů a tabulku faktur, nechceme změnou kontaktních údajů ovlivňovat již vystavené faktury. Kontaktní údaje dodavatelů a odběratelů jsou v momentě vytváření faktury zkopírovány do řádku s fakturou, protože je rozhodující jejich podoba právě v tomto momentě a pozdější změny na ni nesmí mít vliv. To samé platí třeba pro platební systémy, kdy rozhoduje hodnota zboží v momentě objednání a pozdější úpravy cen nesmí mít na historii objednávek vliv.

Oblast, kterou jsem se rozhodl v této práci zabývat, se také týká jednoho specifického využití denormalizace ve spolupráci s aplikací využívající ORM. Mým cílem je eliminovat potřebu této denormalizace a zjednodušit tak datovou strukturu databáze i kód aplikace.

Některé typy entit (objekty reprezentující záznam z databáze) mohou nabývat stavů, které ovlivňují jejich význam v aplikaci a množinu operací, jež s nimi lze provádět. Tento stav je odvoditelný z hodnoty v jednom sloupci, z kombinace hodnot ve více sloupcích nebo z navázaných záznamů přes cizí klíče. Např. platná zaplacená rezervace na vstupenkovém portálu bude mít vyplněné datum zaplacení a zároveň nevyplněný důvod zrušení. Spárovaná bankovní transakce s rezervací bude obsahovat cizí klíč na propojenou rezervaci.

Pokud potřebujeme na základě takového kritéria vybírat záznamy z databáze, musíme ho vyjádřit v jazyce SQL. Pokud ale zároveň potřebujeme zjistit stav nějaké entity v aplikaci, musíme ho vyjádřit i v programovacím jazyce aplikace. Tím vzniká duplikace – tu samou informaci máme vyjádřenou dvěma způsoby.

Jaká se nabízí řešení? Reprezentace v SQL bychom se mohli zbavit tak, že z databáze vybereme všechny záznamy a vyfiltrujeme je na straně aplikace. Toto je velmi neefektivní způsob, který zatíží spojení mezi aplikací a databází množstvím přenesených dat. Když už aplikace všechna data získá, nedokáže je filtrovat tak efektivně jako databáze s vybudovanými indexy, které s tímto úkolem mají pomoci.

Další řešení je redundantní sloupec pro zkoumané kritérium. Jakmile entita přejde do nějakého stavu, tuto skutečnost zaznamenáme do zvláštního sloupce (viz zdrojový kód 1.1 a tabulka 1.1). Filtrování na straně databáze i aplikace se pak zjednoduší na porovnání jedné hodnoty, což je dostatečně jednoduché na to, abychom to nepovažovali za duplikovaný kód a možný zdroj chyb.

Zdrojový kód 1.1: Udržování informace o stavu entity v redundantním sloupci.

```
public function pay()
{
    $this->paidDate = new DateTime('now');
    $this->status = StatusEnum::PAID;
}
```

Tabulka 1.1: Redundantní sloupec status určující stav záznamu

id	status	paidDate	cancellationReason
1	unpaid	NULL	NULL
2	paid	2014-04-18	NULL
3	cancelled	NULL	manual
4	cancelled	2014-04-20	manual

Vyvstává s tím ale problém s konzistencí dat na straně databáze. Pokud dojde ke změně stavu bez aktualizace tohoto sloupce, bude v něm nepravdivá informace a entita bude považována za něco, co ve skutečnosti není. Zajištění konzistence dat je běžně jen otázkou pozornosti toho, kdo data upravuje, avšak ne vždy se stav entity mění na základě uživatelské nebo systémové akce. Rozhodnout mohou i externí vlivy, typicky čas. Pokud např. čeká rezervace na zaplacení a zákazník na toto zaplacení má 15 minut, po jejich uplynutí nelze nijak automaticky status aktualizovat. Ano, mohli bychom nasadit pravidelně spouštěný skript pomocí cronu, ale v databázi by se mezi skutečným vypršením a spuštěním aktualizačního skriptu vyskytovaly nekonzistentní záznamy. Šlo by o případ tzv. eventual consistency, která může být nežádoucí.

Řešení, které chci naimplementovat, by mělo řešit všechny tyto problémy – eliminovat redundanci v databázi, eliminovat duplicity v kódu aplikace a umět správně vyhodnocovat i stavy měnící se nezávisle na běhu aplikace.

Kapitola 2

Analýza problému

2.1 OOP versus SQL

Objektově orientované programování a SQL jsou dvě různá programovací paradigmata. OOP je prostředek, jak modelovat a strukturovat řešení problému v univerzálních programovacích jazycích. SQL slouží k definici a úpravě schémat a k výběru a úpravě dat z relačních databází. Mým cílem není umožnit transformovat mezi těmito platformami 100 % kódu, ale pouze tu část, u které často dochází k duplikaci logiky – pokud potřebuji podle stejných kritérií hodnotit objekt v paměti aplikace a databázový záznam, musím nyní manuálně daný hodnotící výraz přepsat a udržovat v jazycích obou platforem. Alternativní řešení, která si ale neporadí v některých situacích, jsou rozebírána v předchozí kapitole 1.6.

2.2 Jazyk PHP

Pro vývoj webových aplikací používám PHP. Ačkoli jej doprovází řada problémů [25] daných historickým vývojem, kdy PHP z počátku sloužilo pro tvorbu jednoduchých dynamických webů (zkratka PHP původně znamenala „Personal Home Page“), v posledních letech se z něj stala platforma srovnatelná s jinými platformami pro vývoj webových aplikací. Velkou měrou k tomu přispěla verze 5.3 vydaná v červnu 2009 [13], která přinesla podporu jmenných prostorů, anonymních funkcí a late static bindings. Za svou popularitu vděčí své jednoduchosti a přístupnosti pro začátečníky, syntaxi podobné jazyku C a „shared nothing“ architektuře. Každý PHP proces je jednovláknový a každý HTTP požadavek je vykonáván zcela izolovaně od ostatních, což eliminuje komplexní problémy se souběhy a synchronizací. Nyní PHP celosvětově pohání 81,9 % webů, u kterých je známý použitý programovací jazyk [36].

Nedostatky PHP a repetetivní úkoly vývojářů řeší populární a rozšířené open-source frameworky. Mezi jejich zástupce patří např. Nette, Symfony, Doctrine nebo Guzzle.

Aktuální hlavní vývojovou řadou PHP je 5.5.

2.3 Doctrine 2 ORM a jazyk DQL

ORM frameworky umožňují za uživatele generovat SQL dotazy. Ty ale mohou trpět neefektivitou, protože framework v době vybírání vyžádaných záznamů a jejich transformování do objektů neví, co s nimi uživatel zamýšlí udělat. Mezi znaky neefektivity patří vybírání hodnot ze sloupců, které se následně v aplikaci vůbec nevyužijí, a provádění lazy loadingu uvnitř cyklu, známé také jako N+1 problém [14].

Zdrojový kód 2.1: Ukázka N+1 problému. Každá iterace si z databáze stáhne autora aktuálního článku.

```
$articles = $entityManager->getRepository('Article')->findAll();
foreach ($articles as $article) {
    echo $article->getAuthor()->getName();
}
```

Řešením je umožnit uživateli vyjádřit svůj úmysl, aby se všechna data, se kterými bude pracovat, z databáze stáhla optimálně. K tomu je ideální jazyk SQL, ale v kontextu s ORM ho nelze využít, protože bychom pro každý SQL dotaz museli manuálně konfigurovat mapování jeho výsledku na objekty. Z toho důvodu existují jazyky jako je DQL, JPQL nebo HQL. Vypadají obdobně jako SQL, ale namísto tabulek a sloupců se v nich odkazuje na entitní třídy a jejich atributy. Díky tomu, že při psaní dotazu pracujeme s touto úrovní abstrakce, si framework mapování automaticky zařídí sám. Výsledné SQL je DQL dotazu velmi podobné.

Zdrojový kód 2.2: Řešení N+1 problému z ukázky 2.1 za pomoci DQL. Články i jejich autoři se stáhnou jedním SQL dotazem.

```
$articles = $entityManager->createQuery('
    SELECT ar, au FROM Article ar
    JOIN ar.author au
')->getResult();
foreach ($articles as $article) {
    echo $article->getAuthor()->getName();
}
```

DQL je tak na půli cesty mezi SQL a OOP, ale neumožňuje nám stále zajistit kód bez duplikace. Jelikož ale v navrhovaném frameworku budeme chtít pracovat s objekty a jejich atributy, využijeme ho jako prostředníka pro tvorbu výsledných SQL dotazů.

2.4 Systémy na správu byznys pravidel

Systémy pro business pravidla (angl. *business rules engines*) představují alternativní výpočetní model ke klasickému imperativnímu modelu [18]. Poskytují soubor izolovaných pravidel vyjadřujících, co se má vykonat pro splnění byznys požadavků. Každé pravidlo se skládá ze dvou částí – podmínky a akce. Pokud se daná podmínka vyhodnotí jako splněná, vykoná se přiřazená akce.

Zdrojový kód 2.3: Příklady byznys pravidel

```
if car.owner.hasCellPhone
       then premium += 100;
if car.model.theftRating > 4
       then premium += 200;
if car.owner.livesInDodgyArea && car.model.theftRating > 2
       then premium += 300;
```

Pro splnění stanovených cílů v kapitole 1.6 se budu zabývat jen částí tradičních byznys pravidel – podmínkami. Potřebuji najít takový zápis, který bude dostatečně intuitivní a umožní mi jejich transformaci do různých prostředí, kterou také naimplementuji. Vyvolávání akcí na základě vyhodnocení těchto podmínek nebude v kompetenci tohoto frameworku. V tom kontextu, v jakém chci framework aplikovat, by to ani příliš nedávalo smysl a zužovalo možnosti jeho použití. Pod akcí si v relační databázi lze představit příkaz `UPDATE` či `DELETE`, ale podmínky přeložené do jazyka SQL lze aplikovat mnoha různými způsoby - v příkazu `SELECT` jako vybírané hodnoty, kritéria agregačních funkcí, omezení v `JOIN` i jako tradiční podmínky v klauzulích `WHERE` a `HAVING`; v integritních omezeních pomocí DDL příkazu `CHECK` a v uložených procedurách. V serverové aplikaci pak bude vyhodnocené podmínky možné využívat taktéž libovolným způsobem a dokonce na jejich základě vybudovat i obecný systém pro byznys pravidla – stačí realizovat konfiguraci, která prováže podmínky s definovanými akcemi.

2.5 Podoba pravidel

Je třeba zvolit takový zápis, který nejenže umožní vyjádřit účel pravidla, ale i pohodlnou a efektivní transformaci do jazyka SQL a vykonání v prostředí PHP. Nabízí se následující varianty [38].

2.6 Programovací jazyk

Programovací jazyk je univerzální prostředek pro vyjádření jakéhokoli algoritmu. Pokud bychom dovolili zapisovat byznys pravidla pro účel vykonávání ve více různých prostředích v programovacím jazyce (v našem případě v PHP), bylo by velmi těžké až nemožné tato pravidla plnohodnotně převádět. Mohli

Zdrojový kód 2.4: Jak by vypadalo byznys pravidlo zapsané v PHP

```
class ReservationIsPaid implements Rule
{

    public function rule(Reservation $r)
    {
        return $r->isPaid()
            && $r->getCancellationReason() === NULL;
    }

}
```

bychom je sice dovolit zapisovat do `.php` souborů s tím, že budeme podporovat jen takovou podmnožinu, kterou umíme převádět, ale nic bychom tím nezískali. Strukturování jednotlivých pravidel by vyžadovalo příliš mnoho šumu (viz ukázka 2.4) a tento způsob by vytvářel iluzi, že píšeme přímo vykonávaný PHP kód. Přitom bychom ho psali za účelem inspekce. Zároveň bychom si tím odstřihli možnost vlastního DSL, protože kód by svou syntaxí i sémantikou musel odpovídat PHP.

2.7 Meta-instrukce programovacího jazyka

PHP nemá integrovanou podporu pro meta-instrukce jazyka jako jsou anotace. Komunita sice anotace široce využívá pomocí jejich zápisu do dokumentačních komentářů, neexistuje ale žádný standard pro jejich definici, zápis a čtení.

Použití anotací znamená svázat je s konkrétní jednotkou zdrojového kódu – funkcí, třídou, atributem nebo metodou. Jelikož jsme zápis v podobě zdrojového kódu zavrhli a anotace slouží jen jako jeho doplněk, nemáme pro ně žádné využití.

2.8 Expression language

Jedná se o jazyk podobný obecnému programovacímu jazyku, ale možnost jeho uplatnění je uměle omezená. Nalezl využití například v technologiích prezentačních vrstev, jako jsou JavaServer Pages, JavaServer Faces [38], šablonovací systém Twig nebo javascriptový framework Angular. Slouží v těchto případech k zápisu krátkých výrazů, které ovlivňují chování komponent uživatelského rozhraní, ale nemusí být nutně svázaný jen s ním. Typické operace, které expression language podporuje, jsou:

Zdrojový kód 2.5: Ukázka anotací Doctrine 2 ORM v PHP

```
/**
 * @Entity
 */
class Article
{

    /**
     * @Id
     * @Column(type="integer")
     * @GeneratedValue(strategy="AUTO")
     */
    private $id;

}
```

- Odkazování se na hodnoty proměnných
- Odkazování se na literály (přímo zapsané booleovské, číselné a řetězcové hodnoty)
- Přístup k atributům objektu
- Vyvolávání a získávání návratových hodnot z funkcí a metod
- Matematické operace
- Vyhodnocování a řetězení logických operátorů
- Porovnávání

Co naopak neumí:

- Definovat nové funkce, třídy a metody
- Řídicí struktury – větvení a cykly s výjimkou ternárního operátoru, který umožňuje mít ve svých větvích pouze jednoduché výrazy
- Sekvenční zřetězení výrazů

Tyto vlastnosti ho činí ideálním kandidátem pro zápis pravidel za účelem transformace do relační databáze, protože výčet podporovaných vlastností odpovídá tomu, co potřebujeme na obou platformách vyjádřit. Zapsané výrazy bude třeba analyzovat a vytvářet z nich syntaktický strom, jehož jednotlivé uzly budou převáděny do syntaxe cílových platforem.

2.9 Doménově specifický jazyk

Doménově specifické jazyky (zkr. angl. *DSL*) se oproti univerzálním programovacím jazykům vymezují svou specializací na nějakou konkrétní doménu. Dělí na dva druhy: interní a externí [16]. Interní využívají vyjadřovací prostředky univerzálního jazyka k dosažení specifického cíle a mívají podobu netypicky vypadajícího rozhraní metod využívajícího tzv. fluent interface. Externí DSL uplatňují vlastní syntax a potřebují proto vlastní parser. Mezi příklady patří CSS, SQL, Makefile nebo regulární výrazy.

Externí DSL v případě této práce nalezne využití v kombinaci s expression language v jednotlivých uzlech transformovaných výrazů.

Zdrojový kód 2.6: Ukázka interního doménově specifického jazyka – konfigurace mock objectu v PHPUnitu

```
$maker = $this->getMock('ReservationMaker');
$maker->expects($this->exactly(2))
    ->method('makeReservation')
    ->with($this->identicalTo($occupiedSeat))
    ->will($this->throwException(
        new ReservationCouldNotBeMadeException()
    ));
```

2.10 Funkční požadavky

Framework umožní:

- Zápis hodnotících výrazů (pravidel) ve vlastním expression language a DSL.
- Vykonávání pravidel nad skalárními hodnotami a objekty v jazyce PHP.
- Transformaci pravidel do jazyka SQL relačních databází prostřednictvím DQL frameworku Doctrine 2 ORM. Kromě SELECT/UPDATE/DELETE dotazů bude možné transformovaná pravidla využít i v integritních omezeních.
- Hledat v pravidlech chyby, aniž by je bylo potřeba spouštět – pomocí statické analýzy.
- Kompilaci pravidel pro optimální vykonávání v PHP (aby při každém požadavku nemuselo docházet ke čtení souboru s pravidly, jejich parsování a sestavování syntaktického stromu).

2.11 Nefunkční požadavky

- Framework bude implementovaný v jazyce PHP a bude podporovat verzi 5.5 a novější.
- Framework bude otestovaný jednotkovými a integračními testy.
- Framework bude podporovat integraci do projektu pomocí nástroje Composer pro správu závislostí.
- Framework bude ctít princip otevřenosti a uzavřenosti. Implementace transformace pravidel do dalších prostředí nebude vyžadovat modifikaci existujících částí frameworku.

Kapitola 3

Návrh frameworku

3.1 YAML

Specifikace formátu YAML [15] popisuje způsob, jak psát textové dokumenty za účelem převodu do datových struktur (skalárních hodnot a klasických a asociativních polí) programovacího jazyka. Oproti XML a JSON používá stručnější a lépe lidsky čitelnou syntaxi[1]. Z tohoto důvodu jsem se ho rozhodl použít pro reprezentaci pravidel. K parsování formátu použiji komponentu frameworku Symfony YAML [11].

3.2 Zvolená syntaxe pravidel

Pravidla budou mít podobu jednořádkových výrazů. Při výběru konkrétní syntaxe výrazů bych mohl zvolit existující expression language, nebo vlastní. Existující by měl tu výhodu, že není neznámý a uměla by ho už nějaká skupina vývojářů používat. Také by měl funkční implementaci, takže bych si ušetřil práci s vlastním parserem. Nabízí se použít Symfony Expression Language [9], který vznikl oddělením ze šablonovacího systému Twig. V představujícím článku autor dokonce zmiňuje „The expression language component is a perfect candidate for the foundation of a business rule engine.“ [29] Bohužel při bližším ohledání jsem zjistil, že neumožňuje přístup k vnitřní reprezentaci výrazu a tudíž ho lze použít pouze k vyhodnocování v PHP, nikoli transformaci do SQL.

Před tím, než bych se býval pustil do vlastního expression language, jsem se zamyslel, jak by se ještě dalo ušetřit si implementaci parseru a nezahodit zkušenosti vývojářů s existujícími technologiemi. Došel jsem k tomu, že mohu využít syntaxi samotného PHP, ale omezit ji na úroveň expression language – zakázat řídicí struktury a více výrazů oddělených středníkem. Díky tomu

[1] Ke strukturování používá odsazování bloků mezerami.

Zdrojový kód 3.1: Výsledek tokenizace výrazu `<?php echo strtoupper('Hello world!');` pomocí funkce token_get_all()

```
[
        [T_OPEN_TAG, "<?php "],
        [T_ECHO, "echo"],
        [T_WHITESPACE, " "],
        [T_STRING, "strtoupper"],
        "(",
        [T_CONSTANT_ENCAPSED_STRING, "'Hello world!'"],
        ")",
        ";"
]
```

Zdrojový kód 3.2: Výsledek parsování výrazu `<?php echo strtoupper('Hello world!');` pomocí knihovny PHP Parser

```
[
        PhpParser\Node\Stmt\Echo_: [
                PhpParser\Node\Expr\FuncCall: [
                        name: "strtoupper",
                        args: [
                                PhpParser\Node\Scalar\String: [
                                        value: "Hello world!"
                                ]
                        ]
                ]
        ]
]
```

nebude zvolená syntaxe úplně neznámá. Některé jazykové konstrukty budu ale interpretovat po svém, *sémantika* jazyka [33] bude tedy odlišná.

Tokenizaci PHP kódu lze provádět prostřednictvím interní funkce token_get_all(). Tokenizace je proces, během kterého tokenizer parsuje vstupní text, rozpoznává v něm klíčová slova a další symboly jazyka a text podle nich rozdělí. Výsledkem je ploché pole (viz ukázka 3.1).

Sekvenční výčet tokenů mi ovšem pro inspekci kódu nestačí. Bez hierarchie nelze bez další práce poznat význam spouštěného kódu. Existuje ovšem knihovna PHP Parser [27], která z kódu dokáže vytvořit syntaktický strom, tedy hierarchickou strukturu, se kterou se už bude pracovat lépe (viz ukázka 3.2).

Zdrojový kód 3.3: Ukázka definice dvou pravidel v souboru formátu YAML

```
reservation_is_cancelled:
  arguments:
    r: Amphibian\Reservation
  rule: $r->cancellationReason !== NULL
reservation_is_paid:
  arguments:
    r: Amphibian\Reservation
    is_cancelled: @reservation_is_cancelled
  rule: $r->paidDate !== NULL && !$is_cancelled
```

3.3 Argumenty pravidel

Vedle samotného pravidla potřebuji definovat vstupní argumenty pro hledání chyb pomocí statické analýzy – především kontrolu existence a viditelnosti metod a atributů. Lze to přirovnat k definici vstupních argumentů metody. Framework bude podporovat čtyři typy argumentů:

Skalární `boolean`, `string`, `integer` a `float`

Odkazující se na jiná pravidla Např. `@reservation_is_cancelled`

Objektové V definici argumentu musí být název existující třídy

Kolekce objektů Název třídy následovaný `[]` (syntaxe pro pole)

Klíč nejvyšší úrovně představuje název pravidla. Každé pravidlo pak minimálně obsahuje klíč `rule` s vyhodnocovaným a transformovaným výrazem. V případě, že pravidlo přijímá argumenty zvenku, pak musí obsahovat i pole `arguments`. Klíč v tomto poli představuje proměnnou, kterou je argument reprezentován ve výrazu (např. `$foo`), hodnota klíče pak představuje typ argumentu. Ty jsou popsány výše.

V žádném typu argumentů neumožňuji předávat `NULL` hodnoty, které jsou častým zdrojem chyb.

3.4 Přístup k atributům objektu

Neumožňuji volat metody, protože je to nepřenositelné do SQL, ale umožňuji se odkazovat na atributy objektu pomocí `->` jako v PHP. Tyto atributy ORM knihovna mapuje na sloupce v databázi. Pokud je atribut `private` nebo `protected`, nelze k němu přistoupit zvenku v PHP přímo. Před vyhozením výjimky o nepřístupném atributu ještě zkontroluji, zdali neexistuje k danému atributu veřejný getter – metoda s názvem vytvořeným podle předpisu `'get'`

`. ucfirst($propertyName)`. Funkce `ucfirst` změní první písmeno řetězce na velké. Toto je první odlišení od sémantiky PHP, které tuto funkcionalitu neobsahuje. Implementuje ji ale např. třída Object z frameworku Nette, která slouží jako předek všech objektů [4].

3.5 Porovnávání a logické operátory

Framework bude podporovat matematické porovnávací operace `<`, `<=`, `>` a `>=`. Kromě čísel je lze využít i pro interní objekty PHP, u kterých lze přetěžovat význam operátorů, např. DateTime. Přetěžování operátorů není povoleno pro vlastní objekty.

PHP umožňuje dva způsoby porovnávání rovnosti – „nepřesné“ (`==` a `!=`) a „striktní“ (`===` a `!==`). Nepřesné se vyznačuje tím, že u skalárů před porovnáním hodnoty přetypuje, což je velmi nebezpečné. Pokud budu porovnávat např. řetězec a číslo, řetězec se nejdříve přetypuje na číselný typ. Pokud řetězec začíná číslem, použije se toto číslo, pokud ne, přetypuje se na 0. Při nepřesném porovnání se tedy všechny řetězce nezačínající číslem rovnají 0 [8] [7]. V případě objektů se porovnává obsah jejich atributů, což porušuje zapouzdřenost. Toto chování PHP může být zdrojem mnoha chyb, protože není na první pohled zřejmé, co se děje. Nepřesné porovnání by se mělo používat jen výjimečně a to v momentě, že je to opravdu to, co potřebujeme.

Bezpečnější je upřednostnit a používat striktní porovnávání. To se vyhodnotí kladně jen v případě, že porovnáváme hodnoty stejných typů a se stejným obsahem. U objektů tehdy, pokud na levé i pravé straně porovnání máme tu samou instanci.

Pro snížení rizika chyb podporuji v pravidlech pouze striktní porovnávání.

3.6 Odkazování se na ostatní pravidla

Byznys pravidla jsou důležitou součástí aplikace a je potřeba na ně klást stejné nároky jako na zbytek kódu. Jestliže má pravidlo vzniknout složením již existujících pravidel, je nepřípustné to realizovat zkopírováním výrazů, protože tak dochází k duplikaci a hrozí desynchronizace a nekonzistentní chování, stejně jako v běžném kódu programovacího jazyka.

Statickou analýzou bych sice vytvořením certifikátů[1] pro všechny podvýrazy a výrazy mohl zkopírovaná pravidla detekovat a upozorňovat na ně, ale jelikož je zkopírování poměrně začátečnickou chybou a u pokročilých programátorů k němu nedochází, tak by tato detekce přinášela spíše falešné poplachy a odhalovala oprávněné případy duplikace, kdy se dva výrazy shodují dílem náhody. Tedy tehdy, kdy jejich desynchronizace nepřinese žádné nežádoucí

[1] Certifikáty se používají pro detekci grafového izomorfismu. Jde o takovou reprezentaci datové struktury, která bude shodná pro každé dvě instance, které považujeme za shodné a rozdílná pro každé dvě instance, které považujeme za rozdílné [34].

účinky, protože spolu nemají nic společného. Odstranění takové duplikace by vedlo k nežádoucímu provázání.

Odkazování se na existující pravidla je realizováno pomocí syntaxe `@nazev_pravidla`, jak lze vidět v ukázce 3.3 u argumentu `is_cancelled`. Na pravidlo se odkazuje stejně jako na ostatní argumenty pomocí proměnné. Oproti ostatním typům argumentů je ale při vyhodnocování třeba do odkazovaných pravidel předávat argumenty. Implicitně se předávají všechny pod stejným názvem. Pravidlo `@reservation_is_cancelled` lze uvnitř `@reservation_is_paid` využít díky tomu, že se obě týkají rezervace a v argumentech obou pravidel se vyskytuje pod názvem `r`.

V případě, že chceme využít pravidlo, které není takto kompatibilní (má argumenty s jinými názvy), je třeba upřesnit překlad těchto argumentů. K tomu jsem využil syntaxi inspirovanou pojmenovanými argumenty v Pythonu[1]. Namísto pouhé zmínky o proměnné jsem použil syntaxi pro vyvolání funkce uložené v proměnné. Jako argumenty této funkci předávám pole s předpisem, jaké argumenty se při vyvolání mají předat pod jinými názvy. Klíč v tomto poli představuje název argumentu v odkazovaném pravidle, hodnota by měla být proměnná argumentu v aktuálním pravidle. Zbylé argumenty se předávají implicitně.

Zdrojový kód 3.4: Jak vypadá volání pravidla s nekompatibilními názvy argumentů. Definice `reservation_is_cancelled` je stejná jako v 3.3.

```
reservation_is_paid:
  arguments:
    reservation: Amphibian\Reservation
    is_cancelled: @reservation_is_cancelled
  rule: $reservation->paidDate !== NULL &&
        !$is_cancelled([r => $reservation])
```

3.7 Agregační funkce

V relačních databázích jsou často využívány agregační funkce – vezmou hodnoty z každého řádku vraceného výsledku a na základě nich vypočtou jedno výsledné číslo. V Amphibianu je v obdobné syntaxi podporuji i pro vyhodnocování v PHP. V argumentu volané funkce se smí vyskytnout právě jeden kolekční argument pravidla, protože má speciální význam. Při vyhodnocování iteruji nad poskytnutou kolekcí. V každé iteraci vezmu daný prvek kolekce a vložím ho do výrazu uvnitř agregační funkce pod názvem kolekčního argumentu. Tím napodobuji syntaxi a princip SQL. Jedná se o nejvýraznější prvek doménově specifického jazyka frameworku.

[1] Python umožňuje volat funkce s parametry nejen na základě jejich pořadí v definici funkce, ale také v libovolném pořadí, pokud se u předaného parametru upřesní jeho jméno [26].

Zdrojový kód 3.5: Využití agregační funkce pro sečtení celkové ceny rezervací

```
sum_reservation_price:
  arguments:
    r: Amphibian\Reservation[]
  rule: sum($r->price)
```

Podporované agregační funkce jsou `count`, `sum`, `avg`, `min` a `max`. Při vyhodnocování v PHP mají všechny časovou komplexitu $\Theta(n)$.

Část II

Implementace

Kapitola 4

Objektová reprezentace pravidla

Při implementaci dbám zásad kvalitního kódu a objektového návrhu. Patří mezi ně konzistentní odsazování, čitelnost, srozumitelnost a dodržování principů SOLID (viz kapitola 1.4). Reprezentace pravidla ve frameworku musí být oddělená a nezávislá na částech, které budou vyhodnocovat pravidla v PHP a transformovat je do DQL, aby byla zajištěná rozšiřitelnost i pro další využití.

4.1 Neměnnost objektů

Další zásada, kterou jsem se rozhodl ve frameworku dodržovat, je neměnnost (*immutability*) všech objektů. Ačkoli hlavní výhoda neměnnosti[1] je patrná v jiných jazycích, než je PHP, které je jednovláknové, rozhodl jsem se ji použít. Zjednodušuje kód – uživateli objektu nemusím poskytovat settery. Všechny argumenty nutné k vytvoření objektu vyžaduji v konstruktoru a pokud se jedná o value object, poskytuji k těmto datům gettery. Zařídím tak vytvoření konzistentního objektu a ušetřím si psaní a udržování setterů. Pokud posléze předám reference na neměnný objekt do více míst v systému, nemusím se obávat, že jedno z těch míst změní obsah tohoto objektu, čímž by se zaktualizovaly i všechny jeho reference. To by obzvlášť v případě pravidel bylo nežádoucí (viz kapitola 8 o odkazování se na jiná pravidla ve vztahu k transformaci do DQL).

4.2 Amphibian\Rule\Rule

Každé pravidlo zapsané v YAML souborech je v aplikaci reprezentováno objektem Amphibian\Rule\Rule. Tento objekt obsahuje čtyři atributy:

`name` Název pravidla.

[1] Neměnné objekty jsou bezpečné pro přístup z více vláken, protože po jejich vzniku už umožňují pouze čtení svého obsahu; neumožňují vlastní modifikaci a vynucují vždy vytvoření nové upravené instance.

`arguments`	Kolekce argumentů pravidel – objekt ArgumentsCollection.
`tree`	Syntaktický strom uzlů reprezentující výraz pravidla.
`source`	Řetězec s výrazem pravidla pro případ, že by bylo třeba vytvořit odvozené pravidlo (využívám toho taktéž při transformaci do DQL). PHP Parser neumožňuje klonování svých objektů.

4.3 TypedCollection

Objekt s vlastní kolekcí namísto pole používám, protože rozšiřuji jeho funkcionalitu. PHP nepodporuje generika jako Java, takže neumí v poli vynutit určitý typ hodnot. Naimplementoval jsem do jmenného prostoru Amphibian\Internal třídu TypedCollection, která obsahuje PHP pole[1], ale umožňuje do něj uložit pouze objekty specifikovaného typu. Při přístupu k neexistujícímu klíči vyhazuje výjimku namísto tzv. notice chyby, která nejde tak snad zachytávat. Před vyhozením výjimky projde existující položky v kolekci a spočítá Levenshteinovu vzdálenost jejich názvu od toho vyžádaného a do popisu výjimky přidá ty, jejichž hodnota je menší než 4. Uživateli tak může nabídnout správné názvy, pokud vyžádaný název neexistuje z důvodu překlepu.

TypedCollection je využívaná prostřednictvím kompozice v objektech typu ArgumentsCollection obsahující argumenty konkrétního pravidla a v RulesCollection obsahující všechna načtená pravidla aplikace.

4.4 Typy argumentů

Objekty odpovídají typům argumentů z kapitoly 3.3. Jde o ScalarArgument, RuleReferenceArgument, ObjectArgument a CollectionArgument. Všechny obsahují příslušné informace o daném typu argumentu, tedy typ hodnoty, popřípadě název odkazovaného pravidla. ScalarArgument navíc umí validovat hodnoty.

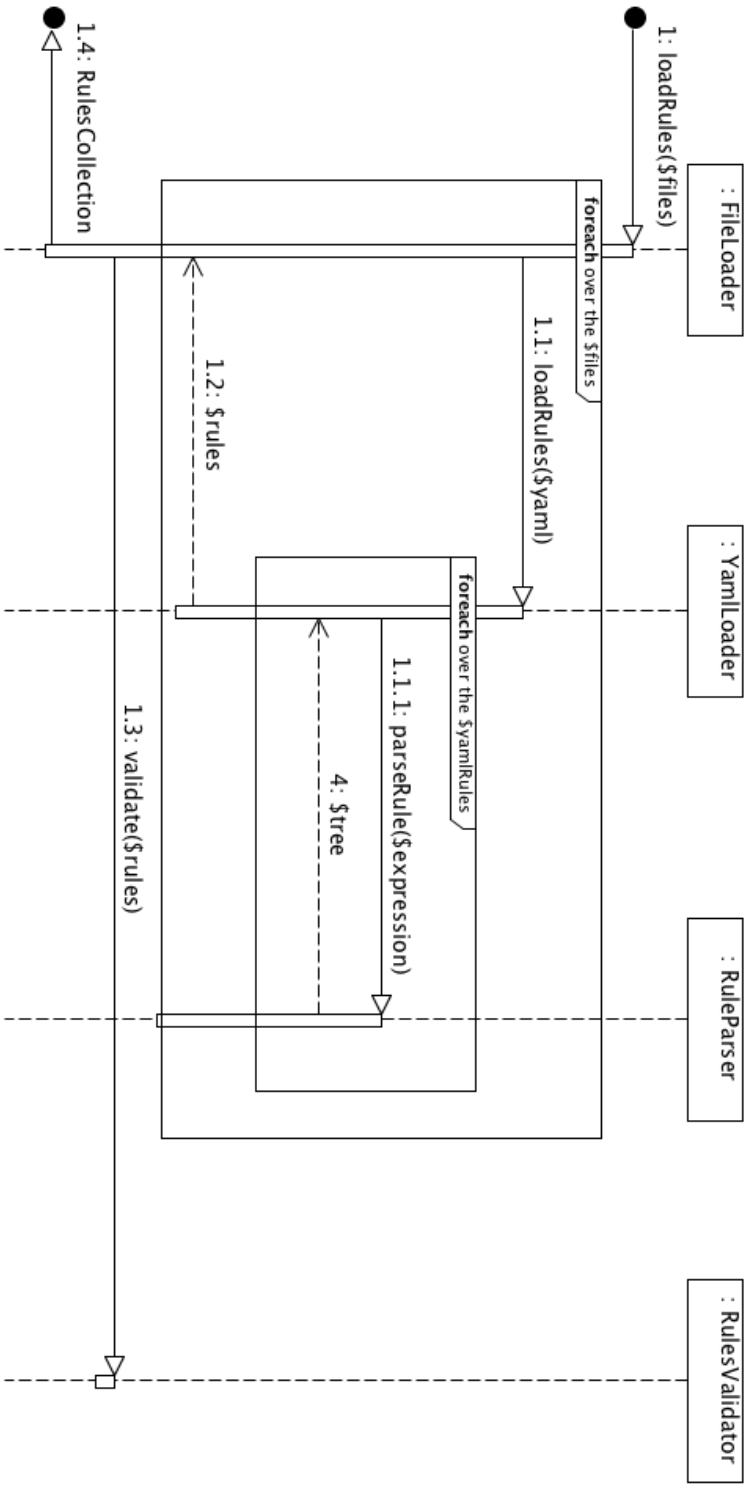
Tyto objekty neobsahují konkrétní data, ale slouží k popisu typů argumentů, které vyžaduje popisované pravidlo.

4.5 Vytvoření objektů Rule a RulesCollection

Pomocí sekvenčního diagramu 4.1 popisuji proces od načtení YAML souboru s pravidly po vytvoření objektů typu Rule a RulesCollection včetně statické analýzy prováděnou objektem RulesValidator, kterou se zabývám v následující kapitole 5.

[1]Pole má v PHP trochu jiný význam, než klasická datová struktura s tímto názvem. Při jeho zakládání není potřeba určovat velikost, není fixní. Kromě klasických číselných indexů umožňuje používat i řetězce. Interní implementace je kombinací hashovací tabulky a spojového seznamu [28].

Obrázek 4.1: Sekvenční diagram vytvoření objektů Rule a RulesCollection



4.6 FileLoader a YamlLoader

Třída FileLoader načte obsah vyžádaných souborů a řetězce ve formátu YAML předá jednotlivě třídě YamlLoader. Ta každý řetězec předloží knihovně Symfony YAML, která jí vrátí vícedimenzionální pole se skalárními hodnotami. Ty je potřeba posléze převést na objektovou reprezentaci popisovanou výše.

V kódu třídy YamlLoader jsou vyjádřena všechna specifika zápisu pravidel v YAML souborech. Pokud by v budoucnu vzešel požadavek na načítání pravidel z jiného zdroje (např. XML nebo databáze), bude potřeba pouze naimplementovat vlastní loader, podobu pravidel ve zbytku frameworku to neovlivní. Jediné, co bude potřeba v jiném zdroji zachovat, je podoba výrazů pravidel pomocí expression language, aby se nadále dal zpracovávat knihovnou PHP Parser a šlo o syntakticky validní PHP kód. Vše ostatní může být vyjádřené jiným způsobem, takže třeba odkazování se na jiná pravidla se dá v databázi realizovat pomocí cizích klíčů.

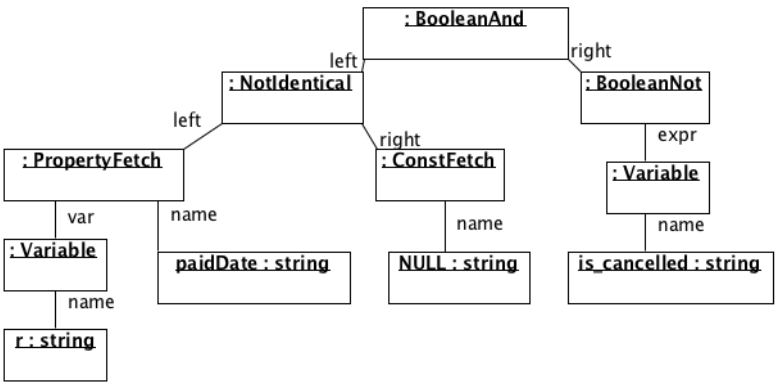
4.7 RuleParser

Tenká abstrakce nad knihovnou PHP Parser zajišťující, že pravidlo obsahuje jediný výraz. V případě, že uživatel do konfigurace pravidla zapíše více výrazů oddělených středníkem, RuleParser vyhodí výjimku.

4.8 Syntaktický strom výrazu pravidla

Předpis každého pravidla je transformován pomocí knihovny PHP Parser na syntaktický strom, se kterým se dále pracuje při vyhodnocování v PHP (kapitola 7) a při transformaci do DQL (kapitola 8).

Obrázek 4.2: Syntaktický strom `$r->paidDate !== NULL && !$is_cancelled`



Kapitola 5

Statická analýza pravidel

Staticky typované jazyky se vyznačují tím, že je určen typ každé proměnné, vstupních argumentů a návratových hodnot metod. Řadu chyb[1] dokáže odhalit kompilátor, který aplikaci s chybou nedovolí vůbec spustit. Eliminuje tím potřebu mít celou kategorii testů, které testují jen „wiring“ kódu a při testování aplikací se tak lze zaměřit na byznys logiku, kterou již kompilátor zkontrolovat nedokáže.

Statická analýza kódu je v PHP stále otevřené téma. PHP je dynamicky typovaný a interpretovaný jazyk, takže se chyba projeví až v momentě, kdy se spustí ta cesta v kódu, která ji obsahuje. Dynamická typovost se projevuje tak, že v PHP kódu nemá žádná proměnná ani výraz předem určený typ. Ten se vždy odvíjí od hodnoty, kterou obsahuje, což výrazně ztěžuje možnosti statické analýzy.

Existují ovšem cesty, jak tuto zabudovanou nejistotu alespoň částečně obejít. Vstupním argumentům funkcí a metod je možné určit vyžadovaný typ pomocí tzv. *type hintingu*. Lze vynutit pole a konkrétní třídu či implementované rozhraní. Nelze vynutit skalární typy – řetězce, čísla a booleovské hodnoty. Pokud v těle metody nedochází k přiřazení jiné hodnoty do proměnné, případně k volání některé z funkcí, které ovlivňují lokální scope[2], lze se spolehnout, že daná proměnná splňuje vyžadovaný typ.

Další cesty, jak dosáhnout lepší informovanosti vývojáře či podpůrných nástrojů o typech v kódu, již nemají přímou podporu v jazyce. Informace o typech v atributech tříd a o návratových hodnotách funkcí a metod se realizují pomocí dokumentačních komentářů.

[1] Např. volání neexistující metody, předávání špatného počtu argumentů, předávání argumentů jiného typu, odkazování se na neexistující typ, odkazování se na nepřístupné atributy objektů apod.

[2] Funkce `extract` přijímá pole, z jehož indexů vytvoří stejně nazvané proměnné. Vzhledem k nebezpečnosti a nedeterminističnosti takového kódu se nedoporučuje ji používat. Jiným příkladem podobné funkce je `list` nebo již zavržená funkcionalita `register_globals`.

Nejpokročilejším nástrojem na statickou analýzu v PHP, který hledá přesně ty chyby běžně nacházené u staticky typovaných jazyků jejich kompilátory, je PHP Analyzer [3]. Čím více analyzovaný kód používá type hinting a dokumentační komentáře, tím lépe dokáže odhalovat chyby. Využívá k tomu *type inference*[1]. V dynamicky typovaném jazyku ale tyto kontroly nemohou být tak spolehlivé jako ve staticky typovaných jazycích. PHP tomu brání i takovými vlastnostmi, jako je instanciace objektu na základě názvu třídy uložené v proměnné (`new $className`), obdobné volání metod (`$this->$method()`) nebo reflexe.

5.1 Acykličnost pravidel

Statická analýza pravidel se odehrává ve třídě RulesValidator. Aby při jejich vyhodnocování nebo transformování nemohlo dojít k zacyklení (nekonečné rekurzi), je potřeba zajistit, aby graf pravidel, ve kterém hrany představují odkazy na pravidla, neobsahoval kružnici. Hledám ji pomocí procházení do hloubky. Při vstupu do uzlu (procházení odkazovaných pravidel rekurzí) ho označím jako otevřený a vložím na vrchol zásobníku. Při uzavírání uzlu vrchol zásobníku opět odstraním. Pokud v rekurzi narazím na již otevřený uzel, znamená to, že jsem nalezl kružnici. Součástí zprávy vyhazované výjimky oznamující tuto chybu je i výpis všech uzlů, které kružnice obsahuje, pro snadnější ladění a její odstraňování. Jejich seznam získávám tak, že procházím zásobník od jeho vrcholu, dokud nenarazím na aktuální uzel podruhé. Nalezené uzly pak vypíšu v opačném pořadí:

Rule „a“ is part of a circular reference [a -> b -> c -> d -> a].

Při procházení do hloubky pak na jednotlivých pravidlech provádím i další kontroly.

5.2 Nepoužité a nedefinované argumenty

Napříč celým frameworkem používám na několika místech vlastní třídu NodeTraverser s metodou `traverse()` z ukázky 5.1. Ta na pozadí rekurzivně prochází syntaktický strom pravidla a pokud narazí na uzel vyžádaného typu, volá poskytnutý callback. Tento návrh používám všude, kde mě uvnitř pravidla v libovolné hloubce zajímají všechny uzly určitého typu.

Při kontrole nepoužitých a nedefinovaných argumentů používám NodeTraverser pro nalezení všech proměnných – uzlů typu PhpParser\Node\Expr\Variable. Pokud narazím na proměnnou, jejíž název není v kolekci argumentů,

[1] Automatická dedukce typu výrazu.

Zdrojový kód 5.1: Použití metody `NodeTraverser::traverse()` pro procházení syntaktického stromu pravidla. První parametr představuje procházený strom, druhý parametr typ hledaného uzlu a třetí funkci, která je při nalezení vyžadovaného uzlu zavolána. Ukázka vypíše všechny uzly, které představují proměnnou.

```
$nodeTraverser->traverse(
    $rule->getTree(),
    'PhpParser\Node\Expr\Variable',
    function(Node $node) {
        var_dump($node);
    }
);
```

jedná se o nedefinovaný argument a vyhazuji výjimku. Při procházení proměnných si poznačuji, které argumenty byly použité. Pokud po ukončení procházení zbydou nějaké neoznačené, jedná se o nepoužité argumenty a jelikož je zbytečné je definovat a vyžadovat jejich předání při vyhodnocování, také vyhazuji výjimku.

5.3 Volání funkcí

Syntaxe pravidel podporuje dva důvody volání funkcí: odkazování se na jiné pravidlo a volání agregační funkce.

Při odkazování se na jiné pravidlo je názvem funkce proměnná (`$is_cancelled([r => $reservation])`) s názvem argumentu pravidla. Parametr volání je jeden – pole s předpisem, pod jakými názvy se mají do pravidla předat proměnné z toho aktuálního. Ve validátoru tyto náležitosti kontroluji.

Všechny agregační funkce přijímají také právě jeden parametr – vyhodnocovaný výraz. Ten musí obsahovat právě jeden kolekční argument. Agregační funkce popisuji v kapitole Návrh frameworku, sekci 3.7.

5.4 Nedefinované atributy objektů

Při odkazování se na atributy objektů mohu jednoznačně určit, zdali atribut existuje, protože z definice argumentů je znám typ objektu. Oproti PHP totiž neumožňuji uvnitř pravidla přepisovat hodnotu definovaného argumentu. Stačí tedy využít reflexi a zjistit existenci a viditelnost atributu. V případě, že existuje, ale je private či protected, zkontroluji ještě existenci getteru podle konvence popsané v kapitole 3.4.

5.5 Nerealizované kontroly

Z důvodů uvedených na začátku této kapitoly ani statická analýza byznys pravidel nemůže být stoprocentní. Oproti PHP je tento framework ale ve výhodě, protože podporuje pouze omezené vyjadřovací prostředky. Vedle popsaných kontrol se nabízí několik dalších, které by šly relativně snadno realizovat.

Kontrola typů na obou stranách striktních porovnávání
: Ve frameworku podporuji pouze striktní porovnávání (`===` a `!==`), která vždy vrací `FALSE` pokud jsou na jejich stranách hodnoty různých typů. Statická analýza by mohla takové případy detekovat – známe návratové typy logických operátorů (boolean) a agregačních funkcí (integer nebo float). V případě přistupovaných atributů objektů by bylo třeba číst jejich dokumentační komentáře a hledat v nich anotaci `@var`. Tuto anotaci ovšem není povinné uvádět. Statická analýza by ji mohla vyžadovat a v případě její neexistence vyhodit chybu, nebo kontrolu takových atributů přeskakovat.

Detekce tautologických a kontradiktorických podmínek
: Tautologické podmínky jako `$foo || !$foo` mohou zapříčinit nedosažitelnost podvýrazů a jejich existence je pravděpodobně nezamýšlená a nežádoucí. Kontradiktorické (`$foo && !$foo`) pak představují podobný problém. Ne vždy jsou ovšem poznat na první pohled a mohou vzniknout ve složitějším výrazu nebo kombinací několika pravidel. Jejich detekce by šla realizovat redukcí booleovských výrazů na principu Karnaughovy mapy.

Použité třídy by měly být entitní a atributy persistované
: Primární účel frameworku je vyhodnocování v PHP a transformace pravidel do SQL za pomoci Doctrine 2 ORM a příbuzného jazyka DQL. Použité třídy by tedy měly být validní Doctrine entity, jinak překlad do DQL selže. Entitní třídy se vyznačují anotací `@Entity` v dokumentačním komentáři nad třídou a atributem s anotací `@Id` obsahující primární klíč. Další persistované atributy musí mít anotaci `@Column`. Ačkoli je možné tyto požadavky kontrolovat, znemožnilo by to použít podmnožinu pravidel pouze pro vyhodnocování v PHP, což by představovalo zbytečné omezení pro vývojáře. Řešením by mohlo být tuto kontrolu volitelně vypnout či zavést druhou úroveň chyb – „warnings“, které by nezamezovaly zpracování pravidel.

5.6 Statistika o pravidlech

Knihovna Symfony Console [10] usnadňuje implementaci a dokumentaci příkazů pro příkazovou řádku. Bez ní by byl vývojář odkázán na manuální parso-

Zdrojový kód 5.2: Ukázka metody `configure` v příkazech Symfony Console

```
protected function configure()
{
    $this->setName('amphibian:stats')
        ->setDescription('Outputs stats about rules')
        ->addArgument(
            'files',
            InputArgument::REQUIRED | InputArgument::IS_ARRAY,
            'Files containing rules to analyze'
        );
}
```

vání a kontrolu předaných parametrů skriptu. Každý příkaz v Symfony Console má podobu zvláštní třídy poděděné od Symfony\Component\Console\Command\Command a musí implementovat dvě metody - `configure` a `execute`. První obsahuje definici příkazu – název, popis a vyžadované či volitelné argumenty (viz zdrojový kód 5.2). Druhá pak přijímá objekty pro vstup a výstup a obsahuje samotnou implementaci.

Naimplementoval jsem příkaz `amphibian:stats`, který vypíše několik statistických ukazatelů – celkový počet pravidel, kolikrát jsou pravidla odkazována z jiných, zmiňované třídy a volané funkce. Všechna čísla uvádí v absolutních i relativních počtech (%). Příkaz přijímá cesty k YAML souborům s pravidly oddělené čárkami. Ukázka statistik nad testovací sadou pravidel:

```
Number of rules: 23
Number of rules with manual DQL transformation: 1 (4.35%)
Number of classes in object arguments: 2
Number of distinct called functions: 5

Classes

Amphibian\Reservation (mentioned 19 times - 82.61% of all rules)
DateTime (mentioned 1 time - 4.35%)

Functions

count (mentioned 4 times - 17.39%)
max (mentioned 1 time - 4.35%)
min (mentioned 1 time - 4.35%)
avg (mentioned 1 time - 4.35%)
sum (mentioned 1 time - 4.35%)
```

Referenced rules

reservation_is_expensive (mentioned 3 times - 13.04%)
count_expired_reservations (mentioned 2 times - 8.70%)
reservation_is_cancelled (mentioned 2 times - 8.70%)
reservation_cancelled_because_of (mentioned 1 time - 4.35%)
reservation_is_expired (mentioned 1 time - 4.35%)
reservation_is_paid (mentioned 1 time - 4.35%)

Kapitola 6

Struktura projektu

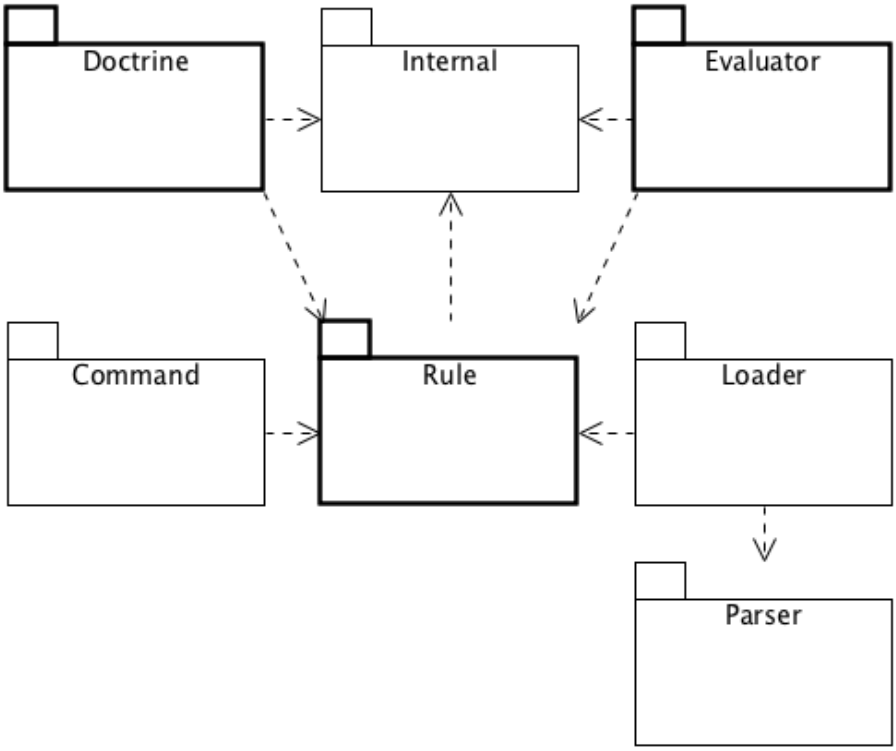
Framework sestává ze 4537 řádků produkčního kódu a 1855 řádků testů[1].

6.1 Adresářová struktura

```
amphibian/
├── src/ ..... Implementace frameworku
│   └── Amphibian/
│       ├── Command/ ..... Příkazy pro příkazovou řádku
│       ├── Doctrine/ ..... Transformace pravidel do DQL
│       │   ├── Functions/ ..... Vlastní DQL funkce
│       │   └── Node/ ..... Syntaktické uzly pro převod do DQL
│       ├── Evaluator/ ..... Vyhodnocování pravidel v PHP
│       │   └── Node/ .. Syntaktické uzly pro vyhodnocování pravidel v PHP
│       ├── Internal/ ..... Utility, co se jinam nevešly
│       ├── Loader/ . Načítání pravidel – FileLoader, YamlLoader a výjimky
│       ├── Parser/ ..... Parsování předpisů pravidel
│       └── Rule/ ..... Objektová reprezentace pravidel a statická analýza
├── tests/ ..... Testy frameworku
│   ├── Amphibian/ ..... Testovací případy ve struktuře zrcadlící src/
│   ├── lib/ ..... Pomocné třídy DIC pro skládání objektů v testech
│   └── ..... Další pomocné soubory pro spouštění testů
├── vendor/ ..... Adresář se závislostmi, který vygeneruje Composer
├── .travis.yml ..... Konfigurace Travis CI
├── composer.json ..... Konfigurace závislostí nástroje Composer
├── composer.lock Vygenerovaný soubor s konkrétními revizemi závislostí
├── Makefile ..... Konfigurace nástroje make
└── README.md ..... Návod, jak projekt zprovoznit pro vývoj
```

[1] Údaje zjištěny pomocí příkazu `git df --stat a6c8d8..origin/master src/`, resp. `tests/Amphibian/`

Obrázek 6.1: Diagram balíčků – závislosti mezi jmennými prostory



6.2 Jmenné prostory

V obrázku 6.1 jsou znázorněny závislosti mezi jmennými prostory. Důležité na něm je, že jmenný prostor Rule nemá kromě Internal žádnou závislost a Doctrine a Evaluator jsou závislé pouze na Rule a na Internal. Jednoduchým grafem závislostí je zařízena snadná rozšiřitelnost – při přidávání další zásadní funkcionality (např. přidání třetího prostředí pro vyhodnocování pravidel) nebude třeba zasahovat do existujících balíčků.

6.3 Composer

Nástroj Composer slouží ke správě závislostí projektu v PHP, lze ho přirovnat k nástrojům npm, NuGet nebo CocoaPods pro jiné platformy. Kromě závislostí řeší i autoloading[1].

[1]Pokud si aplikace vyžádá třídu, o které PHP neví, zavolá se nakonfigurovaný autoloader, který nalezne a načte soubor, který třídu obsahuje. Strategií pro autoloading je řada – PSR-0 a PSR-4 vyžadují, aby adresářová struktura souborů kopírovala strukturu jmenných prostorů a tříd, RobotLoader z Nette naopak nevyžaduje žádnou konvenci a třídy nalezne kdekoli v nakonfigurovaných adresářích.

Zdrojový kód 6.1: Konfigurace závislostí v souboru composer.json

```
{
        "require": {
                "php": ">=5.5",
                "nette/nette": "~2.1",
                "nikic/php-parser": "1.0.*@dev",
                "symfony/yaml": "~2.4",
                "doctrine/orm": "~2.4",
                "symfony/console": "~2.4"
        },
        "require-dev": {
                "phpunit/phpunit": "4.0.*",
                "symfony/dependency-injection": "~2.4",
                "symfony/config": "~2.4",
        }
}
```

Vedle composer.json s konfigurací projektu je důležitý i generovaný soubor composer.lock. Obsahuje konkrétní revize všech načtených závislostí, což je důležité pro sjednocení verzí knihoven na všech místech, kde je aplikace provozována, typicky produkční servery a vývojářská prostředí. Tento soubor by měl být verzovaný.

Kapitola 7

Vyhodnocování v PHP

Cílem vyhodnocování pravidel v PHP je dosažení stejného výsledku, jakého by dosáhl ekvivalentní kód zapsaný v PHP. Jelikož pravidla splňují syntaxi PHP, nabízelo by se k vyhodnocování využít funkci `eval`, která přijímá řetězec s PHP kódem, který vykoná. To ale nemohu z následujících důvodů.

7.1 Proč ne eval?

Funkce `eval` [5] představuje bezpečnostní riziko, pokud se do ní jako parametr může dostat uživatelský vstup. Jelikož vyvíjený framework je obecný a univerzální nástroj, nelze předjímat, jakým způsobem bude používaný. Pokud by vývojář, který ho využívá, umožnil uživatelům aplikace zadávat vlastní pravidla, která by se dostala do funkce `eval`, vytvořil by tak bezpečnostní díru. Kód se spouští v kontextu aplikace a má tak k dispozici volný přístup do databáze nebo k souborovému systému. Z tohoto důvodu bývá na sdílených hostinzích tato funkce často i zakázána pomocí konfigurační direktivy `disable_functions`.

Dalším důvodem je kompatibilita. Pravidla sice splňují syntaxi PHP, ale nikoli již sémantiku. Význam jednotlivých konstruktů může být odlišný. V pravidlech umožňuji volat agregační funkce (viz kapitola 3.7), které v PHP neexistují. Proto by v některých případech volání `eval` bez úprav předávaného řetězce ani nevedlo ke kýženému výsledku.

Posledním důvodem je výkon. Každý řetězec předaný do funkce `eval` musí PHP zparsovat a až poté vykonat. Nelze při tom využít OPCache, která je svázaná se zdrojovými kódy v souborech. Více toto téma rozebírám v kapitole 9 – Optimalizace.

7.2 Průběh vyhodnocení

Implementace vyhodnocování pravidel v PHP je ve jmenném prostoru Amphibian\Evaluator. Fasádou, která vyhodnocování poskytuje, je rozhraní s názvem Evaluator (zdrojový kód 7.1). Přijímá název pravidla a argumenty k vyhodnocení. V této kapitole popíšu tzv. runtime implementaci vyhodnocování. Zkompilovaná varianta je předmětem kapitoly o optimalizaci.

Zdrojový kód 7.1: Rozhraní Evaluator

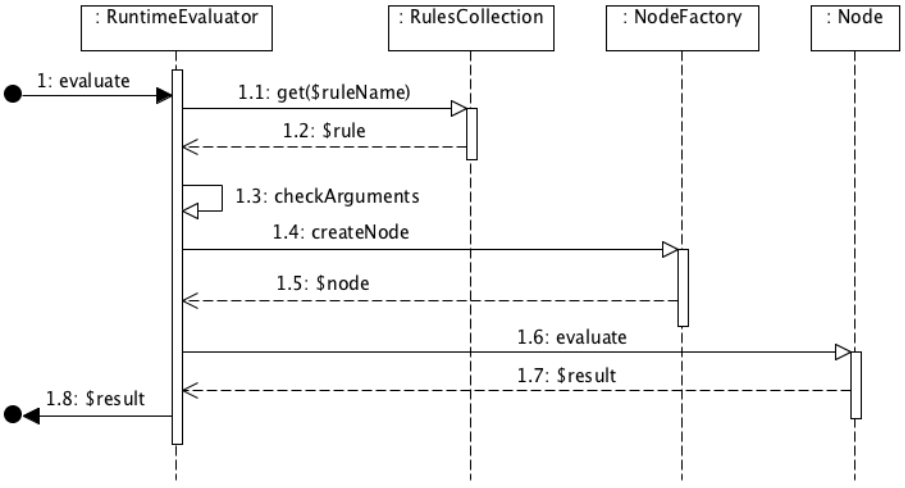
```
interface Evaluator
{

    /**
     * @param string $ruleName
     * @param string[] $arguments
     * @return mixed evaluated result
     */
    public function evaluate($ruleName, array $arguments);

}
```

RuntimeEvaluator si na základě názvu pravidla vyžádá od RulesCollection jeho objektovou reprezentaci. Následně zkontroluje správnost a úplnost předaných argumentů podle definice. Třída NodeFactory ze syntaktického stromu pravidla knihovny PHP Parser vytvoří vlastní obdobný strom s implementací vyhodnocování. Tento strom sestává z uzlů typu Amphibian\Evaluator \Node\Node. Toto rozhraní obsahuje metodu `evaluate` s parametrem `$arguments`. Při vyhodnocování se tato metoda zavolá na kořeni stromu. Všechny uzly ji pak volají na svých potomcích a vrací své mezivýsledky.

Obrázek 7.1: Sekvenční diagram vyhodnocování pravidel v PHP



Kapitola 8

Transformace do DQL a integritních omezení

Probíhá obdobně jako vyhodnocování v PHP. Fasádu reprezentuje rozhraní Amphibian\Doctrine\DqlBuilder. Jeho metoda `buildDql` přijímá název pravidla a pomocí NodeFactory převádí jeho syntaktický strom na vlastní. Každý uzel má taktéž metodu `buildDql`, která ovšem už nepřijímá žádné argumenty. Potřebné informace (typicky poduzly) jsou do uzlů předány v jejich konstruktorech. Skládání DQL probíhá zavoláním `buildDql` na kořeni tohoto stromu.

Jmenné prostory Evaluator a Doctrine mají každý svůj soubor uzlů. Sjednocení do jedné struktury, která by uměla jak vykonávání v PHP, tak transformaci do DQL, by představovala porušení principu otevřenosti a uzavřenosti. Přidání podpory pro další prostředí by znamenalo zásah do existujících zdrojových kódů. Struktury navíc nelze na sebe namapovat jedna ku jedné.

8.1 Proměnné a parametry

Při vyhodnocování v PHP mapuji všechny proměnné na objekt VariableNode. V DQL ovšem rozlišuji mezi proměnnými, které představují jméno z dotazu vzniklé klauzulemi `FROM` nebo `JOIN`, a proměnnými, které se odkazují na skalární hodnoty. Ty představují parametry, v DQL se zapisují s dvojtečkou a jejich hodnotu předává uživatel metodou `setParameter`. Z proměnné ve výrazu může při transformaci vzniknout objekt VariableNode, nebo DqlParameterNode.

Zdrojový kód 8.1: `$r` je proměnná, `$cost` je DQL parametr. Pravidlo se převede na `r.price > :cost`.

```
arguments:
  cost: integer
  r: Amphibian\Reservation
rule: $r->price > $cost
```

Zdrojový kód 8.2: Jak vypadá pravidlo `reservation_is_paid` z ukázky 3.3 na straně 23 po automatickém převodu do DQL

```
r.paidDate IS NOT NULL AND r.cancellationReason IS NULL
```

8.2 Porovnávání a logické operátory

Matematické porovnávací operace `<`, `<=`, `>` a `>=` framework do DQL přepíše jedna ku jedné. `===` a `!==` se v DQL zapisují jako `=` a `!=`, případně `<>` [2]. Výjimku tvoří porovnávání s prázdnou hodnotou `NULL`. V tom případě se na rovnost dotazuje pomocí `IS NULL`, na nerovnost `IS NOT NULL`, stejně jako v SQL. Tato logika je ve jmenném prostoru Amphibian\Doctrine\Node, třídách IdenticalNode a NotIdenticalNode.

Navíc ještě obsahuje ošetření a zjednodušení DQL v případech, kdy se porovnává s booleovskými hodnotami. Pokud se levá strana výrazu má rovnat `true`, mohu do DQL zapsat jen ji. Pokud se má rovnat `false`, vrátím negaci levé strany. Analogicky s nerovností. Kromě toho eliminuji nadbytečné ozávorkování výrazů, pokud je transformován složitější výraz.

8.3 Negování DQL

Pokud je před výrazem v pravidle uveden `!` nebo v případech popsaných v předchozím odstavci, jedná se o negaci pravidla, tedy úmysl vyjádřit jeho význam opačně. V DQL se dá vyjádřit obalením výrazu konstrukcí `NOT(...)`. Někdy lze ale použít čitelnější variantu, ve které se přepíše vnitřek výrazu namísto jeho obalení. Např. z `IS NOT NULL` se tak stane `IS NULL`. Z tohoto důvodu existuje rozhraní CustomNegationNode s metodou `buildNegatedDql`. Jediné místo s kontrolou, zdali ho uzel implementuje, je ve třídě BooleanNotNode. Ten pro negaci použije buďto implementaci této metody, nebo vložený uzel obalí `NOT(...)`.

8.4 Odkazování se na ostatní pravidla

Transformované předpisy odkazovaných pravidel se jednoduše vloží do výsledného DQL dotazu. Pokud při odkazování se na pravidlo dochází k přepisu názvů předávaných proměnných jako v ukázce 3.4 na straně 25, jsou v celé hierarchii odkazovaných pravidel zachovány názvy z toho nejvyššího – kořenového. Jedině tak může transformované DQL fungovat, protože ve výsledném dotazu se bude daný objekt vyskytovat právě pod tímto názvem. Viz

Zdrojový kód 8.3: Pravidlo se odkazuje na `reservation_is_paid` z ukázky 3.3 na straně 23. Převod nealiasované varianty je v ukázce 8.2. Tato varianta se převede na `s.paidDate IS NOT NULL AND s.cancellationReason IS NULL`.

```
reservation_is_paid_alias:
  arguments:
    is: @reservation_is_paid
    s: Amphibian\Reservation
  rule: $is([r => $s])
```

ukázka 8.3. Přepis názvů proměnných probíhá v NodeFactory v metodách `translateVariables` a `translateRule`.

Zde je důležitá neměnnost (*immutability*) objektové reprezentace pravidel. Nutí mě totiž pro odkazovaná pravidla vytvářet nové instance, ve kterých přepisuji názvy předávaných proměnných. Umožnit změnu v existujících instancích by ovlivnilo i jiná pravidla, která se na ně odkazují, ale i využití tohoto pravidla samotného.

8.5 Agregační funkce

SQL i DQL umí agregační funkce, které ve frameworku podporuji nativně (viz kapitola 3.7). Převod jejich volání je tedy triviální. Zařizuje ho třída FunctionCallNode. Na uživateli posléze je, zdali převedené DQL využije v klauzuli `SELECT` nebo `HAVING`. Ve `WHERE` agregační funkce DQL ani SQL nepodporují.

Experimentálně jsem rozšířil význam agregační funkce `COUNT`. Ta v relačních databázích umí dva typy zápisů: `COUNT(sloupec)` a `COUNT(*)`. V případě zmínky o konkrétním sloupci přičítá za každý řádek k výsledku 1, pokud hodnota v tomto sloupci není `NULL`. Nefunguje to ovšem v případě složitějších výrazů a je třeba použít tento konstrukt:

```
SUM(CASE WHEN r.price > ? THEN 1 ELSE 0 END)
```

Pokud FunctionCallNode narazí na volání `COUNT` se složitějším výrazem, převede ho na tento konstrukt. Využívá k tomu vlastní DQL funkci SUM_CASE nalezitelnou v Amphibian\Doctrine\Functions\SumCaseFunction. DQL totiž nepodporuje vkládání složitějších výrazů do funkce SUM.

Pravidlo `count($r->price > $limit)` se převede na DQL takto:

```
SUM_CASE(CASE WHEN r.price > :limit THEN 1 ELSE 0 END)
```

Pro využití této funkcionality je třeba v konfiguraci Doctrine zaregistrovat funkci SUM_CASE metodou `addCustomStringFunction` objektu Doctrine\ORM\Configuration.

8.6 Manuální transformace

Framework má za úkol generovat kritický kód – databázové dotazy. Generovaný kód nemusí být vždy v souladu s tím, jak si ho představuje uživatel frameworku. Proto umožňuji DQL podobu pravidla určit manuálně. Stačí v YAML souboru ke konkrétnímu pravidlu přidat vedle klíčů `arguments` a `rule` klíč `dql` s dotazem. V případě, že YamlLoader na takové pravidlo narazí, vytvoří objekt typu CustomDqlRule namísto Rule. Pokud si uživatel od DqlBuilderu vyžádá DQL takového pravidla, vůbec se nespustí mechanismus transformace výrazu, ale rovnou se vrátí zapsaný řetězec.

Využití této funkcionality způsobí porušení principu DRY, protože do systému budou zaneseny dvě podoby toho samého pravidla – původní výraz určený k automatické transformaci a manuálně transformovaný dotaz.

8.7 Integritní omezení

Pokud EntityManager z Doctrine[1] při ukládání změn způsobí databázovou chybu (vyhodí výjimku PDOException), uzavře se a následná ukládání změn již neumožní. Je to logické – tuto chybu způsobují změny připravené k uložení, které EntityManager obsahuje a z toho se lze zotavit pouze vytvořením nového „čistého“ EntityManageru. Integritní omezení na straně databáze by tedy měla sloužit pouze jako poslední ochrana před nekonzistentními daty bez očekávání, že se z jejich případného porušení dokáže aplikace během aktuálního požadavku zotavit.

Generování kódu integritních omezení nalezne využití při validaci entit – pokud se databázový záznam musí nacházet v jednom z validních stavů, lze vygenerovat integritní omezení spojením pravidel reprezentujících tyto stavy logickou spojkou `OR`. Integritní omezení ve tvaru `ALTER TABLE ... ADD CONSTRAINT ... CHECK (...)` (již v jazyce SQL, ne DQL) vytváří třída SqlConstraintBuilder.

[1] Fasáda, přes kterou probíhá veškerá práce s entitami.

Kapitola 9

Optimalizace

V dosavadně popsané funkcionalitě odvádí framework veškerou práci během runtime fáze. Při každém prováděném HTTP požadavku tak musí načíst a zparsovat YAML soubor s pravidly, zparsovat všechny výrazy pravidel a provést statickou analýzu – viz sekvenční diagram 4.1 na straně 31. Při vyhodnocování či transformaci každého pravidla pak převede syntaktický strom výrazu na vlastní syntaktický strom postavený k danému účelu a nakonec využije všechny jeho uzly k výpočtu či sestavení DQL. Za cenu menšího výkonu tento způsob umožňuje rapidní vývoj, protože není potřeba při každé změně nic kompilovat, stačí uložit soubor a obnovit stránku v prohlížeči nebo spustit testy.

Pro dosažení vyššího výkonu je nutné co nejvíce práce frameworku přesunout do compile time fáze. I když je PHP interpretovaný a ne kompilovaný jazyk, tento pojem se používá. Označuje se tak soubor činností, které se odehrají jednou při nasazování aplikace či při prvním požadavku a z jejichž výsledků pak aplikace těží při vyřizování každého následujícího požadavku až do dalšího nasazování aplikace, nebo smazání těchto výsledků.

Pro optimalizaci práce frameworku vytvořím generátor, který na základě načtených pravidel vygeneruje třídy splňující stejné rozhraní jako runtime varianty. Do zdrojového kódu těchto tříd budou „vštípeny“ všechny relevantní informace o pravidlech pro jejich optimální vyhodnocování a transformaci.

9.1 OPcache

PHP je interpretovaný jazyk. Zdrojový kód je převeden do podoby bytecodu[1], který je spouštěn interpretem. Bytecode PHP sestává z tzv. opcodes, což jsou jednotlivé vykonávané instrukce. Ve výchozím nastavení PHP parsuje zdrojové kódy a generuje z nich bytecode při každém požadavku. Od verze PHP 5.5

[1] Prostředník mezi zdrojovým kódem vyššího programovacího jazyka a zkompilovaným strojovým kódem.

Zdrojový kód 9.1: Vygenerované tělo metody pravidla `reservation_is_cancelled` z ukázky 3.3 na straně 23 pro vyhodnocování v PHP

```
if (isset($arguments['r'])) {
    if (!($arguments['r'] instanceof \Amphibian\Reservation)) {
        throw new \InvalidArgumentException(sprintf(
            'Argument r passed to the evaluator must be an instance of
                Amphibian\Reservation, %s given.',
            get_class($arguments['r'])
        ));
    }
} else {
    throw new \InvalidArgumentException('Argument r of type
        Amphibian\Reservation missing from the passed arguments.');
}

return $arguments['r']->getCancellationReason() !== NULL;
```

je ovšem součástí distribuce rozšíření OPcache[1], které bytecode při prvním spuštění nakešuje a v každém následujícím požadavku ho už PHP interpret pouze vykonává. OPcache je svázaná se souborovým systémem, při načítání zdrojových kódů z jiných míst a jejich spouštění pomocí funkce `eval` se nevyužije. „Klíčem“ opcodes v keši je cesta k souboru na disku. Při jeho změně lze nakešovaný bytecode invalidovat manuálně, na základě času poslední úpravy souboru, nebo v pravidelných intervalech [6].

9.2 Vyhodnocování v PHP

Pro vygenerování třídy vykonávající pravidla jsem do rozhraní všech uzlů syntaktického stromu přidal metodu `compile`, která vrací řetězec se zdrojovým kódem pravidla. Samotné generování provádí třída `CompiledEvaluatorGenerator`, která přijímá název třídy a cestu na disku, kam se má třída uložit. Pro každé pravidlo se do třídy vygeneruje privátní metoda. Ta kromě zdrojového kódu získaného zavoláním `compile` na kořenovém uzlu obsahuje i kontrolu konkrétních předaných argumentů, jejichž definici si generátor přečetl při kompilaci. Tyto privátní metody se jmenují podle předpisu `'evaluate_' . $ruleName`. Veřejná metoda `evaluate` pouze „rozstřeluje“ volání do těchto privátních metod. Třídu generuji za pomoci šablonovacího systému Latte pocházejícího z Nette Frameworku.

[1] V dřívějších verzích tuto funkcionalitu zastávala odděleně vyvíjená rozšíření APC nebo Zend OPcache

9.3 Transformace do DQL

Díky tomu, že pravidla při transformaci do DQL vrací vždy stejný řetězec, je i kompilovaná varianta této funkcionality jednodušší na implementaci. Do jednotlivých uzlů nemusím přidávat žádnou další metodu, ale pouze v generátoru proiteruji všechna pravidla a řetězce s DQL dotazy vracím v jednotlivých vygenerovaných metodách.

Zdrojový kód 9.2: Vygenerovaná třída s pravidlem `reservation_is_cancelled` z ukázky 3.3 na straně 23 pro transformaci do DQL

```
class CompiledDqlBuilder implements \Amphibian\Doctrine\DqlBuilder
{

    /**
     * @param string $ruleName
     * @return string DQL
     */
    public function buildDql($ruleName)
    {
        $method = 'build_' . $ruleName;
        return $this->$method();
    }

    /**
     * $r->cancellationReason !== NULL
     *
     * @return string
     */
    private function build_reservation_is_cancelled()
    {
        return 'r.cancellationReason IS NOT NULL';
    }

}
```

Kapitola 10

Testy

Motivace pro psaní a udržování automatizovaných testů je neustálé ověřování veškeré očekávané funkcionality. Testy samy o sobě nezajistí software bez chyb. Záleží, jak důkladné testy vývojář napíše a zdali pokryje mezní hodnoty, zvláštní případy apod. Testy slouží primárně k tomu, aby upozornily, pokud se úpravami kódu rozbije existující (otestovaná) funkcionalita. Proto je nutné je pravidelně pouštět, nejlépe nad každým commitem ve verzovacím systému. K tomu slouží prostředí tzv. průběžné integrace (angl. *continuous integration*). Při vývoji frameworku jsem použil nástroj Travis CI [12] s napojením na GitHub, kde je umístěn repozitář projektu. Konfiguraci sestavení lze nalézt v adresáři frameworku v souboru `.travis.yml`.

10.1 PHPUnit

Nejpoužívanějším nástrojem v PHP pro testování je PHPUnit. Jeho API je podobné nástrojům z jiným platforem, jako je JUnit nebo NUnit. Každá třída obsahující testy (TestCase) musí splňovat určité náležitosti, aby ji PHPUnit dokázal najít a spustit. Název třídy musí končit na `Test` a název souboru, který třídu obsahuje, na `Test.php`[1]. Jednotlivé testy musí být ve veřejných metodách třídy a jejich názvy musí začínat prefixem `test`. Viz ukázka 10.1.

10.2 @dataProvider

Tato funkcionalita se podobá tzv. parametrizovaným testům z JUnitu. Pokud chci vykonat ten samý test nad více soubory dat, data provider elegantně zabraňuje duplikaci kódu. V PHPUnitu se nad testovací metodu uvede anotace `@dataProvider` s názvem jiné metody, která vrací pole s daty, nad kterými se bude test spouštět. Položky první úrovně pole se mapují na jednotlivá volání

[1] Pojmenování souboru po třídě, kterou obsahuje, je v PHP best practice. Zároveň se tím i vynucuje existence právě jedné třídy v souboru.

Zdrojový kód 10.1: Ukázka testu v nástroji PHPUnit

```
class PropertyFetchNodeTest extends \Amphibian\TestCase
{

    public function testPublicProperty()
    {
        $node = new PropertyFetchNode(new VariableNode('t'),
            'provider');
        $transaction = new Transaction();
        $transaction->provider = 'foo';
        $result = $node->evaluate(array(
            't' => $transaction,
        ));
        $this->assertSame('foo', $result);
    }

}
```

testu, položky druhé úrovně pole jsou vkládány jako argumenty volané metody testu.

V testech frameworku tuto možnost široce využívám, protože se pro jejich povahu hodí – potřebuji jednotným způsobem otestovat řadu kombinací pravidel a očekávaných výsledků, jak při vyhodnocování v PHP, tak transformaci do DQL. Očekávané výsledky všech testovacích případů porovnávám s výstupy jak z runtime, tak kompilovaných variant. Je tak zajištěna jejich parita. Testy těchto funkcionalit jsou ve třídách EvaluatorTest a DqlBuilderTest.

10.3 Pokrytí kódu testy

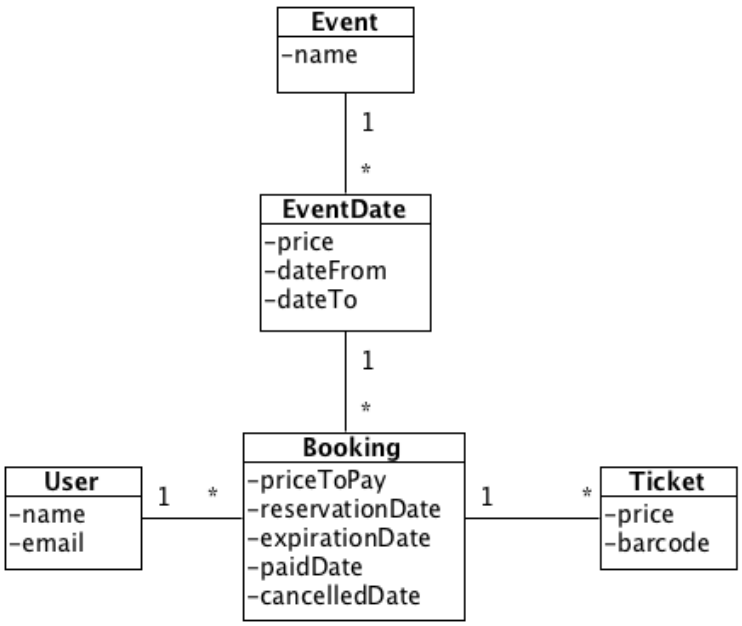
Framework se velmi jednoduše testuje, všechny cesty v kódu jsou snadno dosažitelné. Zásluhu na tom má kvalita kódu a dodržení zásad dobrého objektového návrhu SOLID (viz kapitola 1.4 na straně 7), ale také fakt, že nástroj nemá kromě příkazové řádky žádné uživatelské rozhraní, jehož testování je nejnáročnější. Z těchto důvodů jsem si vytyčil 100% pokrytí kódu testy, kterého se mi podařilo dosáhnout. Vygenerovaný report pokrytí z PHPUnitu je k dispozici na přiloženém DVD.

Kapitola 11

Ukázková aplikace

Ukázková aplikace demonstruje, k čemu byl framework navržen. Jako téma jsem si zvolil zjednodušené administrační rozhraní e-ticket portálu. Běžící aplikace je k dispozici na adrese `http://amphibian.mirtes.cz/`. Při vývoji jsem využil framework Nette. Vzhled určuje CSS framework Bootstrap.

Obrázek 11.1: Diagram entit v ukázkové aplikaci



Ve třídách, kde chci využít vyhodnocování pomocí pravidel, je v konstruktoru vyžadováno rozhraní Amphibian\Evaluator\Evaluator. V konfiguraci dependency injection určuji, že se na tato místa má vložit zkompilovaná varianta pro optimální výkon. Stejně tak s rozhraním DqlBuilder pro transformaci do DQL.

Evaluator v aplikaci nevyužívám přímo v controllerech, ale skrz fasádu modelové vrstvy. Jelikož metoda `evaluate` přijímá jako řetězec název pravidla, těžím z toho, že se za tímto účelem tato informace vyskytuje v aplikaci právě jednou. Získávám tím také přirozenější a jednodušší rozhraní pro zjišťování stavu entity:

Zdrojový kód 11.1: Metoda třídy BookingFacade využívající Evaluator

```
public function isPaid(Booking $booking)
{
    return $this->evaluator->evaluate(
        'booking_is_paid',
        array('b' => $booking)
    );
}
```

Entita události (*Event*) nabývá stavů „v prodeji“ a „uplynulá“. Rezervace (*Booking*) může být zaplacená, nezaplacená, nebo zrušená. Těchto stavů se týkají nakonfigurovaná pravidla, která se v aplikaci využívají k filtrování entit na straně databáze a k zobrazování štítků ve výpisu. V ladicím panelu na spodku obrazovky je výpis vykonávaných dotazů do databáze. Na všech obrazovkách, kde se vypisují data, se mi podařilo dosáhnout jejich stažení na jeden dotaz.

Zdrojový kód 11.2: Skládání DQL dotazu při filtrování rezervací

```
$qb = $this->entityManager
    ->createQueryBuilder()
    ->select('b, ed, e, u')
    ->from('AmphibianDemo\Model\Booking\Booking', 'b')
    ->join('b.eventDate', 'ed')
    ->join('ed.event', 'e')
    ->join('b.user', 'u')
    ->orderBy('b.id', 'ASC');

if ($filter === 'paid') {
    $qb->andWhere(
        $this->dqlBuilder->buildDql('booking_is_paid')
    );
} elseif ($filter === 'cancelled') {
    $qb->andWhere(
        $this->dqlBuilder->buildDql('booking_is_cancelled')
    );
} elseif ($filter === 'unpaid') {
    $qb->andWhere(
        $this->dqlBuilder->buildDql('booking_is_unpaid')
    );
}
```

Obrázek 11.2: Výpis rezervací s viditelnými štítky s jejich stavy

Všechny Zaplacené Zrušené Nezaplacené

#	Událost	Termín	Cena	Vstupenky	Datum rezervace	Zákazník	
1	Den direct marketingu 2014	24. 4. 2014	3 120 Kč	1×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
2	Jaroslav Uhlíř s kapelou	26. 4. 2014	2 880 Kč	2×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
3	Jaroslav Uhlíř s kapelou	16. 4. 2014	1 680 Kč	1×	3. 5. 2014	Pepa Vomáčka	Zaplacená
4	Jaroslav Uhlíř s kapelou	14. 4. 2014	7 680 Kč	4×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
5	Jaroslav Uhlíř s kapelou	27. 4. 2014	2 400 Kč	1×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
6	La Putyka	6. 5. 2014	1 890 Kč	3×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
7	Hrdý Budžes	19. 4. 2014	5 280 Kč	4×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
8	Radúz a Mahulena	19. 5. 2014	3 840 Kč	4×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
9	Radúz a Mahulena	27. 4. 2014	1 600 Kč	1×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
10	Radúz a Mahulena	6. 5. 2014	6 720 Kč	3×	3. 5. 2014	Pepa Vomáčka	Nezaplacená
11	Martin Rous a hosté	19. 5. 2014	3 420 Kč	3×	3. 5. 2014	Pepa Vomáčka	Zrušená

Závěr

V této práci se mi podařilo splnit vytyčené cíle. Naimplementoval jsem framework pro správu byznys pravidel, který umožňuje jejich vykonávání v PHP aplikaci i relační databázi, čímž eliminuje nutnost duplikovat kód a usnadňuje tak vývoj a údržbu aplikací.

Mám v plánu framework vydat pod open-source licencí a pokračovat v jeho vývoji. Chci využít svého postavení v české PHP komunitě, rozšířit povědomí o frameworku a získat zpětnou vazbu ohledně jeho využití v praxi.

Jako další možná platforma pro využití frameworku se nabízí skriptování ve webovém prohlížeči v jazyce JavaScript, okolo kterého panují podobná úskalí jako u relačních databází. Pokud dnes potřebuji provádět ten samý výpočet na klientu i na serveru[1], mám dvě možnosti: použít na serveru i na klientu stejný jazyk prostřednictvím node.js, což nemusí vždy vyhovovat, nebo všechny výpočty provádět na serveru a klientem se na ně dotazovat prostřednictvím AJAX požadavků, což vyžaduje připojení k Internetu a přináší nezanedbatelné zpoždění. Řešení podobné frameworku Amphibian je tedy nasnadě.

Dalším vítaným rozšířením bude určitě doplnění kontrol do statické analýzy navržených v kapitole 5.5 na straně 36. V případě, že se framework v praxi opravdu osvědčí, budu přemýšlet o jeho portu na jiné platformy, jako jsou C# nebo Java.

[1] Např. na klientu ukazovat živý náhled, jak dopadne nějaký výpočet po odeslání formuláře a na serveru tento výpočet pak provést.

Literatura

[1] Bootstrap [online]. [cit. 2014-04-19]. Dostupné z: http://getbootstrap.com/

[2] Doctrine Query Language – Doctrine 2 ORM documentation [online]. [cit. 2014-05-02]. Dostupné z: http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html

[3] GitHub: scrutinizer-ci/php-analyzer [online]. [cit. 2014-04-28]. Dostupné z: https://github.com/scrutinizer-ci/php-analyzer

[4] Nette Framework – Rozšíření jazyka PHP [online]. [cit. 2014-04-21]. Dostupné z: http://doc.nette.org/cs/2.1/php-language-enhancements

[5] PHP: eval – Manual [online]. [cit. 2014-05-01]. Dostupné z: http://www.php.net/manual/en/function.eval.php

[6] PHP: OPcache Runtime Configuration – Manual [online]. [cit. 2014-05-02]. Dostupné z: http://www.php.net/manual/en/opcache.configuration.php

[7] PHP: PHP type comparison tables – Manual [online]. [cit. 2014-04-25]. Dostupné z: http://www.php.net/manual/en/types.comparisons.php

[8] PHP: Type Juggling – Manual [online]. [cit. 2014-04-25]. Dostupné z: http://www.php.net/manual/en/language.types.type-juggling.php

[9] Symfony Documentation – Expression Language [online]. [cit. 2014-04-21]. Dostupné z: http://symfony.com/doc/current/components/expression_language/index.html

[10] Symfony Documentation – The Console Component [online]. [cit. 2014-04-29]. Dostupné z: http://symfony.com/doc/current/components/console/introduction.html

[11] Symfony Documentation – The YAML Component [online]. [cit. 2014-04-21]. Dostupné z: http://symfony.com/doc/current/components/yaml/introduction.html

[12] Travis CI: Continuous Integration [online]. [cit. 2014-05-03]. Dostupné z: https://travis-ci.com/

[13] PHP 5 Changelog: PHP 5.3 [online]. červen 2009, [cit. 2014-04-21]. Dostupné z: http://php.net/ChangeLog-5.php#5.3.0

[14] Bauer, C.; King, G.: *Java Persistence with Hibernate*. Manning Publications Co., druhé vydání, ISBN 1-932394-88-5.

[15] Ben-Kiki, O.; Evans, C.; döt Net, I.: YAML™Specification [online]. září 2009, [cit. 2014-04-21]. Dostupné z: http://yaml.org/spec/

[16] Fowler, M.: *Domain-Specific Languages*. Addison-Wesley, první vydání, ISBN 0321712943.

[17] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley, první vydání, ISBN 0-321-12742-0.

[18] Fowler, M.: RulesEngine [online]. leden 2009, [cit. 2014-04-21]. Dostupné z: http://martinfowler.com/bliki/RulesEngine.html

[19] Hunt, A.; Thomas, D.: *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley, první vydání, ISBN 0-201-61622-X.

[20] IBM Global Making Change Work (MCW) Study [online]. Technická zpráva, IBM, 2008, [cit. 2014-04-15]. Dostupné z: http://www-935.ibm.com/services/us/gbs/bus/pdf/gbe03100-usen-03-making-change-work.pdf

[21] Kelly, A.: Why do requirements change? Z *Overload Journal*, ACCU, únor 2004, issue 59.

[22] Marcotte, E.: Responsive Web Design [online]. květen 2010, [cit. 2014-04-19]. Dostupné z: http://alistapart.com/article/responsive-web-design/

[23] Martin, R. C.; Martin, M.: *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, první vydání, ISBN 978-0-13-185725-4.

[24] Mirtes, O.; Škvára, J.: Active Record & Data Mapper. Z *Studentské fórum návrhových vzorů*, České vysoké učení technické v Praze, první vydání, ISBN 978-80-01-04874-0.

[25] Munroe, A.: PHP: a fractal of bad design [online]. duben 2012, [cit. 2014-04-21]. Dostupné z: http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/

[26] Pilgrim, M.: Using Optional and Named Arguments. Z *Dive Into Python*, Apress, ISBN 978-1590593561.

[27] Popov, N.: GitHub: nikic/PHP-Parser [online]. [cit. 2014-04-21]. Dostupné z: https://github.com/nikic/PHP-Parser

[28] Popov, N.: Understanding PHP's internal array implementation [online]. [cit. 2014-04-27]. Dostupné z: http://nikic.github.io/2012/03/28/Understanding-PHPs-internal-array-implementation.html

[29] Potencier, F.: New in Symfony 2.4: The ExpressionLanguage Component [online]. listopad 2013, [cit. 2014-04-21]. Dostupné z: http://symfony.com/blog/new-in-symfony-2-4-the-expressionlanguage-component

[30] Protalinski, E.: IE10 blows past IE7 and IE6's combined market share, Firefox gains too, but Chrome hits 21-month low [online]. červen 2013, [cit. 2014-04-19]. Dostupné z: http://thenextweb.com/insider/2013/06/01/ie10-blows-past-ie7-and-ie6s-combined-market-share-firefox-gains-too-but-chrome-hits-21-month-low/

[31] Schwartz, B.; Zaitsev, P.; Tkachenko, V.; aj.: Scaling and High Availability. Z *High Performance MySQL*, O'Reilly Media, druhé vydání, ISBN 978-0-596-10171-8.

[32] Thakur, D.: What is Object Oriented Database (OODB)? Advantages and Disadvantages of OODBMSS. [online]. [cit. 2014-04-20]. Dostupné z: http://ecomputernotes.com/database-system/adv-database/object-oriented-database-oodb

[33] Turbak, F.; Gifford, D.: *Design Concepts in Programming Languages*. The MIT Press, první vydání, ISBN 0262201755.

[34] Vyskočil, J.; Mařík, R.: Pokročilá algoritmizace – Combinatorial algorithms. Přednáška, 2012, [cit. 2014-04-25]. Dostupné z: https://cw.felk.cvut.cz/wiki/_media/courses/a4m33pal/pal07.pdf

[35] HTML5 – A vocabulary and associated APIs for HTML and XHTML [online]. Technická zpráva, W3C, 2014, [cit. 2014-04-19]. Dostupné z: http://www.w3.org/TR/html5/

[36] W3Techs: Usage of server-side programming languages for websites [online]. duben 2014, [cit. 2014-04-21]. Dostupné z: http://w3techs.com/technologies/overview/programming_language/all

[37] Wikipedie: Normalizace databáze [online]. [cit. 2014-04-20]. Dostupné z: http://cs.wikipedia.org/wiki/Normalizace_datab%C3%A1ze

[38] Čemus, K.: *Automatická integrace byznys pravidel v adaptabilních uživatelských rozhraních*. Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, 2013.

Příloha **A**

Seznam použitých zkratek

AJAX Asynchronous JavaScript and XML

CI Continuous integration

CPU Central processing unit

CSS Cascading Style Sheets

DDL Data definition language

DIC Dependency Injection Container

DQL Doctrine Query Language

DRY Don't repeat yourself

DSL Domain-specific language

DVD Digital versatile disc

HQL Hibernate Query Language

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IaaS Infrastructure as a service

JPQL Java Persistence Query Language

JSON JavaScript Object Notation

OOP Objektově orientované programování

ORM Objektově relační mapování

PaaS Platform as a service

PHP PHP: Hypertext Preprocessor

RAM Random-access memory

SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation a Dependency inversion

SQL Structured Query Language

W3C World Wide Web Consortium

XML Extensible markup language

YAML YAML Ain't Markup Language

Příloha **B**

Obsah přiloženého DVD

- `amphibian/` – Zdrojové kódy frameworku – adresář popsaný v kapitole 6.1
- `amphibian-demo/` – Zdrojové kódy ukázkové aplikace
- `code-coverage/` – Vygenerovaný report pokrytí frameworku testy
- `thesis/` – Text práce
  - `source/` – Zdrojový kód práce
  - `DP_Mirtes_Ondrej_2014.pdf` – Text práce ve formátu PDF