FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Rozšíření nástroje pro anonymizaci dat
Student:	Pavel Perner
Vedoucí:	Ing. Jiří Mlejnek
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Analyzujte požadavky na existující nástroj Winch, které by umožnily formou pluginů dynamicky přidávat do tohoto nástroje nové funkčnosti.
Proveďte architektonickou analýzu nástroje Winch, popište aktuální stav nástroje vzhledem k analyzovaným požadavkům. Proveďte návrh nového řešení, které by umožňovalo naplnění specifikovaných požadavků.
Na základě návrhu proveďte refaktoring existujícího řešení a implementujte chybějící části.
Implementované řešení důkladně zdokumentujte a otestujte. V dokumentaci se zaměřte na programátorskou dokumentaci, umožňující tvorbu nových pluginů.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Rozšíření nástroje pro anonymizaci dat

Pavel Perner

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Mlejnek

13. května 2019

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce Ing. Jiřímu Mlejnkovi za cenné rady, věcné připomínky a vstřícnost při konzultacích. Velké díky patří také rodině, přátelům a partnerce, kteří mě po dobu tvorby práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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 13. května 2019 .....

Odkaz na tuto práci

Perner, Pavel. *Rozšíření nástroje pro anonymizaci dat.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Bakalářská práce se věnuje možnostem rozšiřování existujícího anonymizačního nástroje. V aplikaci byl přislíben tzv. „pluginovací“ systém, jenž by umožňoval dodávat vlastní komponenty podle definovaného postupu. Protože je rozvoj nástroje komplikovaný, nebyla popisovaná funkcionalita stále dodána.

V teoretické části práce pojednává o existujících možnostech rozšiřování aplikací. Dále seznamuje čtenáře s návrhovými vzory, konceptem refactoringu a samotným anonymizačním nástrojem. Praktická část se pak věnuje návrhu, realizaci a ověření pluginovacího systému pro aplikaci včetně nutných úprav kódu, které implementace vyžaduje. Obsahem je také programátorská příručka v elektronické podobě, jež koncové uživatele provází postupem tvorby pluginů pro nástroj.

Výsledkem práce je tak upravený nástroj, který nyní nejen koncovým klientům ale i samotným vývojářům aplikace umožňuje dodávat vlastní komponenty v podobě pluginů. Přidávat lze neomezené množství rozšíření a každé z nich může program obohacovat o jednu či více základních komponent.

Klíčová slova anonymizace, Winch, pluginovací systém, rozšiřování aplikace, Java, volitelné balíčky

Abstract

Bachelor thesis is focused on finding way to create extension mechanism for existing anonymization tool. This feature was promised to end users although was not introduced yet. Situation described is mainly caused by complications during app development.

The theoretical part of this thesis deals with possibilities of application extending. Besides those, it also introduce the reader to design patterns, concept of refactoring and the anonymization tool itself. The practical part is focused on design, realization and verification of the application extensions mechanism. It includes description of necessary code modifications required by the implementation as well. The thesis also contains the programming guide in its electronic form. The programming guide helps end users with the process of creating custom extensions for the tool.

The result of this thesis is a modified tool that now allows not only clients but developers of the tool themselves to supply their own components in the form of plugins. The anonymization tool is now able to handle unlimited number of extensions and each can add one or more basic components.

Keywords anonymization, Winch, plugin system, application extending, Java, optional packages

Obsah

Seznam obrázků

Seznam tabulek

Výpisy kódu

Úvod

Tato bakalářská práce se zabývá rozšiřováním existujícího anonymizačního nástroje *Winch* od společnosti GEM System a.s. Aplikace by měla být doplněna o tzv. „pluginovací“ systém, jenž by klientům umožňoval přidávat a používat vlastní komponenty. Popisovaná funkcionalita je jedním z výsledků analýzy požadavků, kterou prováděla firma samotná. Protože s sebou ale implementace nástroje nesla a stále nese mnoho obtíží, nebyla do aplikace tato možnost rozšiřování stále přidána, ačkoliv byla klientům přislíbena. Aktuálně se tedy jedná o businessově důležitou záležitost.

Téma práce bylo zvoleno na základě zájmu o problematiku okolo evropského obecného nařízení o ochraně osobních údajů (známém též pod anglickou zkratkou *GDPR*). Právě s příchodem nařízení v platnost stoupl zájem o nástroj Winch.

První kapitola představuje čtenáři hlavní a dílčí cíle bakalářské práce, které vychází primárně ze zadání. Část Analýza nástroje Winch více přibližuje aplikaci Winch a popisuje jeho aktuální stav vzhledem k cílům práce. Objevují se v ní i poznatky z již existujících prací, které se zabývaly obdobnou tématikou kolem programu Winch. Kapitola Požadavky konkrétně specifikuje požadavky na pluginovací systém a nutné úpravy v nástroji. Teoretická část Rešerše zkoumá existující řešení v možnostech rozšiřování aplikací a pojednává o návrhových vzorech a refactoringu pro účely správných úprav kódu aplikace. Návrhy řešení a jejich realizace je popsána v kapitole Návrh a realizace. Poslední kapitolou práce je Ověření. Ta seznamuje s postupem testování a celkovým ověřením realizovaného řešení podle vytvořené programátorské příručky.

Kapitola 1

Cíle práce

Hlavním cílem práce je doplnění chybějící funkcionality do existujícího nástroje Winch, určeného k anonymizaci dat. Absentovanou funkčností je rozšiřitelnost aplikace o vlastní komponenty, které budou přidávat dodatečné chování k dodávanému softwaru. Možnost rozšiřování bude poskytnuta koncovým uživatelům, zároveň by však měla usnadnit budoucí rozvoj nástroje, včetně přípravy a realizace individuálních řešení pro konkrétní klienty. Samotné rozšiřování nástroje bude realizováno formou „pluginů".

Dílčím cílem, který je zároveň nezbytný k naplnění hlavního cíle práce, je analýza požadavků. Ty se dělí na požadavky na samotné rozšiřování nástroje, které jsou obecně nastíněny v zadání práce, a dále se pak člení na požadavky na úpravu aplikace v její aktuálním stavu. Požadavky na úpravy nástroje budou výsledkem rozboru klíčových částí aplikace z pohledu architektury a designu.

Dalším z dílčích cílů je úprava aplikace. Dříve provedená architektonická a designová rozhodnutí mají za následek výskyt duplicitního kódu, těžkopádné rozšiřování a obecnou nemožnost dodržování dobrých principů při vývoji objektově orientovaných aplikací. Z těchto důvodů bude nástroj podroben refactoringu.

Posledním cílem je příprava programátorské příručky, ve které bude detailně (ale srozumitelně) popsáno, jak si pro nástroj vytvořit vlastní plugin. Návod bude následně poskytnut všem stávajícím zákazníkům spolu s novou verzí aplikace.

Kapitola 2

Analýza nástroje Winch

Kapitola seznamuje čtenáře s anonymizačním nástrojem Winch. Představuje jeho funkcionality a komponenty, které jsou důležité z pohledu cílů práce. Pro lepší představu o funkčnostech aplikace jsou také uvedené konkrétní příklady využití.

2.1 Anonymizační nástroj Winch

Nástroj Winch od společnosti GEM System a.s. slouží k anonymizaci a objevování osobních a citlivých údajů v relačních databázích. Aplikace je rozdělena na dva moduly – *Winch Add-in* a *Winch Actor*. Winch Add-in je *add-in* (rozšíření) pro nástroj Enterprise Architect (EA). Skrze něj se provádí konfigurace anonymizačního procesu, kterou následně čte Winch Actor při svém spuštění. Winch Actor je komponenta, která načítá informace o databázovém modelu z EA. Na základě těchto dat pak spouští proces anonymizace nebo objevování údajů přímo v relační databázi. [1]

Své využití nalézá nástroj Winch při tvorbě testovacích dat na základě dat produkčních. Vývojáři či testeři tak nemají přístup k údajům o uživatelích klientovy databáze, a výrazně se tak snižuje riziko úniku těchto citlivých informací. Winch lze mimo popsaný scénář využít také k anonymizaci dat, která už nejsou pro společnost aktuální nebo jinak relevantní. Takové informace může nástroj anonymizovat a uložit například do archivní databáze.

2.1.1 Proces anonymizace

Běžný proces anonymizace začíná v nástroji Enterprise Architect. Datový analytik (nebo jiná osoba odpovědná za data klientské společnosti) přenese databázový model ze zdrojového databázového stroje do EA, ve kterém následně provede konfiguraci anonymizačního procesu skrze Winch Add-in.

Pokud tabulka obsahuje sloupec, jenž nese osobní nebo citlivou informaci, přiřadí se mu patřičná anonymizační třída, která popisuje data ve sloupci.

Anonymizační třídu je třeba svázat s anonymizační funkcí, která provádí samotnou anonymizaci údaje. Anonymizační funkce jsou implementovány v transakčním rozšíření jazyka *Structured Query Language* (SQL), jež je obvykle specifické pro konkrétní databázový stroj. Aktuálně jsou připravené anonymizační třídy (a k nim patřičné anonymizační funkce) pro všechny typické osobní a citlivé údaje.

Není-li žádoucí, aby byl výsledný objem anonymizovaných dat stejný jako objem dat původních, lze na tabulce nastavit řez. Ten je realizován pomocí SQL klauzulí *where* (nutná podmínka, kterou musí anonymizovaná data splňovat) nebo *limit* (celkové omezení výsledného počtu řádků). Klauzuli *where* lze nastavit i na sloupci, podmínky jsou pak spojeny pomocí konjunkce.

Přímo ve Winch Add-inu lze spustit i dílčí funkcionality Winch Actora, které vedou k anonymizaci dat. Winch Actor se spouští z příkazové řádky, při exekuci na vstupu očekává globálně unikátní identifikátor (známý pod anglickou zkratkou *GUID*) aplikace, GUID kontextu a mód. Identifikátory jednoznačně určují objekty v EA, ze kterých si má Winch Actor načíst konfiguraci anonymizace – z tohoto důvodu musí během spuštění nástroje běžet instance Enterprise Architect s načteným datovým modelem.

Mezi módy (důležité pro proces anonymizace) patří `generate` (generování SQL kódu do souborů), `deploy` (nasazení kódů do databázového stroje) a `execute` (spuštění kódů v databázi). [1]

2.1.2 Proces objevování osobních a citlivých údajů

Při spuštění modulu Winch Actor v *discovery* módu dojde k analýze databázového modelu v EA. Během tohoto procesu jsou kontrolovány sloupce tabulek a na základě předem definovaných pravidel se určí pravděpodobnost, že daný sloupeček obsahuje osobní nebo citlivý údaj. Při vysoké šanci výskytu takové informace je uživatel vyzván ke kontrole sloupce. [1]

2.2 Technologie a implementace

Sekce představuje technologie užívané k rozvoji modulu Winch Actor a způsob, kterým je komponenta realizována.

2.2.1 Programovací jazyk Groovy

Groovy je programovací jazyk, který lze označit jako nadmnožinu jazyka Java. Je plně kompatibilní s Java platformou, protože je stejně jako Java kompilován do *bytecode*, kterému rozumí virtuální stroj *Java Virtual Machine* (JVM). Aplikace lze tak psát v obou jazycích najednou. Groovy přináší například dynamické typování, méně striktní syntaxi nebo podporu pro Domain-Specific jazyky. I přes upravená syntaktická pravidla Groovy stále plně podporuje syntaxi Javy. [2]

2.2.2 Nástroj Gradle

Gradle je sestavovací nástroj, který na rozdíl od starších (jako Maven nebo Ant), nabízí větší svobodu při psaní sestavovacích skriptů. Jeho vlastnost snadné přizpůsobitelnosti ale zároveň neimplikuje absenci pravidel a dobrých praktik, jimiž se může vývojář řídit. Gradle se od dalších sestavovacích nástrojů liší také jazykem, kterým se píšou sestavovací soubory (`build.gradle`) – využívá Groovy. Při psaní těchto skriptů ale není vyžadována žádná hlubší znalost tohoto programovacího jazyka. Užitím skriptovacího jazyku tak Gradle umožňuje snadnou úpravu sestav, tato varianta není ale obvykle doporučována. Pro případy, kdy je třeba upravit sestavovací skript nebo rozšířit možnosti nástroje, nabízí Gradle vlastní *domain specific language* (DSL). [3]

2.2.3 Moduly komponenty Winch Actor

Winch Actor je rozdělen do několika Gradle modulů. Hlavním modulem je `disl-winch-connector`, který představuje abstraktní kostru pro všechny databázové moduly. Mezi ně aktuálně patří:

- `disl-winch-db2`;
- `disl-winch-mssql`;
- `disl-winch-oracle`;
- `disl-winch-postgresql`.

Každý z databázových modulů poskytuje SQL kód specifický pro konkrétní databázový stroj (určený podle přípony v názvu modulu). Aktuálně ale nejsou všechny funkcionality dostupné ve všech modulech – například proces anonymizaci není podporován v modulu `disl-winch-postgresql`. V budoucnu se předpokládá přidání podpory i pro další databázové stroje.

Speciálním modulem komponenty je potom `disl-winch-license`, jenž se stará o generování licencí a klíčů pro klienty.

2.3 Sestavování a distribuce Winch Actora

Při sestavování komponenty Winch Actor se využívá speciálního Gradle pluginu pro distribuci aplikací (*Distribution Plugin*). Plugin během sestavování připraví složku (případně její komprimovanou verzi v podobě `zip` nebo `tar` souboru), která obsahuje sestavovaný modul včetně jeho závislostí (součástí složky je tak vždy modul `disl-winch-connector`). Mimo dalšího obsahu se v instalaci nachází i dva skripty (`.bat` pro platformu Windows a `sh` skript pro unixové systémy), které slouží k exekuci komponenty z příkazové řádky. Právě tyto skripty spouští Winch Add-in v EA.

2.4 Komponenty

Implementace Winch Actora obsahuje několik klíčových komponent, kterými se zabývají následující podkapitoly.

2.4.1 Anonymizační funkce

Anonymizační funkce představují základní stavební kámen pro anonymizaci dat. Každá funkce je zaměřena na anonymizaci určitého osobního údaje. U anonymizačních funkcí lze evidovat informace o jejich vlastnostech, jako například zda pro konkrétní vstup vrátí vždy stejný výstup nebo zda může vracet *null* hodnotu.

Příkladem může být anonymizační funkce, která na svém vstupu přijímá název města a okres, ve kterém se nachází. Je-li okres evidován, je původní město na vstupu nahrazeno názvem jiného města ze stejného okresu, čímž je zajištěna konzistence dat z pohledu validací, ale zároveň není zachováno jméno původního města.

2.4.2 Dekorátory

Dekorátory vrací SQL kód, který je aplikován na výstupy anonymizačních funkcí. Jejich cílem je dodatečná úprava anonymizovaných dat.

Lze si představit situaci, kdy chce klient anonymizovat příjmení klientů v tabulce, ale vyžaduje, aby byla všechna písmena ve výsledném rodinném jméně kapitalizována. Výsledkem anonymizačního procesu pro příjmení je náhodně vybrané jméno z patřičného slovníku (tedy pouze s velkým počátečním písmenem). Koncovému uživateli stačí k anonymizační třídě pro příjmení přiřadit správný dekorátor, jenž se před uložením do cílové tabulky postará o převedení všech písmen na velká.

2.4.3 Tabulkové vzory

Tabulkové vzory určují postup, jakým má být tabulka zpracovávána. Postup je ve vzoru reprezentován kolekcí kroků, jež jsou spouštěny v pořadí, ve kterém byly do kolekce vloženy. Kroky rozlišujeme podle jejich činnosti, která je udána jejich předponou – `Generate` kroky generují SQL kód do souborů, `Deploy` kroky nasazují generovaný kód do databázového stroje a `Execute` kroky spouští nasazený SQL kód v databázi.

Výchozí tabulkový vzor nejprve vygeneruje SQL skript, který se stará o vytvoření tabulky v anonymizovaném schéma. Následně připraví soubor s SQL kódem, jenž po spuštění vkládá anonymizovaná data do přichystané tabulky. Posledním generovaným souborem je skript, který tabulce přidá všechna omezení na cizí klíče, které obsahovala tabulka původní. Následují kroky, jež kód nasadí do databáze. Dalším krokem je exekuce nasazených skriptů v cílovém databázovém stroji. Po něm se spouští validace (`Validate`

krok), která v databázi ověřuje, že struktura nově vzniklé tabulky odpovídá struktuře tabulky původní. Navíc nad tabulkami spouští validační SQL výběry (poskytované v `Generate` krocích starajících se o vkládání dat do nových tabulek). Posledním krokem je potom spuštění procesu objevování osobních a citlivých údajů nad danou tabulkou.

2.4.4 Objevitelé

O vyhledávání osobních údajů v tabulkách se starají objevitelé, kteří při spuštění Winch Actora v *discovery* módu přiřazují sloupcům pravděpodobnost výskytu daného osobního údaje (určeného anonymizační třídou, kterou má sloupec přiřazenou).

Každý objevitel obsahuje seznam běžných názvů pro danou anonymizační třídu, jenž je kontrolován proti názvům sloupců v tabulce. Dále obsahuje typická klíčová slova, která se mohou vyskytovat v komentáři sloupečku, vyhovující datový typ pro daný osobní údaj a jeho minimální délku. Na základě těchto údajů a následných porovnání se sloupečky tabulky je určena pravděpodobnost, že sloupec obsahuje danou anonymizační třídu. Překročí-li pravděpodobnost jistou hranici, je uživatel informován o možném výskytu osobního údaje ve sloupci a vyzve jej k jeho kontrole.

2.4.5 Slovníky

Slovníky v nástroji existují pro podporu objevování a anonymizace. Každý slovník je ve Winch Actoru reprezentován třídou, jež ve svých metodách vrací například název souboru s výčtem dat (seznamy jmen, měst, náboženství a jiné) nebo jméno tabulky, kam jsou tato data nahrána v rámci inicializace anonymizačního procesu.

2.5 Export informací o dostupných komponentách

Winch Actor je realizován jako konzolová aplikace, všechny výpisy jsou tedy přenášeny na standardní výstup systému. Pokud Winch Add-in v nástroji Enterprise Architect vyžaduje data, spustí komponentu Winch Actor s patřičným příkazem a výstup exekuce si přečte. K úspěšnému načtení informací musí v obou komponentách existovat *data transfer object* (DTO) reprezentující patřičné objekty. Aktuálně v nástroji existuje DTO pouze pro anonymizační funkce. Informace o dostupných komponentách ve Winch Actoru jsou v textu označovány i jako tzv. „metadata“.

2.5.1 Formáty poskytovaných dat

Všechna data jsou z Winch Actora poskytována ve čtyřech formátech — *CSV*, *Confluence wiki*, *Markdown* a *JSON*.

Formát **CSV** (zkratka pro anglické *comma-separated values*) se hojně využívá při výměně dat mezi tabulkovými procesory. Jak z anglického názvu vypovídá, data jsou ve formátu CSV odděleny čárkami. Jeden záznam je reprezentován řádkem, každý další záznam je pak uváděn na řádku novém. Volitelně může CSV obsahovat i hlavičku popisující jednotlivé sloupce. Hlavička je vždy uváděna na prvním řádku. [4]

Confluence wiki je formát určený k formátování textu bez nutnosti uvádění značek. Podobně jako formát Markdown je i Confluence wiki zaměřen primárně na snadnou čitelnost pro člověka. Confluence wiki obsahuje rozdíly oproti klasickému Wiki formátu, jedním z příkladů může být zápis tabulek. Mluví-li se v textu práce o formátu Wiki, implicitně se jím rozumí formát Confluence wiki. [5]

Markdown je nástroj napsaný v programovacím jazyce Perl, který slouží ke konverzi lidsky čitelného textu do HTML. Markdown se zaměřuje především na snadnou čitelnost a snahu publikovat dokumenty v tomto formátu tak, jak jsou, bez značek či formátovacích instrukcí. [6]

JSON (*JavaScript Object Notation*) je formát určený k serializaci strukturovaných dat. Jeho nadmnožinou je programovací jazyk Javascript. JSON si z něj přebírá struktury jako objekt (neuspořádaná kolekce), který obsahuje 0 až N párů klíč – hodnota, nebo pole hodnot (uspořádaná kolekce). Formát podporuje hodnoty v podobě primitivních datových typů - řetězce, čísla, boolovské hodnoty true či false a null. [7]

2.5.2 Aktuální využití jednotlivých formátů

V momentální implementaci je velmi důležitým formátem JSON. Ten se využívá k přenosu informací mezi komponentou Winch Actor a Winch Add-inem v EA.

Dalšími aktivně využívanými formáty jsou Confluence wiki a Markdown. Export dat v těchto formátech připomíná tabulku slouženou z ASCII znaků. Metadata lze zkopírovat přímo ze standardního výstupu a vložit je do dokumentačního nástroje Confluence, jenž umí tabulky v obou formátech graficky interpretovat.

CSV formát je dnes využíván minimálně. Data v něm nicméně umožňují snadný import do tabulkových procesorů, kde mohou být nad poskytnutými informacemi prováděny další operace.

Výpis dat ve výše popsaných formátech lze získat spuštěním komponenty Winch Actor s patřičnými příkazy a přepínači. Do formátu JSON se data formátují i interně při zmiňované komunikaci s Winch Add-inem.

Každý příkaz popsaný v tabulce 2.1 může být následován jedním z parametrů představující formát exportovaných dat, tedy CSV, Wiki, Markdown nebo JSON (parametry CSV, WIKI, MARKDOWN nebo JSON). Není-li parametr uveden, vypíší se data ve formátu JSON.

Příkaz	Popis	Příklad
-m	Vypíše dostupné anonymizační funkce.	-m WIKI
-tp	Vypíše dostupné tabulkové vzory.	-tp CSV

Tabulka 2.1: Exportní příkazy komponenty Winch Actor

2.5.3 Realizace poskytování metadat

Anonymizační funkce: Komponenta Winch Actor umožňuje získat seznam všech dostupných anonymizačních funkcí. Tento výpis je realizován dynamicky, při přidání nové anonymizační funkce tak není nutné zasahovat do části kódu zajišťující výstup těchto metadat.

Informace o konkrétní anonymizační funkci jsou uvedeny v anotacích každé třídy reprezentující anonymizační funkci. První anotace nese informace v podobě řetězce, kde jsou dílčí data (sloupce) oddělena svislou čárou. V druhé anotaci se pak vyplňují dvě pole, první obsahuje názvy parametrů anonymizační funkce a druhé popisy těchto parametrů (v pořadí, v jakém byly parametry uvedeny v prvním poli). Implementace anonymizační funkce pro konkrétní databázový stroj je pak navíc obohacena o anotaci, do níž se vkládají názvy vyžadovaných SQL souborů (tyto informace ale nejsou součástí výpisu).

Tabulkové vzory: Výpis informací o tabulkových vzorech není realizován dynamicky. V případě rozšíření nástroje o další tabulkový vzor je tak nutno upravit i část kódu, které se stará o poskytování informací o nich.

Dekorátory: Ve stávající implementaci neumožňuje Winch Actor podávat informace o dostupných dekorátorech.

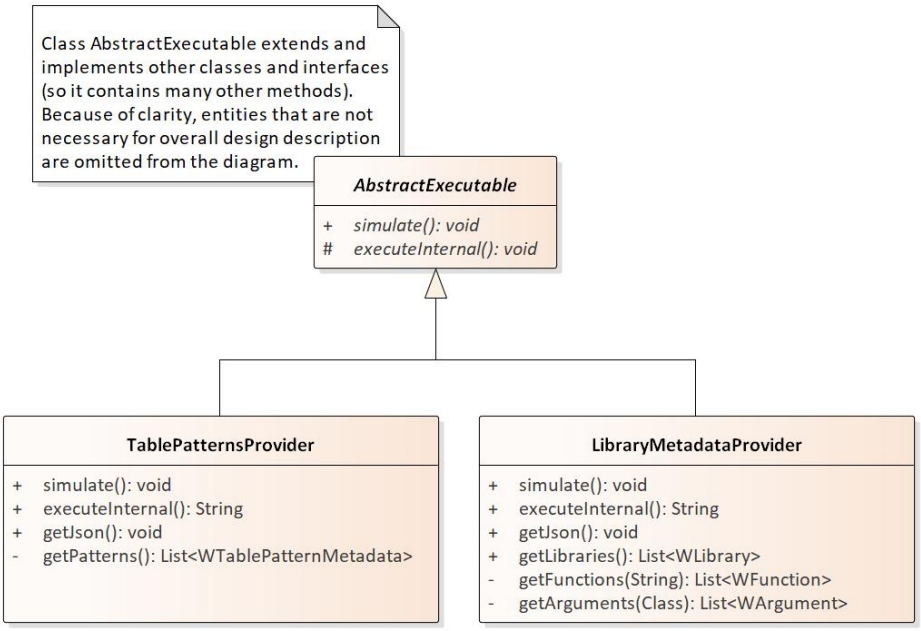
Objevitelé: Ve stávající implementaci neumožňuje Winch Actor podávat informace o dostupných objevitelích.

Slovníky: Ve stávající implementaci neumožňuje Winch Actor podávat informace o dostupných slovnících.

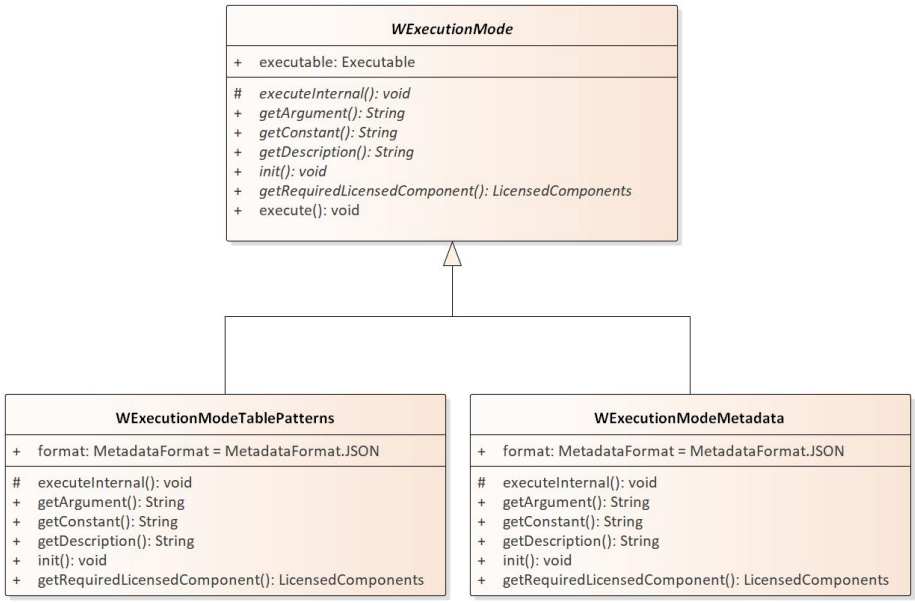
Struktura zachycená class diagramem na obrázku 2.1 popisuje aktuální stav poskytovatelů metadat. I z diagramu je zřejmé, že jedinými poskytovanými informacemi jsou seznamy anonymizačních funkcí a tabulkových vzorů.

Poskytovatelé metadat se starají o export informací na standardní výstup. Jejich spouštění (a spouštění dalších potomků třídy `AbstractExecutable`) zajišťují potomci třídy `WExecutionMode` – v tomto případě se konkrétně jedná o třídy `WExecutionModeTablePatterns` a `WExecutionModeMetadata`. Class diagram popisující částečnou strukturu exekučních módů (souvisejících s poskytováním metadat) je zachycen na obrázku 2.2.

Nedostupnost informací o některých komponentách aplikace s sebou přináší úskalí především z pohledu vedení dokumentace nástroje a informovanosti koncových uživatelů. Mají-li být například dokumentovány seznamy jednotlivých komponent, musí autor dokumentace nahlížet do zdrojových kódů komponenty Winch Actor. Protože zatím není v nástroji implementován žádný systém



Obrázek 2.1: Aktuální stav poskytovatelů metadat



Obrázek 2.2: Aktuální stav exekučních módů

rozšiřování, není klient zásadně dotčen absencí informací o některých komponentách. Pokud by ale nebylo poskytování metadat rozšířeno i na další části aplikace, potýkal by se programátor při tvorbě pluginu s nepříjemnostmi. Například při realizaci nové anonymizační funkce by neměl žádné informace o dostupných slovnících, což by mohlo vést ke značnému omezení při tvorbě anonymizačních funkcí a zbytečnému přidávání práce koncovému uživateli, který by si například již existující výčty musel vytvářet sám.

2.6 Aktuální možnosti rozšiřování nástroje

Ve stávající implementaci nelze nástroj Winch rozšiřovat jinak než samotnou úpravou zdrojového kódu aplikace. Pokud chce tedy klient přidat komponentu (například nový specifický dekorátor, který by rád využil k úpravě výstupu anonymizačních funkcí), musí o rozšíření nástroje požádat společnost samotnou.

Kapitola 3

Požadavky

Kapitola se zabývá analýzou požadavků, která vychází primárně ze zadání a postupného zkoumání nástroje Winch.

3.1 Analýza požadavků na poskytování metadat

Přímo ze zadání práce nevyplývají žádné konkrétní požadavky na poskytování informací o dostupných komponentách. Vstupem pro tvorbu požadavků na export metadat je tak analýza nástroje z pohledu rozšiřování a konzultace se zadavatelem.

Tabulka 3.1 uvádí výčet požadavků, které jsou kladeny na poskytování dat o dostupných komponentách na standardní výstup. Každý požadavek je pak popsán v patřičné podkapitole.

Požadavek	Kód požadavku
Poskytování informací o dalších komponentách	M1
Minimalizace duplicit v poskytovaných datech	M2
Komunikace s Winch Add-Inem	M2.1
Snadný export do Confluence	M2.2
„Human readable“ podoba dat	M2.3
Dynamické načítání dostupných komponent	M3

Tabulka 3.1: Požadavky na poskytování metadat

3.1.1 M1 – Poskytování informací o dalších komponentách

Z analýzy komponenty Winch Actor vyplývá, že aktuálně lze získávat informace pouze o anonymizačních funkcích a tabulkových vzorech. V rámci rozšiřování aplikace by měl Winch Actor nabízet data o:

- anonymizačních funkcích;
- tabulkových vzorech;
- dekorátorech;
- objevitelích;
- a slovnících.

3.1.2 M2 – Minimalizace duplicit v poskytovaných datech

Vzhledem k faktu, že se po dobu existence nástroje Winch využívají reálně pouze dva formáty (JSON a Confluence wiki), může být celkový počet dostupných formátů snížen.

3.1.3 M2.1 – Komunikace s Winch Add-Inem

Winch Actor musí poskytovat informace o dostupných komponentách v takovém formátu, kterému bude Winch Add-in v EA po jejich přečtení ze standardního výstupu rozumět, a bude je tak následně schopný interpretovat uživateli ve svém grafickém rozhraní.

3.1.4 M2.2 – Snadný export do Confluence

Společnost využívá dokumentační nástroj Confluence k zachycení technického i uživatelského popisu aplikace Winch. Zaznamenávají se do něj mimo jiné i výčty dostupných komponent (například seznam anonymizačních funkcí), je tedy proto nezbytné, aby byl v rámci realizace požadavku M2 – Minimalizace duplicit v poskytovaných datech zachován takový formát, který umožní snadný export těchto informací do Confluence.

3.1.5 M2.3 – „Human readable“ podoba dat

Při práci s nástrojem v příkazové řádce by měla být poskytovaná data čitelná. Čitelností se rozumí export informací v takovém formátu, který není technického rázu, a bude co nejvíce přirozený pro člověka (například výpis do tabulky, formátovaného stromu nebo jiné struktury).

3.1.6 M3 – Dynamické načítání dostupných komponent

Pokud dojde k přidání nové komponenty, musí být tato komponenta zahrnuta ve výpisu bez nutnosti úpravy kódu, která se o výstup stará.

3.2 Požadavky na rozšiřování nástroje

Požadavky na tvorbu pluginovacího systému vychází primárně ze zadání práce.

Požadavek	Kód požadavku
Rozšiřitelnost nástroje	W1
Dynamičnost rozšíření	W1.1
Souběžná existence několika rozšíření	W1.2
Programátorská příručka	W2

Tabulka 3.2: Požadavky na rozšiřování nástroje Winch

3.2.1 W1 – Rozšiřitelnost nástroje

Pluginovací systém by měl umožňovat rozšiřování nástroje Winch o:

- anonymizační funkce;
- tabulkové vzory;
- a dekorátory.

3.2.2 W1.1 – Dynamičnost rozšíření

Při dodání rozšíření nesmí být vyžadována rekompilace Winch Actora nebo Winch Add-inu. Ať už rozšíření přidá koncový uživatel nebo společnost samotná (na základě požadavku od klienta), musí aplikace načíst a umět pracovat s novými funkcionalitami i ve stavu, v jakém byl nástroj původně dodán klientovi.

3.2.3 W1.2 – Souběžná existence několika rozšíření

Pro nástroj Winch může být vytvořeno několik pluginů. Aplikace musí umět načíst všechna dodaná rozšíření a nabídnout klientovi veškeré dodané komponenty k užití.

3.2.4 W2 – Programátorská příručka

Pro hotový pluginovací systém by měla vzniknout programátorská příručka, která bude koncové uživatele (případně vývojáře samotné) provázet postupem tvorby a dodání pluginů.

Kapitola 4

Rešerše

Tato kapitola obsahuje nutný teoretický základ pro zkoumání možností rozšiřování aplikací na platformě Java. Dále čtenáři přibližuje návrhové vzory, které mohou být využity během fáze návrhu a realizace.

4.1 Java platforma

Všechna níže uvedená specifika jsou platná pro Javu ve verzi 8. Jedná-li se o informace platné pouze pro určitou systémovou platformu, pak je za výchozí platformu považován operační systém Windows, kvůli aktuální závislosti nástroje Winch na nástroji Enterprise Architect, který je dostupný pouze pro právě zmiňovaný systém Windows. [8]

4.1.1 Java Virtual Machine (JVM)

Java Virtual Machine (JVM) je základní komponentou Java platformy, která se stará o odstranění závislosti mezi hardwarem a operačním systémem. Jak z názvu vypovídá, JVM je virtuální stroj, který disponuje vlastní sadou instrukcí a manipuluje s pamětí za běhu systému. JVM může běžet na různých architekturách – stačí virtuální stroj zkompilovat na konkrétním procesoru. Java Virtual Machine nemá nic společného se samotným jazykem Java. Rozumí pouze binární podobě zkompilovaného kódu, tzv. *bytecode*, uloženého v souborech s koncovkou `.class`. Kromě abstrakce mezi hardwarem a operačním systémem zajišťuje JVM i bezpečný běh aplikací pomocí silných syntaktických a strukturálních omezení v `.class` souborech. [9]

4.1.2 JRE, JDK a JAR

Java Runtime Environment (JRE) poskytuje všechny nutné komponenty pro běh Java aplikací. Hlavními částmi jsou knihovny a Java Virtual Machine. Společně s aplikací může být distribuováno i JRE.

Java Development Kit (JDK) obsahuje JRE a k němu i další nástroje potřebné pro vývoj aplikací v Javě. Jmenovitě se jedná například o kompilátor nebo debugger.

Java aplikace se obvykle skládá z několika `.class` souborů a mimo jiné může navíc pro svůj běh vyžadovat i jiné prostředky (například obrázky nebo audio). Všechny tyto soubory mohou být agregovány do jednoho souboru formát Java Archive (zkráceně JAR). Výhodou formátu je sjednocení všech potřebných souborů na jedno místo. JAR navíc podporuje kompresi nebo digitální podpisy pro autentizaci autora archivu. [10]

4.1.3 Class path

Class path lze popsat jako cestu, kde JRE hledá třídy a zdroje. Class path lze nastavit pomocí přepínače `-cp` či ekvivalentní `-classpath` za jedním z příkazů ze sady nástrojů *JDK Tools and Utilities* nebo pomocí proměnné prostředí `CLASSPATH`. Na class path lze nastavit cesty ke konkrétním JAR nebo zip souborům, složkám nebo zkompilovaným třídám samotným (soubory s koncovkou `.class`). Formát uvedení cesty se liší podle prostředku, který má být na class path vložen – v případě JAR a zip archivů se uvádí cesta k souboru, pro třídy, které nepatří do žádného balíčku se píše cesta přímo ke třídě a v poslední řadě, pokud se jedná o třídu zařazenou do balíčku, uvádí se cesta ke složce, která představuje kořenový balíček.

Na pořadí uvedených prostředků záleží, při načítání konkrétní třídy začíná prohledávání od prvního záznamu na class path. V případě nenalezení požadované třídy se pokračuje hledáním v další uvedené cestě.

Záznamy na class path podporují syntaxi s wildcard znakem (*). Při uvedení složky s wildcard (například `slozka/*`) se ze složky `slozka` načtou všechny soubory s koncovkou `.jar`, které se v ní nachází. Uvedená složka s wildcard není procházena rekurzivně. [11]

4.1.4 Classloading v Javě

Při spuštění Javy pomocí nástroje `java` se spustí JVM. Jako první se hledají a načítají tzv. bootstrap třídy, které jsou nezbytné pro běh jakékoliv aplikace psané v Javě. Třídy se načítají z JAR souborů (například z `rt.jar` a i několika dalších ze složky `jre/lib`). Jako další se načítají *extension* třídy, které využívají mechanismus *Java Extension*. Tyto třídy se automaticky načítají z `.jar` nebo `.zip` souborů umístěných ve speciální složce pro rozšíření `jre/lib/ext`. Jako poslední se načítají uživatelské třídy. Sem patří třídy, které poskytuje vývojář nebo třetí strany, a které nevyužívají rozšiřovacího mechanismu Javy. Uživatelské třídy se hledají na class path. [12]

4.1.5 Reflexe

Reflexe umožňuje Java aplikaci přistupovat k vlastnostem, metodám a konstruktorům načtených tříd. Takto zpřístupněné prvky třídy pak mohou být používány podobně jako jejich objektové protějšky. [10]

4.2 Extension Mechanism

Java Extension Mechanism je funkcionalita JRE, která umožňuje načítání tzv. volitelných balíčků (anglicky *optional packages*). Volitelný balíček představuje několik Java balíčků sjednocených do jednoho nebo více JAR souborů, jejichž cílem je rozšířit platformu Java. Tyto volitelné balíčky rozšiřují platformu tak, že načtení tříd a ostatních prostředků není podmíněno jejich přítomností na class path aplikace. Protože volitelné balíčky rozšiřují jádro celé platformy, jejich využití by mělo být vždy řádně zváženo. Má-li být rozšiřována jedna nebo pouze několik málo aplikací, nemusí být rozšiřování samotné platformy nejvhodnější řešení.

Volitelné balíčky jsou viditelné všemi Java procesy, jejich pojmenování a umístění by tak mělo následovat klasické jmenné a hierarchické konvence. Implementace balíčků sestává z Java kódu, rozšiřovací mechanismus nicméně neomezuje vývojáře pouze na rozšiřování na aplikační úrovni. JRE lze rozšiřovat i nativním kódem, jenž je specifický pro platformu, na které JRE běží.

Volitelný balíček může být dodáván s aplikací nebo instalován přímo do JRE. V prvním případě jsou balíčky poskytovány ve stejném stylu jako aplikace samotná. Pokud se jedná o síťovou aplikaci, je tento druh balíčků automaticky stahován, z čehož plyne i jejich název *download optional packages*. Tento druh volitelných balíčků ale není z pohledu rozšiřování konzolových aplikací zajímavý. Jsou-li volitelné balíčky instalovány přímo do JRE, nazývají se *installed optional packages*. Takové balíčky jsou dostupné všem aplikacím, které běží na daném JRE. [13]

4.2.1 Installed optional packages

Installed optional packages se do jisté míry podobají třídám načítaným při startu JVM (nemusí se nacházet na class path aplikace). Jak bylo uvedeno v kapitole Classloading v Javě, instalované balíčky musí být umístěny ve složce `jre/lib/ext`. Výchozí umístění instalovaných balíčků může být změněno pomocí systémové proměnné `java.ext.dirs`. [13]

Pro další pokračování práce jsou informace o standardním rozšiřování Java aplikací podstatné. Ačkoliv nebyly ve výsledné realizaci použity, bez načerpání patřičných znalostí o volitelných balíčcích mohlo dojít k jejich mylnému použití pro pluginovací systém.

4.3 Návrhové vzory

Návrhové vzory pomáhají řešit problémy, které se velmi často během vývoje objektově orientované aplikace vyskytují. Základní myšlenkou každého návrhového vzoru je znovupoužitelnost a dobrá rozšiřitelnost. Návrhové vzory jsou koncipovány tak, aby nebyly závislé na konkrétní vlastnosti programovacího jazyka. Výjimkou je pouze objektová orientovanost. Následující podsekce přibližují vybrané návrhové vzory, které by mohly být využity v praktické části práce. [14]

4.3.1 Stavitel

Návrhový vzor stavitel (anglicky *builder pattern*) umožňuje provést granulaci procesu tvorby komplexních objektů s mnoha parametry. Zaměřuje se na konstrukci objektu vlastnost po vlastnosti, takže stejný proces stavby objektu může pokaždé vrátit objekt odlišný. Blízkým návrhovým vzorem je abstraktní továrna. Od něj se ale liší právě zmiňovanou granularitou během procesu tvorby objektu. [14]

4.3.2 Tovární metoda

Tovární metoda (anglicky *factory method*) umožňuje vytvářet objekty bez znalosti cílové třídy. Zavoláním metody továrny se zvolí její správný potomek, který je vrácen, případně je objekt vracen přímo továrnou samotnou. [14]

4.3.3 Abstraktní továrna

Abstraktní továrna (anglicky *abstract factory*) je návrhový vzor, který úzce souvisí s tovární metodou, která je součástí továren. Činnost abstraktní továrny spočívá právě ve výběru vhodné továrny, a tedy další delegaci během procesu tvorby objektů. Klient nemusí znát konkrétní třídu, a tak může nechat odpovědnost za dodání správného objektu právě na továrně, kterou poskytne abstraktní továrna. [14]

Kapitola 5

Návrh a realizace

Kapitola se zabývá zhodnocením existujících řešení popsaných v kapitole Rešerše vůči analyzovaným požadavkům z části Požadavky. Následně jsou pak popsány návrhy a realizace řešení.

5.1 Plán rozvoje

Před samotným návrhem a implementací pluginovacího systému je nutné vytvořit rozhraní pro poskytování potřebných metadat. Metadaty se rozumí informace o konkrétních komponentách aplikace. Uživatel si tak po dodání pluginu může ověřit, že přidané funkcionality jsou nástrojem skutečně načítány. Mimo jiné lze poskytované seznamy dostupných funkcionalit využít i na jiných místech (dokumentace nebo uživatelské a programátorské příručky).

Po implementaci a otestování části, která se bude starat o poskytování dostupných komponent, začne probíhat analýza rozšiřování nástroje, následný návrh a realizace spojená s nutným refactoringem kódu. Samozřejmostí je rovněž konečné testování pluginovacího systému podle programátorské příručky.

5.2 Poskytování metadat

Následující podsekce se věnují návrhu a realizaci úprav spojených s poskytováním informací o dostupných komponentách.

5.2.1 Refactoring exekučních módů

Exekuční módy se v nástroji starají o spouštění veškerých funkčností aplikace. Při startu aplikace jsou přečteny vstupní parametry, na základě kterých se vybere patřičný exekuční mód. Ten je pak zodpovědný za spuštění správné funkčnosti nástroje. Exekuční mód také vrací druh licence, jež je ke spuštění dané funkčnosti vyžadována.

Spouštění činnosti poskytovatelů metadat je tedy řízeno exekučními módy. Analytická část představila dva (jeden pro spouštění poskytovatele informací o anonymizačních funkcích a druhý pro tabulkové vzory). Protože mají být (podle požadavku M1 – Poskytování informací o dalších komponentách) přidány výpisy i pro další komponenty, je třeba rozšířit i množinu exekučních módů. Ty by mohly být přidány stejným způsobem, jako existující, což by ale vedlo k tvorbě duplicitního kódu – poskytování metadat není například podmíněno vlastněním speciální licence, tato informace je tak vracena všemi exekučními módy, které se starají o spouštění poskytovatelů metadat.

Společné rysy a chování mohou být abstrahovány do předka, čímž vznikne přehlednější hierarchie tříd. Nová struktura exekučních módů je zachycena obrázkem 5.1.

5.2.2 Refactoring poskytovatelů metadat

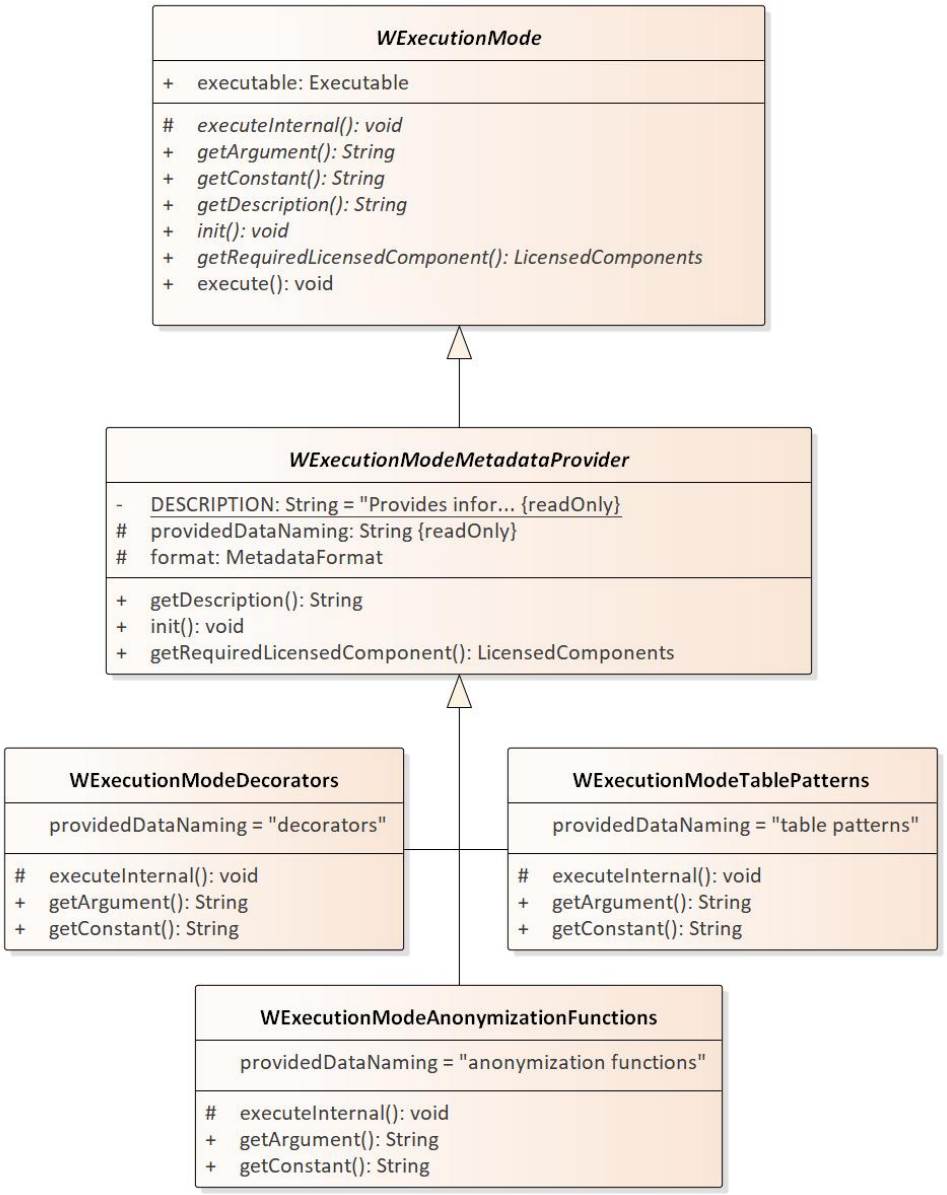
Poskytovatelé metadat spolu sdílí několik společných rysů, které je (vzhledem k rostoucímu počtu zmiňovaných poskytovatelů) vhodné abstrahovat do předka.

První varianta nové architektury je popsaná class diagramem na obrázku 5.2. Umožňuje snadné přidání dalšího poskytovatele metadat. Ten musí dědit z nové třídy `DataProvider` a tranzitivním kontraktem implementovat všechny metody rozhraní `Printable`. Při vzniku nového formátu je situace komplikovanější, neboť je nutno rozšířit výčtový typ `MetadataFormat`, následně přidat novou metodu do rozhraní `Printable` a tuto metodu pak implementovat v každém potomku třídy `DataProvider`.

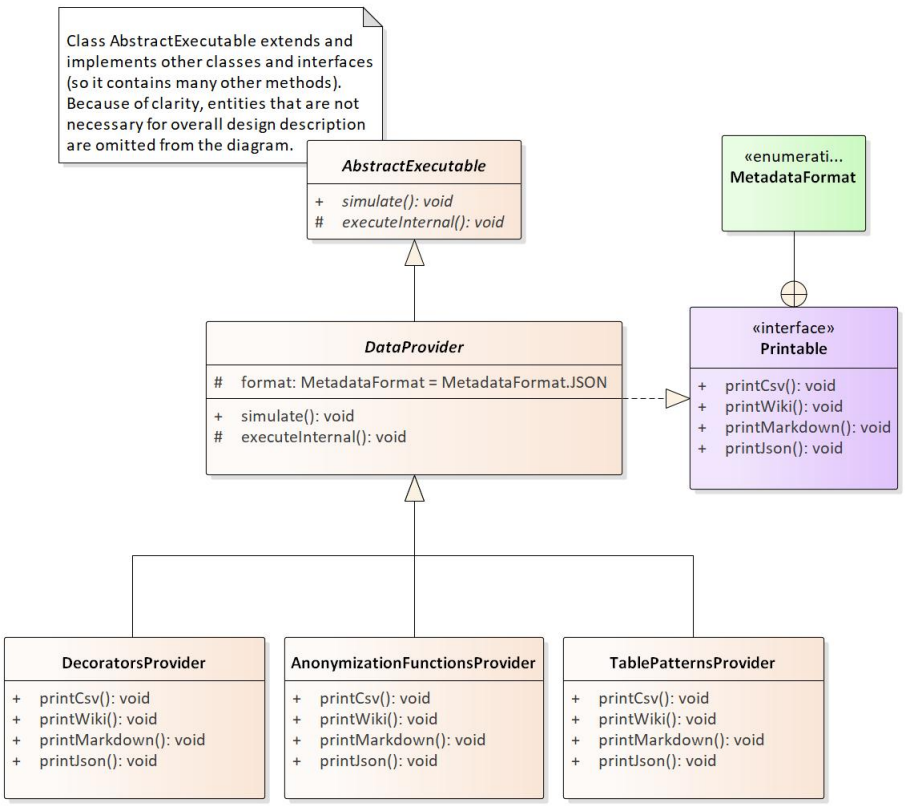
Export dat ve formátu JSON je zajištěn třídou obsaženou přímo v jazyce Groovy (`groovy.json.JsonOutput`). Výstup informací v ostatních formátech je implementován ručně a svým vzhledem připomínají tabulku. Odlišnosti spočívají především v ohraničení na koncích tabulky a oddělovačích jednotlivých sloupců. Řádky jsou ve všech formátech odděleny pouze znakem pro nový řádek. Počet sloupců tabulky závisí na počtu evidovaných vlastností exportované komponenty.

Vzhledem k podobnosti tabulkových formátů se nabízí možnost abstrahovat jejich tvorbu. Variabilitu v počtu a typech sloupců může zajistit návrhový vzor stavitel. Prototyp na vytváření tabulek zachycuje obrázek 5.3.

Samotná tabulka je vytvářena pomocí vnořené stavitelské třídy `Table::TableBuilder`. Jiným způsobem nelze instanci třídy `Table` vytvořit, což je zajištěno privátním konstruktorem. Stavitelská třída umožňuje přidání hlavičky metodou `Table::TableBuilder::withHeader` a libovolného počtu sloupců pomocí metody `Table::TableBuilder::withColumn`. Každý vložený sloupec se vkládá do kolekce `Table::columns`. Po vytvoření požadovaného počtu sloupečků se tabulka vygeneruje metodou `Table::TableBuilder::build`. Do tabulky lze vkládat řádky pomocí metody `Table::insertRow`, která přijímá proměnlivý počet argumentů. Implementace metody zajistí konzistenci mezi počtem argumentů a celkovým počtem sloupců v tabulce. Celou tabulku lze pak



Obrázek 5.1: Návrh na novou architekturu exekučních módů

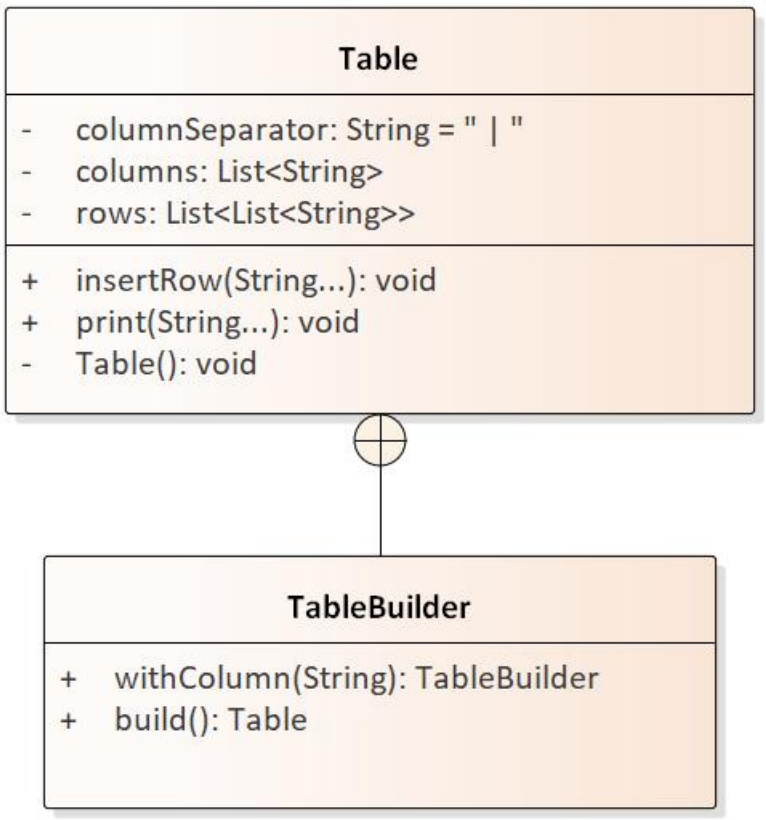


Obrázek 5.2: Návrh na novou architekturu poskytovatelů metadat

vypsat metodou `Table::print`, jejíž implementace se stará o dodání dalších částí tabulky (například oddělovače na koncích řádků). Metoda `Table::print` taktéž přijímá proměnlivý počet argumentů, které představují názvy jednotlivých sloupců. Je-li metoda volána bez argumentů, vypíše se tabulka bez hlavičky.

5.2.3 Minimalizace duplicitních výpisů

V optimálním stavu by nástroj Winch mohl exportovat data pouze v jednom formátu, který bude splňovat všechny definované požadavky. Druhý návrh na novou architekturu poskytovatelů metadat se tak ubírá směrem minimalizace formátů, v nichž jsou informace podávány. Aktuální stav obou komponent (Winch Actor a Winch Add-In) klade na existující formáty jistá omezení. Protože je ke komunikaci mezi Winch Actorem a Winch Add-inem využíván formát JSON, a úprava samotného add-inu v EA je mimo rámec této práce, musí být právě tento formát zachován. JSON tak splňuje požadavek M2.1 – Komunikace s Winch Add-Inem a je zároveň prvním kandidátem pro



Obrázek 5.3: Tvorba tabulek pomocí návrhového vzoru builder

nahrazení všech formátů.

Dalším požadavkem na formáty je M2.2 – Snadný export do Confluence. Webový dokumentační nástroj Confluence umožňuje zobrazovat data v tabulce. Tato data mohou být importována ve formátu Confluence wiki nebo Markdown. Pro import dat ve formátu JSON a následném „human readable“ zobrazení importovaných dat je třeba do nástroje nainstalovat makro (případně plugin), která jsou běžně nabízena komunitou nebo jinými společnostmi. Tato makra jsou ve většině případech poskytována za paušální poplatek, který se odvíjí od koncového počtu uživatelů nástroje Confluence. Při odebrání všech formátů kromě JSONu by tak vznikly požadavky vypsané níže, kde by alespoň jeden z nich musel být splněn, aby došlo k naplnění požadavku M2.2 – Snadný export do Confluence:

- Zakoupení makra / pluginu do nástroje Confluence.
- Dodání či vytvoření vlastního makra, které by umožňovalo načítání dat z formátu JSON.

- Dodání či vytvoření nástroje pro převod dat ve formátu JSON na formát Confluence wiki nebo Markdown, se kterými umí nástroj Confluence nativně pracovat.

Poslední z požadavků na formáty je M2.3 – „Human readable“ podoba dat. Ze všech aktuálně dostupných formátů je pro tento účel JSON nejméně vhodný, neboť jeho neformátovaná podoba (tedy JSON v jednom řádku) je pro člověka téměř nečitelný a formátovaný výstup je příliš rozsáhlý pro výpis do příkazové řádky. Ostatní formáty (Confluence wiki, Markdown a CSV) jsou pro naplnění tohoto požadavku akceptovatelné.

	M2.1	M2.2	M2.3
JSON	✓	✗	✗
Wiki	✗	✓	✓
Markdown	✗	✓	✓
CSV	✗	✗	✓

Tabulka 5.1: Srovnání formátů poskytovaných metadat podle požadavků

Z tabulky 5.1 je zřejmé, že původní kandidát JSON nemůže být jediný dostupný formát v nástroji Winch. Zároveň se ale jedná o jediný formát, který splňuje požadavek M2.1 – Komunikace s Winch Add-Inem, a musí proto zůstat zachován. Pro udržení minimálního počtu dostupných formátů je třeba vybrat další formát, který pokrývá ostatní požadavky. CSV splňuje nejméně požadavků, a není tak vhodným doplňkem k formátu JSON. Zbývají tak formáty Confluence wiki a Markdown, které splňují stejné požadavky. Oba si jsou velmi podobné, a může tak být vybrán jakýkoliv z nich. Subjektivně působí Confluence wiki formát estetičtěji, pro konečnou implementaci by tak mohly být vybrány formáty JSON a Confluence wiki.

5.2.4 Finální návrhy struktur

Vzhledem k upřesnění požadavku na minimalizaci duplicit v poskytování metadat, se první návrh architektury jeví technicky nadbytečný. I když by nenaplnění požadavku M2 – Minimalizace duplicit v poskytovaných datech nezpůsobilo problémy při navrhování a realizaci pluginovacího systému, vznikl by v aplikaci další kód, jenž by nenacházel patřičná uplatnění při běžném používání nástroje. Výsledně byla tedy vybrána druhá varianta návrhu nové architektury poskytovatelů metadat.

Ačkoliv byla první varianta návrhu zamítnuta, mohou být některé její myšlenky stále aplikovány. Návrh na abstrakci společných informací u exekučních módů, které následně spouští činnost poskytovatelů metadat, může být zachován. Kromě exekučního módu pro spuštění exportu dat o dostupných dekorátorech přibudou ještě další exekuční módy, konkrétně exekuční módy pro spuštění výpisu dat o dostupných objevitelích a slovnících. Výsledný

Komponenta	Abstrakce	Typ
anonymizační funkce	`AnonymizationFunction`	abstraktní třída
tabulkový vzor	`WTablePatternAbstract`	abstraktní třída
dekorátor	`Decorator`	rozhraní
objevitel	`AnonymizationClassDiscoverer`	abstraktní třída
slovník	`AbstractDictionary`	abstraktní třída

Tabulka 5.2: Abstraktní předkové a rozhraní

návrh struktury exekučních módů pro spouštění exportů je zachycen na obrázku 5.4.

Protože byla vybrána druhá varianta návrhu, snížil se celkový počet formátů, ve kterých jsou metadata poskytována (na množinu formátů JSON a Confluence wiki). Pro serializaci dat do formátu JSON se využívá interní knihovna jazyka Groovy. Pro jediný tabulkový formát (Confluence wiki) již není nutné zavádět technicky náročnější proces tvorby tabulek pomocí návrhového vzoru stavitel. Povinnost podávat informace o komponentách udává poskytovatelům metadat jejich společný předek, jak je znázorněno na obrázku 5.5.

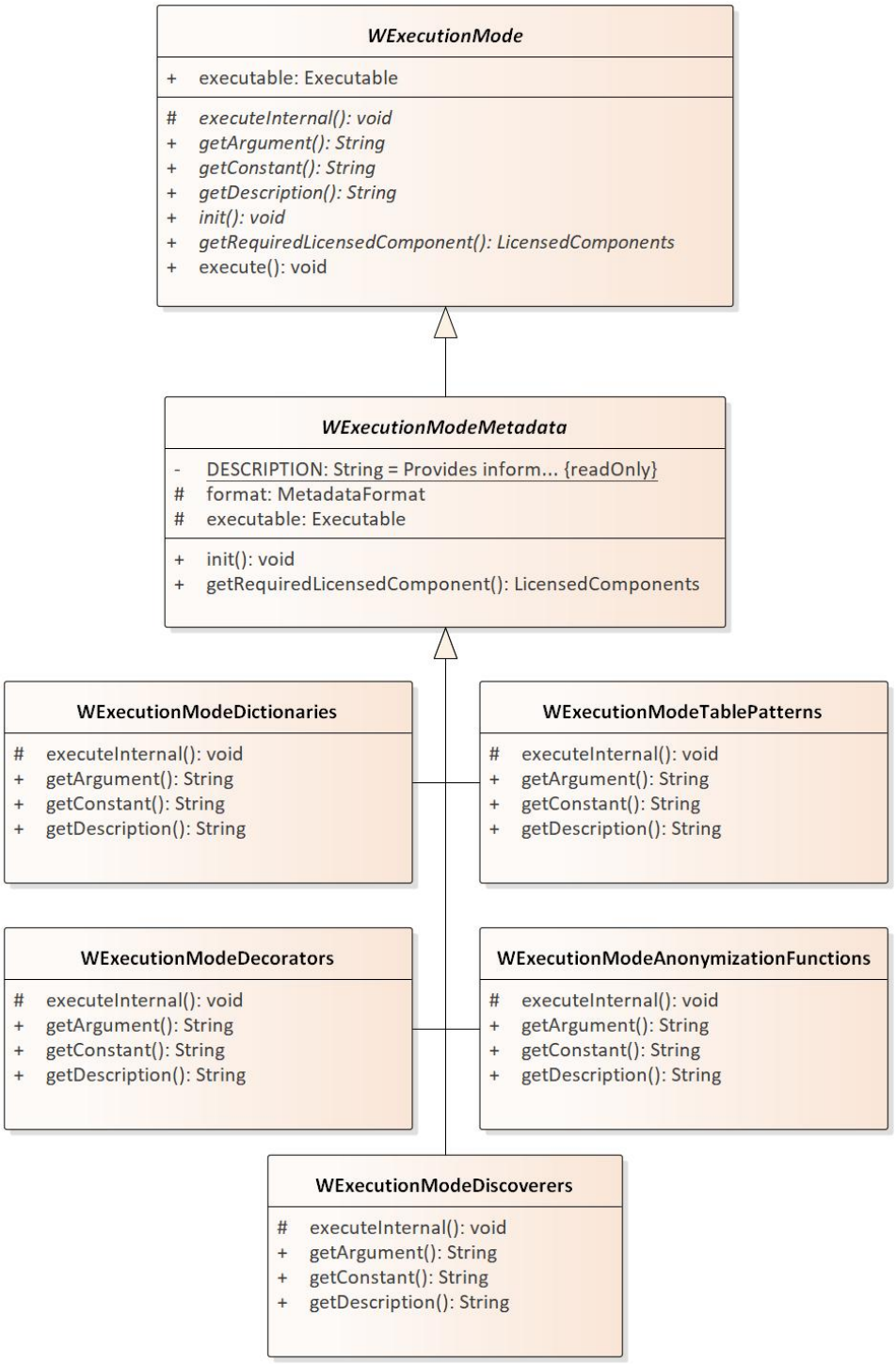
5.2.5 Dynamické načítání komponent

Každý poskytovatel si kromě metod na class diagramu 5.5 navíc implementuje ještě několik pomocných privátních metod, které se starají o dynamické načítání patřičných komponent. K načítání tříd je využívána reflexe. Každá z funkcionalit se nachází ve speciálním balíčku, což je prvotní místo, odkud probíhá načítání tříd. Protože se ale může stát, že jiná třída bude nesprávně umístěna do balíčku, do kterého sémanticky nepatří, musí být po načtení tříd z balíčku provedena dodatečná kontrola každé třídy.

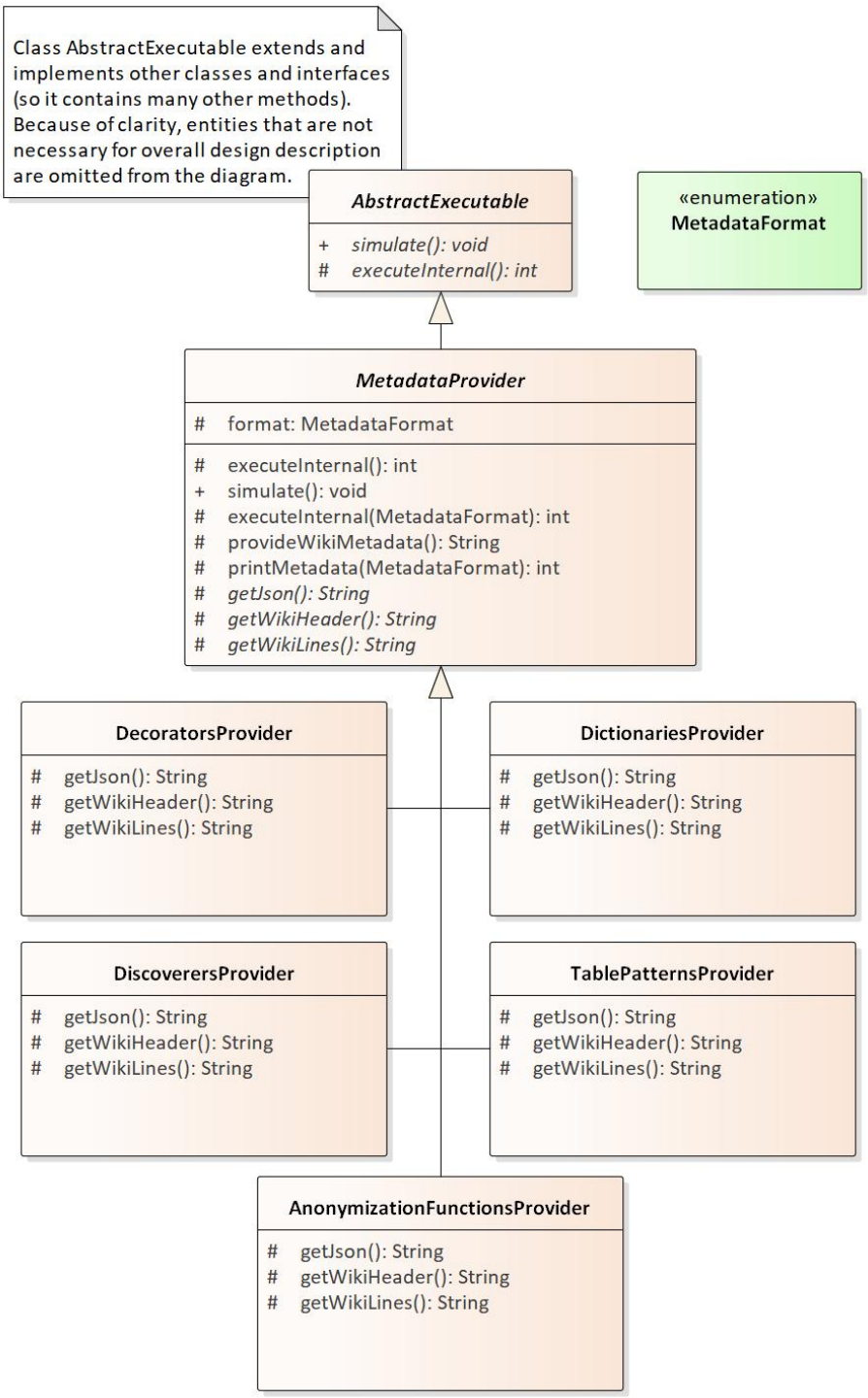
První z validací může být kontrola předka nebo implementovaného rozhraní. Při načtení třídy pomocí reflexe je přístup k těmto informacím velmi snadný. Protože všechny vypisované komponenty vždy dědí z nějakého abstraktního předka nebo implementují nějaké obecné rozhraní (viz tabulka 5.2), mohou být tyto informace využity ke kontrole, že načtené třídy jsou opravdu ty, které je třeba vypsat.

Další z validací mohou být vlastní anotace, které jsou aktuálně použity u anonymizačních funkcí. Kromě označení třídy jako anonymizační funkce nesou anotace `@AnonFunc` a `@AnonFuncParameter` navíc ještě informace o jejích vlastnostech.

Anotace `@AnonFunc` poskytuje informace o vlastnostech anonymizační funkce v podobě jednoho textové řetězce. Jednotlivé vlastnosti jsou v řetězci odděleny svislou čárou. Takto připravený řetězec se pak vkládal do výpisu ve formátech Confluence wiki nebo Markdown.



Obrázek 5.4: Výsledný návrh struktury exekučních módů



Obrázek 5.5: Výsledný návrh struktury poskytovatelů metadat

Druhá anotace `@AnonFuncParameter` představuje vstupní parametry anonymizační funkce a jejich popis. Anotace přijímá dva parametry – pole názvů parametrů a pole popisující jednotlivé parametry.

```
@AnonFunc("Anonymizuje město | Konzistentní |
    Neunikátní | Vrací
    null")
@AnonFuncParameter(
     names = ["puvodni"],
     descriptions = ["vstupní město"]
)
```

Listing 5.1: Aktuální anotace pro anonymizační funkce

Implementace anotací zobrazena na listingu 5.1 je zjevně náchylná na provedení lidské chyby. Anotace `@AnonFunc` poskytuje všechny informace v jednom řetězci, čímž odebírá možnost jiného formátování. Pokud chce vývojář pracovat s konkrétní vlastností dané anonymizační funkce, musí si řetězec sám rozparsovat na atomické informace. Anotace `@AnonFuncParameter` přidává programátorovi odpovědnost za správné pořadí parametrů a jejich popisu, protože jsou tyto informace rozděleny do dvou polí.

```
@AnonFunc(
     description = "Anonymizuje město",
     consistent =  true,
     unique =  false,
     nullable =  true,
     parameters = [
          @AnonFuncParameter(name = "puvodni",
             description = "vstupní město")
     ]
)
```

Listing 5.2: Nová struktura anotací pro anonymizační funkce

Stejné informace jako původní anotace nese i implementace na listingu 5.2.

Anotace je na načtené třídě stále dostupná a každá vlastnost je atomicky uložená ve své vlastní proměnné. Parametry anonymizační funkce jsou realizovány jako pole přímo v anotaci `@AnonFunc`. Každý parametr je pak reprezentován původní anotací `@AnonFuncParameter`, jež nyní přijímá dva řetězce – název parametru a jeho popis. Informace o parametru jsou tak sdruženy u sebe.

Pro možnost přesnější validace dalších funkcionalit byly implementovány anotace i pro další komponenty.

```
@TablePattern(description = "Create as  new")
```

Listing 5.3: Nová anotace pro tabulkové vzory

Pro dekorátory vznikla anotace `@Decoration`, neboť symbol `Decorator` je již ve stejném balíčku použit.

```
@Decoration(description = "Lower the anonymization
    function output.")
```

Listing 5.4: Nová anotace pro dekorátory

Všichni objevitelé byly dekorovány novou anotací na listingu 5.5.

```
@Discoverer(description = "Discover city name in table
    column.")
```

Listing 5.5: Nová anotace pro objevitele

Pro slovníky nebyla vytvořena žádná anotace, neboť se jejich výpis řídí odlišnou logikou, než u ostatních komponent. U slovníků probíhá validace pouze na základě umístění ve správném balíčku a přítomnosti abstraktního předka. Slovníky jsou následně rozděleny podle zemí, pro které jsou určeny. Takto rozřazeny jsou pak již klasicky vypsány na standardní výstup.

Pomocí reflexe a validací popsaných výše byl naplněn požadavek M3 – Dynamické načítání dostupných komponent.

5.3 Pluginovací systém

Ačkoliv Java platforma nabízí extension mechanism popsaný v rešeršní části této práce, ani jedna z variant není vhodná pro rozšiřování nástroje Winch. Instalované volitelné balíčky jsou dostupné pro všechny aplikace, které běží na daném JRE. Pluginy dodávané klientem nebo společností jsou ale specifické pro aplikaci Winch a koncového uživatele. Pluginy by tedy měly být dostupné pouze na class path nástroje při spuštění.

5.3.1 Gradle a Distribution Plugin

Aby byly třídy z rozšíření dostupné na class path aplikace, musí na ni být přidány pomocí přepínače `-cp` (nebo ekvivalentního `-classpath`) při startu komponenty Winch Actor.

Během sestavovacího Gradle kroku `installDist` (přidán pomocí pluginu Distribution Plugin) je mimo samotné distribuce vytvořena i nová složka `extensions`, jež je určena právě pro pluginy. Tato složka je pak společně

s cestou k distribuci a wildcard (*) přidána k ostatním záznamům přepínače `-classpath` ve spouštěcím skriptu nástroje. Výsledně jsou tedy k přepínači přidány záznamy níže.

- `%APP_HOME%\extensions\*` pro Windows.
- `$APP_HOME/extensions/*` pro Linux.

Přidáním celé složky pomocí wildcard na class path komponenty Winch Actor jsou splněny všechny dílčí části požadavku W1 – Rozšiřitelnost nástroje.

5.3.2 Pluginy

Pluginy jsou po strukturální stránce velmi podobné samotné komponentě Winch Actor. Aby správně fungovalo načítání tříd pomocí reflexe, musí být zachovány jmenné konvence u balíčků a některých názvů tříd. Každý plugin je závislý na modulu `disl-winch-connector`, který obsahuje abstraktní předky a rozhraní všech rozšiřovaných komponent. Pro přístup k třídám specifických pro konkrétní databázový stroj je nutno přidat závislost i na modul obsahující implementace pro danou databázi.

Protože instalace nástroje (kterou koncový uživatel disponuje) obsahuje jak společný, tak i konkrétní databázový modul v podobě JAR souborů, mohou být v sestavovacím skriptu pluginu přidány jako závislosti umístěné na cílové stanici. Přesune-li si klient složku s nástrojem Winch na jiné místo, než na které je během klasické instalace umístěna, je zároveň odpovědný i za patřičnou úpravu cesty k závislostem v sestavovacím skriptu nástroje Gradle.

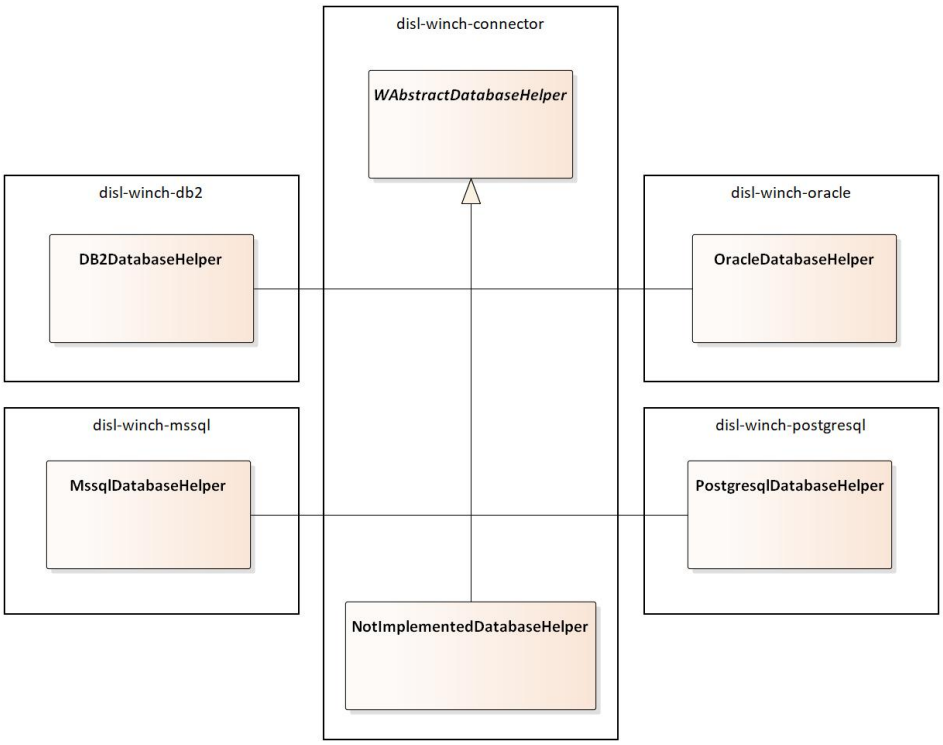
Pro usnadnění vývoje pluginů bude klientům zpřístupněn repozitář s připravenou kostrou a vzorovými komponentami, které si bude moci po zkompilování přidat do nástroje Winch. Celý postup tvorby a dodávky rozšíření je detailně sepsán v programátorské příručce.

5.3.3 Refactoring databázových pomocníků

Databázový pomocník (anglicky *database helper*) představuje v komponentě Winch Actor pomocnou třídu, která poskytuje specifické informace o databázovém stroji, včetně celých kusů SQL kódu pro danou databázi. Struktura databázových pomocníků je popsána na obrázku 5.6.

Abstraktní předek `WAbstractDatabaseHelper` udává svým potomků povinnost poskytovat konkrétní informace o databázi (například jaký oddělovač jednotlivých SQL příkazů se používá nebo kontrakt na implementaci metody pro získávání identifikátorů schémat, tabulek, procedur nebo funkcí z databáze.

Speciálním databázovým pomocníkem je `NotImplementedDatabaseHelper`, což je výchozí třída, jež je načtena v případě, že se na class path nenachází žádná implementace databázového pomocníka pro konkrétní databázi.



Obrázek 5.6: Struktura databázových pomocníků

Kromě informací popsaných výše také každý databázový pomocník poskytuje tzv. generátory kódu. Ty vrací kusy SQL kódů specifické pro konkrétní databázi. Z těchto fragmentů se nakonec sestaví celý proces anonymizace (vygenerování tabulek, logování a jiné).

Poskytování generátorů kódu je v databázových pomocnících aktuálně realizováno pomocí několika abstraktních metod v předkovi (metoda pro každý typ generátoru). Intuitivně by ale měl databázový pomocník nabízet komponentu, která se bude starat o distribuci správných generátorů. S touto myšlenkou byl využit návrhový vzor abstraktní továrna.

Z databázových pomocníků (abstraktního předka a všech implementací) proto byly odstraněny metody, které přímo poskytovaly konkrétní generátory, a nahradila je jediná metoda, jež poskytuje továrnu generátorů. Ta obecně vrací abstraktního předka všech generátorů SQL kódů a konkrétní metody pak vrací správné generátory.

```
databaseHelper.getAnonymizeTableCodeGenerator(table)
```

Listing 5.6: Získání generátoru kódu přímo z DB pomocníka

```
databaseHelper
    .getCodeGeneratorFactory()
    .getAnonymizeTableCodeGenerator(table)
```

Listing 5.7: Delegace tvorby generátoru kódu do továrny

5.3.4 Generalizace kroků v tabulkových vzorech

Stávající proces implementace tabulkových vzorů je triviální – ve zděděné inicializační metodě se vytvoří seznam specifických kroků, které definují proces anonymizace tabulky. Všechny tyto kroky přijímají ve svém konstruktoru jediný parametr, jímž je reference na instanci tabulkového vzoru, ve které vznikly.

Při přidávání nového tabulkového vzoru je programátor omezen na existující specifické kroky. V případě, že by během procesu anonymizace chtěl provést činnost, která ještě nebyla v jiném tabulkovém vzoru provedena, musí si pro novou funkci naprogramovat krok nový. Pro vývojáře, jenž je s nástrojem Winch a jeho implementací dobře obeznámen, se může jednat o jednoduchý úkon. Koncový uživatel nicméně nebude mít k dispozici kód aplikace a příprava nového kroku by pro něj byla velmi obtížná, čímž by se prodloužil celkový čas nutný k vytvoření a dodání pluginu.

Řešením je generalizace kroků na tři základní typy, které v nástroji aktuálně existují – `generate` kroky, `deploy` kroky a `execute` kroky. Zobecněné kroky pro tabulkové vzory přijímají při svém vzniku více parametrů. Tyto přijímané informace byly u specifických kroků definovány přímo v jejich vnitřku (například který generátor SQL kódu se má použít při `generate` krocích nebo jaký SQL kód se má spustit v rámci `execute` kroků).

Výsledkem je tak razantní snížení počtu druhů kroků, ačkoliv byly původní specifické kroky kvůli zpětné kompatibilitě dočasně v aplikaci ponechány. Generalizované kroky jsou navíc více flexibilní v tom, co mohou reálně dělat. Klient tak není vázán na existující generátory SQL kódů, ale může kroku předat konkrétní SQL skript, který se má v rámci anonymizace tabulky spustit.

Specifický krok pro generování tabulky je vyobrazen na listingu 5.8.

```
GenerateTableStep(pattern:  this)
```

Listing 5.8: Specifický krok pro generování tabulky

Ekvivalentní zápis za pomoci zobecněného kroku pro generování kódu popisuje listing 5.9.

```
GenericGenerateStep(
        pattern:  this,
        file: generateTableFile,
        codeGenerator: createTableCodeGenerator
)
```

Listing 5.9: Obecný krok pro generaci tabulky

Proměnná `generateTableFile` představuje soubor, do kterého se má zapsat vygenerovaný SQL kód a proměnná `createTableCodeGenerator` obsahuje generátor kódu, který dodá správný SQL skript pro tvorbu tabulky.

Kapitola 6

Ověření

Kapitola popisuje postup, kterým bylo ověřeno, že implementace poskytování metadat a pluginovacího systému splňuje definované požadavky.

6.1 Poskytování metadat

Poskytování metadat je v nástroji ověřováno pomocí jednotkových (*unit*) testů. Testováno je poskytování anonymizačních funkcí, tabulkových vzorů, dekorátorů, slovníků a objevitelů. Každý test má na vstupu statický řetězec, který představuje očekávaný výstup příslušného příkazu pro výpis.

Jednotkové testy jsou v tomto případě křehké. Pokud v budoucnu dojde k přidání či odebrání jedné z poskytovaných komponent, musí být adekvátně upraven i řetězec, proti kterému se výstup validuje. V opačném případě by patřičný test neproběhl úspěšně.

Testování výstupu metadat nekontroluje ve výpisech komponenty, které jsou dodávány v pluginech.

6.2 Pluginovací systém

Testování pluginovacího systému vyžaduje více práce než ověřování správného poskytování informací o dostupných komponentách. Prvně musí existovat správně implementované rozšíření, které je umístěno ve složce `extensions` v instalaci nástroje Winch. Dále je třeba mít nachystaná testovací data pro anonymizaci (v jedné z podporovaných relačních databázích) a projekt v EA s nahranou strukturou testovací databáze.

Pro účely práce jsou vytvořena dvě rozšíření. Tato skutečnost poukazuje na schopnost pluginovacího systému načítat do nástroje i několik rozšíření najednou.

První plugin obsahuje nový tabulkový vzor, jenž je implementován za pomoci nových generalizovaných kroků. Proces anonymizace je shodný s výchozím

tabulkovým vzorem (`WTablePattern`). Navíc jsou do něj ale přidány kroky, které volají vlastní proceduru připravenou v databázi.

Druhé rozšíření pak přidává novou anonymizační funkci a dekorátor, čímž je zároveň dokázáno, že jedno rozšíření může obsahovat několik různých komponent, a nástroj je schopný pracovat s každou z nich. Vytvořená anonymizační funkce umožňuje anonymizovat IP adresu (čtvrtou verzi). Dodaný dekorátor odstraňuje z výstupu jakékoliv anonymizační funkce diakritiku.

6.2.1 Programátorská příručka

Pro ověření realizovaného pluginovacího systému vznikla programátorská příručka, která představuje podrobný návod pro tvorbu vlastních pluginů pro koncové uživatele nástroje Winch. Programátorská příručka má textovou podobu a je součástí práce v elektronické podobě.

Postupy pro programování jednotlivých komponent a následného vytvoření finálního rozšíření vznikaly před samotným ověřováním rozšiřovacího mechanismu. Díky tomu mohla být programátorská příručka „otestována“, když se podle ní tvořila testovací rozšíření popsaná níže.

6.2.2 Plugin přidávající tabulkový vzor

Protože se při tvorbě nového tabulkového vzoru využilo nových generalizovaných kroků, je implementace třídy rozsáhlejší, než u ostatních tabulkových vzorů. Pro přípravu všech nově potřebných argumentů vznikly ve třídě nové vlastnosti a metody, které se starají o jejich inicializaci. Samotná třída představující tabulkový vzor se vytváří stejně, jako v nástroji samotném – stačí pro třídu vybrat vhodný název, dědit od abstraktního předka `WTablePatternAbstract` a nezapomenout přidat anotaci, jež popisuje činnost nového tabulkového vzoru. Plugin přidává třídu `ExemplarTablePattern`, která je vytvořena podle listingu 6.1.

```
@TablePattern(description = "Custom client table
    pattern.")
 class ExemplarTablePattern
    extends WTablePatternAbstract
```

Listing 6.1: Tvorba vlastního tabulkového vzoru

Pro správnou činnost tabulkového vzoru je třeba přetížit metodu `init`, ve které se definuje seznam kroků, jež mají být během zpracovávání tabulky spuštěny. Jak bylo zmíněno v úvodu sekce, nový tabulkový vzor kopíruje chování výchozího tabulkového vzoru `WTablePattern`. Ten ve své implementaci metody `init` využívá staré specifické kroky, jak je vyobrazeno na listingu 6.2.

```
@Override
 void init() {
    steps.clear()
    add  new GenerateTableStep(pattern:  this)
    add  new GenerateAnonymizeTableStep(pattern:  this)
    add  new GenerateForeignKeyTableStep(pattern:  this)
    add  new DeployTargetTableStep(pattern:  this)
    add  new DeployAnonymizeTableStep(pattern:  this)
    add  new DeployForeignKeysTableStep(pattern:  this)
    add  new ExecuteAnonymizeTableStep(pattern:  this)
    add  new ValidateTableStep(pattern:  this)
    add  new DiscoverTableStep(pattern:  this)
}
```

Listing 6.2: Metoda `init` výchozího tabulkového vzoru

Pro ukázku využití nových generalizovaných kroků je realizace metody `init` v dodávaném tabulkovém vzoru `ExemplarTablePattern` prováděna právě nimi.

```
@Override
 void init() {
    steps.clear() // do not remove this line

    initializeGenerateSteps()
    initializeDeploySteps()
    initializeExecuteSteps()

    // GENERATE STEPS SECTION
    add generateTableStep
    add generateAnonymizeTableStep
    add generateForeignKeyTableStep
    // END OF GENERATE STEPS SECTION

    // DEPLOY STEPS SECTION
    add deployTableStep
    add deployAnonymizeTableStep
    add deployForeignKeyTableStep
    // END OF DEPLOY STEPS SECTIONS

    // EXECUTE STEPS SECTION
    add  new GenericExecuteStep(code:
       getCustomProcedureCodeToExecute("Before
       executing anonymize step
```

```
        for ${table.getName()}."))

    add executeAnonymizeTableStep

    add  new GenericExecuteStep(code:
        getCustomProcedureCodeToExecute("After executing
        anonymize step
        for ${table.getName()}."))
    // END OF EXECUTE STEPS SECTION

    add  new ValidateTableStep(pattern:  this)
    add  new DiscoverTableStep(pattern:  this)
}
```

Listing 6.3: Metoda `init` dodávaného tabulkového vzoru

Jak si lze na listingu 6.3 všimnout, je implementace metody `init` delší. Jednotlivé generické kroky jsou přidávány v podobě proměnných, které jsou inicializovány v rámci volání metod `initializeGenerateSteps`, `initializeDeploySteps` a `initializeExecuteSteps`.

Za zmínku stojí přidání dvou dodatečných kroků okolo `executeAnonymizeTableStep`. Jedná se o generické kroky, které spouští proceduru v databázovém stroji. Tato jednoduchá procedura přijímá na svém vstupu zprávu, kterou vkládá do speciálně vytvořené logovací tabulky. Procedura v tomto scénářovém testu představuje zákazníkův vlastní SQL skript, který si přeje spouštět během zpracovávání tabulky. Metoda `getCustomProcedureCodeToExecute` je zachycena na listingu 6.4.

```
private String getCustomProcedureCodeToExecute(
   final String message) {
    "${dbHelper.getProcedureNameInSql("winch",
       CUSTOM_PROCEDURE_NAME)} N'${message}'"
}
```

Listing 6.4: Získání vlastního SQL skriptu ke spuštění

Pro demonstraci je na listingu 6.5 ukázána metoda, jež se stará o inicializaci *deploy* kroků.

```
private  void initializeDeploySteps() {
    AbstractCreateTableCodeGenerator
       createTableCodeGenerator =
       dbHelper.getCodeGeneratorFactory()
       .getCreateTableCodeGenerator(table)
```

```
    AbstractAnonymizeTableCodeGenerator
       anonymizeTableCodeGenerator =
       dbHelper.getCodeGeneratorFactory()
       .getAnonymizeTableCodeGenerator(table)
    AbstractForeignKeysTableCodeGenerator
       foreignKeysCodeGenerator =
       dbHelper.getCodeGeneratorFactory()
       .getForeignKeysCodeGenerator(table)

    deployTableStep =
       new GenericDeployStep(codeGenerator:
       createTableCodeGenerator, commandSeparator:
       dbHelper.getCommandSeparator())
    deployAnonymizeTableStep =
       new GenericDeployStep(codeGenerator:
       anonymizeTableCodeGenerator, commandSeparator:
       dbHelper.getCommandSeparator())
    deployForeignKeyTableStep =
       new GenericDeployStep(codeGenerator:
       foreignKeysCodeGenerator, commandSeparator:
       dbHelper.getCommandSeparator())
}
```

Listing 6.5: Inicializace deploy kroků

Protože dodávaný tabulkový vzor `ExemplarTablePattern` kopíruje chování výchozího tabulkového vzoru `WTablePattern`, měl by být výstup ve zpracovaných tabulkách stejný. Navíc by přidaná logovací tabulka měla obsahovat dva řádky za každou zpracovanou tabulkou (první záznam před spuštěním procesu anonymizace a druhý po něm).

Před samotným testováním byl plugin v podobě JAR souboru vložen do složky `extensions` v instalaci nástroje Winch. Pro ověření, že je tabulkový vzor aplikací načítán, byl spuštěn výpis všech dostupných tabulkových vzorů, který úspěšně obsahoval i dodaný tabulkový vzor `ExemplarTablePattern`.

Výstup tabulky byl po zpracování tabulkovým vzorem `WTablePattern` exportován do souboru, který sloužil jako vzorová validační data. Následně byl u stejné tabulky ve Winch Add-inu v nástroji EA změněn tabulkový vzor na `ExemplarTablePattern`. Po spuštění procesu byla tabulka opět exportována. Následně byla provedena pozitivní kontrola výsledků, neboť byl obsah obou souborů (a tedy tabulek) shodný. Logovací tabulka byla navíc řádně naplněna dvěma řádky – prvním před spuštěním anonymizace tabulky a druhým po ní.

6.2.3 Plugin přidávající anonymizační funkci a dekorátor

Kromě přidávání tabulkových vzorů ukládají požadavky na pluginovací systém povinnost rozšiřovat nástroj Winch ještě o anonymizační funkce a dekorátory. Tyto komponenty nejsou součástí prvního rozšíření, jenž přidává nový tabulkový vzor, ale jsou implementovány v pluginu novém.

Ve stávajícím stavu neexistuje v nástroji Winch anonymizační funkce pro IP adresu. Scénář tedy představuje situaci, kdy zákazníkovo databázové schéma obsahuje sloupeček s IP adresami verze čtyři, které by se neměly dostat do nového anonymizovaného schéma. Za tímto účelem je proto v novém rozšíření implementována anonymizační funkce `MssqlFunctionIpv4Address` pro databázový stroj MSSQL, která generuje náhodnou IP adresu. Protože se jedná pouze o ukázku možností rozšiřovacího mechanismu, je realizace nové anonymizační funkce jednoduchá – na svém výstupu má vždy čtyři náhodná čísla v rozsahu 0–255 oddělená tečkami. Výstupní řetězec se generuje v cyklu tak dlouho, dokud je generovaný řetězec shodný se vstupem. U výsledné IP adresy nedochází k validaci z pohledu IP standardu (rezervovanost některých IP adres a jiné).

Tvorba anonymizační funkce v rámci pluginu je opět velmi podobná tvorbě anonymizační funkce v nástroji samotném (viz listing 6.6).

```
@RequiredResources([
  "mssql/anon/_common/number/fc_decimal_number_rnd.sql",
  "mssql/anon/ip/func/fc_ipv4_address.sql"
])
@AnonFunc(
    description = "Anonymizuje IPv4 adresu",
    consistent =  false,
    unique =  false,
    nullable =  true
)
 class MssqlFunctionIpv4Address
   extends AnonymizationFunction
```

Listing 6.6: Tvorba vlastní anonymizační funkce

Skript `mssql/anon/_common/number/fc_decimal_number_rnd.sql` je přímo součástí nástroje Winch, neboť se používá u několika existujících anonymizačních funkcí. Nebude-li ale seznam společných skriptů poskytnut klientům, musí si koncový uživatel veškeré SQL funkcionality, které pro anonymizační funkci vyžaduje, implementovat sám a následně vložit jejich výčet do anotace `@RequiredResources`.

Důležitou částí anonymizační funkce je statická metoda `getFunctionSql`. Z důvodu zpětné kompatibility a nutnosti velmi rozsáhlého refaktorování,

neexistuje žádný kontrakt, jenž by třídě představující anonymizační funkci udával povinnost tuto metodu implementovat. Její existence ve třídě je tak založená na společné úmluvě. Metoda vrací název funkce (procedury) v databázovém stroji včetně všech parametrů pro její úspěšné zavolání v podobě řetězce. Implementace metody je vyobrazena na listingu 6.7.

```
static String getFunctionSql(String firstInput,
   String... otherInputs) {
    MssqlFunctionIpv4Address anonymizationFunction =
       new MssqlFunctionIpv4Address()
    return anonymizationFunction
        .getDbFunctionNameWithParameters(firstInput,
           otherInputs)
}
```

Listing 6.7: Metoda `getFunctionSql`

Samotný název databázové funkce či procedury se anonymizační funkci předává v konstruktoru.

```
MssqlFunctionIpv4Address() {
    super(DB_FUNCTION_NAME)
}
```

Listing 6.8: Konstruktor anonymizační funkce

`DB_FUNCTION_NAME` je konstanta, jež nese název databázové funkce – v tomto případě je její hodnotou řetězec „fc_ipv4_address“. Funkce musí existovat v jednom z požadovaných SQL souborů.

První ověření spočívalo v úspěšném načítání nové anonymizační funkce poté, co byl plugin vložen do složky `extensions`. Podobně jako u testování tabulkového vzoru se opět spustil nástroj Winch s příkazem pro výpis dostupných anonymizačních funkcí, ve kterém se dodaná anonymizační funkce nacházela.

Dále bylo nutné otestovat samotnou funkčnost nové anonymizační funkce. Do testovacího schéma byl přidán sloupec pro IP adresy. Sloupeček byl následně ve všech řádcích vyplněn privátními adresami. Ve Winch Add-inu se vytvořila nová anonymizační třída pro IP adresu a byla jí předána dodaná anonymizační funkce (v podobě volání statické metody `getFunctionSql` třídy `MssqlFunctionIpv4Address`). Po spuštění procesu anonymizace tabulky obsahoval sloupeček s IP adresami náhodné IP adresy, kde se navíc žádná neshodovala s původní, čímž byla ověřena funkčnost nové anonymizační funkce.

Kromě anonymizační funkce byl v rámci nového rozšíření implementován i dekorátor. Ten na patřičném sloupci přijímá na vstupu výstup anony-

mizační funkce a zbaví jej diakritiky. Dekorátor simuluje případ, kdy koncový uživatel vyžaduje ve výsledném sloupci data bez diakritiky, například z důvodu kódování nebo kompatibility s konzumenty anonymizovaných dat. Dekorátor byl stejně jako anonymizační funkce implementován pro databázový stroj MSSQL (listing 6.9).

```
@Decoration(description = "Remove diacritics from
   anonymization function output.")
 class MssqlRemoveAccentsDecorator  implements Decorator
```

Listing 6.9: Tvorba vlastního dekorátoru

Rozhraní `Decorator` představuje kontrakt, jenž třídě ukládá povinnost implementovat metodu `decorate`. K dispozici má výstup anonymizační funkce v argumentu `inputSql` a název sloupečku tabulky `column`. Na výstup anonymizační funkce může být aplikován SQL kód, který je právě touto metodou vracen. Její implementace se nachází na listingu 6.10.

```
@Override
String decorate( final String column,
   final String inputSql) {
    "${inputSql} COLLATE
       SQL_Latin1_General_Cp1251_CS_AS"
}
```

Listing 6.10: Implementovaná metoda `decorate`

Nový dekorátor byl součástí nového pluginu, ten tedy stačilo znovu zkompilovat do JAR souboru a přepsat jím původní rozšíření ve složce `extensions`.

Následoval stejný postup testování v podobě ověření dostupnosti nového dekorátoru ve výpisu. Test dopadl opět pozitivně.

Způsob ověření samotné funkčnosti dekorátoru byl tentokrát komplikovanější, neboť Winch Add-in v EA prozatím nepodporuje práci s dekorátory v rámci svého grafického rozhraní. Samotná funkcionalita je ale dostupná pomocí tagu v EA. K anonymizační třídě pro příjmení se skrze tag `winch.outputdecorators` přidal nový dekorátor v podobě jeho jména bez databázového prefixu (tedy pouze řetězec `RemoveAccentsDecorator`). Po spuštění anonymizace a následné kontroly výsledné tabulky neobsahovalo žádné příjmení diakritiku, což nasvědčovalo úspěšnému načtení SQL kódu z dekorátoru a jeho aplikaci na výstup anonymizační funkce pro příjmení.

6.3 Shrnutí ověřování

V rámci ověřování byly postupně otestovány všechny realizované součásti této práce. I přes nesčetné množství chyb, které se během testování objevovaly, se vždy podařilo prokázat funkčnost všech dílčích částí implementovaného řešení.

Velkým přínosem bylo ověřování pluginovacího systému. Při tvorbě vlastních pluginů podle připravené programátorské příručky se postupně vynořovaly její nedostatky. Ačkoliv byly mnohé z nich napraveny (jmenovitě například bližší představení anonymizačních tříd a všech užitečných metod jejich abstraktního předka nebo zavedení přesných jmenných konvencí pro určité komponenty), trpí programátorská příručka nedostatkem interakce s koncovým uživatelem. Pro plné pochopení nástroje Winch a jeho nových možností na rozšiřování by tak mohlo být vhodné připravit „vylepšenou“ programátorskou příručku – online kurz s možností přímé komunikace s vývojářem nástroje nebo video tutoriály.

Závěr

V bakalářské práci byl představen anonymizační nástroj Winch společně s jeho aktuálním stavem. Z analýzy aplikace, zadání práce a konzultacemi se zadavatelem (vedoucím práce) pak byla provedena definice konkrétních požadavků. K jejich naplnění byl potřeba provést průzkum v podobě rešerší, které věnovaly svoji pozornost především možnostem rozšiřování aplikací postavených na platformě Java. Analytická část práce se také věnovala přiblížení návrhových vzorů. Teoretický základ se pak stal vstupem pro tvorbu návrhů na úpravu kódu nástroje a samotného pluginovacího systému. Některé návrhy pak dostály využití v implementační fázi. Pluginovací systém a další dílčí části realizace byly nakonec otestovány pomocí klasických jednotkových testů a vlastně naprogramovaných pluginů, jež byly vytvořeny pomocí postupů popsaných v programátorské příručce, která je taktéž výsledkem práce.

I přes mnohé překážky, způsobené především špatnými architektonickými rozhodnutími během dřívějšího vývoje aplikace a formováním finálních požadavků na rozšiřování až po prvotních návrzích, se podařilo kýžených cílů dosáhnout. Nástroj Winch lze nyní rozšiřovat o požadované komponenty, které mohou být do aplikace dodávány v podobě pluginů. Tato možnost je určena jak koncovým klientům tak vývojářům samotným. Všechny cíle práce a z nich následně vydefinované požadavky tak byly splněny v plném rozsahu.

Při finalizaci práce se objevily možnosti, jak pluginovací systém vylepšit. Repozitář s připravenou kostrou pro tvorbu rozšíření může být využit k verzování pluginů, které vyvíjí společnost sama, na základě požadavků od jejích klientů. Připravená rozšíření by navíc mohla být na popud velkého zájmu o dodané komponenty zakomponovány do nástroje samotného. Aktuálně nastavený proces ale nepředpokládá aktualizaci repozitáře (až na úpravy či přidání vzorových komponent), ani přenášení změn mezi repozitářem pro tvorbu pluginů a repozitářem nástroje Winch. Analýza konkrétních požadavků firmy by mohla vést k nastavení lepšího postupu při tvorbě rozšíření a jejich případnému slučování s nástrojem.

Programátorská příručka v textové podobě je dostačující pro člověka, jenž

má zkušenost s programováním aplikací v Groovy a sestavovacím nástrojem Gradle. Taktéž se pro její využití předpokládá u uživatele základní znalost nástroje Winch. Pro nezasvěcené uživatele nástroje by ale mohlo být vhodné v budoucnu vytvořit interaktivnější návod pro tvorbu rozšíření, případně změnit podobu programátorské příručky z textové na audiovizuální.

Bibliografie

1. SCHUH, Matěj. *Implementace datové vrstvy pro anonymizační nástroj.* Praha, 2018. Bakalářská práce. České vysoké učení technické v Praze. Fakulta informačních technologií.
2. *Groovy Language Documentation* [online]. 2015 [cit. 2018-04-15]. Dostupné z: `http://docs.groovy-lang.org/docs/groovy-2.3.10/html/documentation/`.
3. BERLUNG, Tim; MCCULLOUGH, Matthew. *Building and Testing with Gradle: Understanding Next-Generation Builds.* USA: O'Reilly Media, 2011. ISBN 978-1-449-30463-8.
4. SHAFRANOVICH, Y. *RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files* [online]. 2005 [cit. 2019-04-06]. Dostupné z: `https://tools.ietf.org/html/rfc4180`.
5. *Confluence Wiki Markup - Atlassian Documentation* [online]. 2019 [cit. 2019-04-07]. Dostupné z: `https://confluence.atlassian.com/conf614/confluence-wiki-markup-965545663.html`.
6. GRUBER, John. *Daring Fireball: Markdown* [online]. 2004 [cit. 2019-04-07]. Dostupné z: `https://daringfireball.net/projects/markdown/`.
7. BRAY, T. (ed.). *RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format* [online]. 2017 [cit. 2019-04-07]. ISSN 2070-1721. Dostupné z: `https://tools.ietf.org/html/rfc8259`.
8. *System Requirements - Enterprise Architect — Sparx Systems* [online]. 2019 [cit. 2019-04-13]. Dostupné z: `https://sparxsystems.com/products/ea/sysreq.html`.
9. LINDHOLM, Tim et al. *The Java© Virtual Machine Specification: Java SE 8 Edition* [online]. 2015 [cit. 2019-04-11]. Dostupné z: `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`.

10. *Java© Platform Overview* [online]. 2018 [cit. 2019-04-13]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/index.html.

11. *Setting the Class Path* [online] [cit. 2019-04-09]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html.

12. *How Classes are Found* [online] [cit. 2019-04-11]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/tools/findingclasses.html.

13. *The Extension Mechanism* [online]. 2018 [cit. 2019-04-13]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/extensions/spec.html.

14. GAMMA, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995. ISBN 0-201-63361-2.

Příloha **A**

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu média
- `readme_opponent.txt` ..... informace pro oponenta práce
- `appendix`
  - `programming_guide.docx` ..... prog. příručka ve formátu DOCX
  - `programming_guide.pdf` ..... prog. příručka ve formátu PDF
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis_perner_2019.pdf` ..... text práce ve formátu PDF