FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Anonymizace osobních údajů pro databáze MySQL a Teradata
Student:	Ondřej Brychta
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

Seznamte se s existující implementací nástroje Winch, který slouží pro anonymizaci osobních údajů v databázích Oracle, MSSQL, DB2 a PostgreSQL. Proveďte analýzu existujícího řešení z pohledu odlišností implementace v jednotlivých databázových systémech.
Na základě provedené analýzy navrhněte a popište postup, který umožní efektivně vytvářet implementace pro další databázové systémy. Navržený postup musí minimalizovat duplicity ve zdrojových kódech, které se v současné implementaci objevují.Na základě navrženého postupu proveďte implementaci anonymizace pro databázové systémy MySQL a Teradata, která ověří jeho praktickou použitelnost. Implementované řešení otestujte.
Zhodnoťte vámi navržený postup a popište další možná vylepšení pro budoucí rozvoj.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 18. prosince 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Anonymizace osobních údajů pro databáze MySQL a Teradata

Ondřej Brychta

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Mlejnek

27. června 2019

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Jiřímu Mlejnkovi za jeho rady, konzultace a připomínky, kterými přispěl ke zdárnému dokončení této práce.

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 27. června 2019 .....

Odkaz na tuto práci

Brychta, Ondřej. *Anonymizace osobních údajů pro databáze MySQL a Teradata*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce pojednává o rozšíření již existujícího nástroje GEM Winch na provádění datových řezů, vyhledávání a anonymizování osobních údajů v databázích. Ke čtyřem momentálně podporovaným databázovým systémům jsou přidány další dva - MySQL a Teradata. Aktuální podoba implementace způsobuje, že v implementacích pro jednotlivé databázové systémy, ale i v rámci modulů, vzniká mnoho duplicit v kódu či logice, které z vytváření nových implementací nástroje Winch dělá zdlouhavý a náročný proces, při kterém je třeba opakovaně programovat stejné, ve starších modulech již existující, anonymizační funkce. Na základě analýzy stávající implementace nástoje Winch předkládám návrh postupu, jak tento proces zjednodušit a množství duplicit snížit. Protože velkou část opakované logiky představují zdrojové SQL kódy jednotlivých anonymizačních funkcí pro různé databáze, zvolil jsem jako řešení pokusit se tyto zdrojové SQL kódy generovat dynamicky, za pomoci generátorů a šablon. Efektivita a použití navrženého postupu je ukázáno právě na nově přidaných implementacích pro databáze MySQL a Teradata. Detailnější popis navrhovaného postupu je obsahem dalšího textu, stejně tak počáteční analýza, implementace, testování a zhodnocení.

Klíčová slova DBMS, databáze, sql, generátor kódu, anonymizace osobních údajů, Winch, anonymizační funkce, generátor SQL, osobní údaje, šablony, šablony funkcí

Abstract

The work deals with extending an existing tool GEM Winch, which allows to perform data cuts and to find and anonymize personal data in databases. Winch is extended by adding support for MySQL and Teradata database systems on top of the four systems currently supported by the tool. Because of Winch's design, a lot of contained code or logic is duplicated within individual modules as well as across implementations for different database systems and so adding support for new database systems is a challenging and very time-consuming process requiring a lot of anonymization functions, already implemented in older modules, to be repeatedly written again with minimal changes. Therefore, based on the analysis of current implementations of Winch and their differences, I am proposing a new approach, which would lead to simplification of this process and reduce the amount of duplicated logic. Because a big part of the duplicated code is contained within SQL source codes of individual anonymization functions for different databases, the solution I have chosen is based on using generators and templates as an attempt to generate these SQL source codes dynamically. The effectiveness and usage of this approach is shown on newely added Winch implementations for MySQL and Teradata. More detailed look on the proposed process of generating SQL source codes is covered by the following text, as well as its implementation, testing and evaluation.

Keywords DBMS, database, sql, code generator, anonymization of personal data, Winch, anonymization function, SQL generator, personal data, templates, function templates

Obsah

Seznam obrázků

Úvod

Ochrana osobních údajů je v dnešní digitální době velice důležitým tématem. V roce 2018 vešlo v platnost nařízení EU o ochraně osobních údajů GDPR[1], které s sebou přineslo velmi přísná pravidla pro nakládání s osobními údaji, jejich skladováním a zabezpečením a rovněž vysoké sankce za jejich porušení. Jedním ze způsobů, jak se pohybovat v mezích těchto pravidel, je anonymizování již nepotřebných osobních dat v databázích.

Právě anonymizace osobních údajů v databázích je jedna z funkcí již existujícího a stále se rozvíjejícího nástroje GEM Winch (dále jen Winch). V aktuální verzi funguje s databázemi Microsoft SQL Server, Oracle, DB2 a PostgreSQL, avšak v současnosti je přidávání podpory dalšího databázového systému velice zdlouhavý proces, při kterém se velké množství kódu opakuje napříč implementacemi pro jednotlivé databázové systémy.

V této práci se zabývám návrhem a implementací nového postupu, který umožní efektivně přidávat podporu dalších databázových systémů, s ohledem na snížení množství duplikovaného kódu oproti současnému stavu. Cílem navrhovaného postupu bude zjednodušení a zrychlení procesu rozšiřování nástroje Winch o podporu dalších databázových systémů. V dalším textu popisuji jak návrh řešení a jeho implementaci, tak i jeho použití při implementaci Winche pro databázové systémy MySQL a Teradata, což zároveň umožňuje zhodnotit jeho efektivitu, zda je nový postup efektivní a míru zjednodušení implementace nových DBMS[2].

V návrhu řešení se zaměřím hlavně na možnosti, jak generovat zdrojový kód SQL funkcí, které mají na starost anonymizaci jednotlivých osobních údajů přímo v databázích, jelikož je každá z nich implementována několikrát, pro každý DBMS zvlášť v jeho SQL dialektu. Kódy těchto funkcí představují největší podíl duplikovaného kódu, jehož generování by zjednodušilo rozšiřování nástroje Winch o podporu dalších databázových systémů.

[1] General Data Protection Regulation

[2] Database Management System (Systém řázení báze dat)

Na začátku textu se věnuji stávající implementaci nástroje Winch a jsou identifikována místa, která jsou nejvíce problematická s ohledem na duplikovaný kód a kde je tedy nejvhodnější prostor pro zlepšení. Následuje popis navrhovaného řešení a jeho architektury. V dalších kapitolách je podrobněji rozebrána implementace navrhovaného řešení a jeho použití při rozšíření nástroje o podporu databází MySQL a Teradata. Posledním bodem je zhodnocení efektivity navrženého a implementovaného řešení na základě jeho využití v praxi a návrhy na další možné změny a vylepšení.

Kapitola 1

Cíl práce

Cílem této práce je navrhnout postup, který povede k efektivnějšímu a rychlejšímu přidávání podpory dalších databází do nástroje Winch, sloužícího k provádění datových řezů a anonymizování osobních dat, a zlepší přehlednost a udržovatelnost jeho zdrojových kódů. Nutným předpokladem pro dosažení tohoto cíle je provedení analýzy stávající implementace nástroje Winch. Analýza bude zaměřena na rozdíly v implementacích nástroje pro různé databázové systémy s ohledem na množství duplicit ve zdrojových kódech a náročnost implementace nástroje pro nový databázový systém. Návrh nového postupu a změn musí vést ke snížení velkého množství duplicit ve zdrojových kódech a tím usnadnit budoucí rozvoj nástroje. Navržený postup poté bude otestován v praxi tak, že s jeho pomocí budou vytvořeny implementace Winche pro databáze MySQL a Teradata. To poté umožní zhodnotit praktickou použitelnost nového řešení a míru zlepšení oproti původnímu postupu. Výsledkem práce je nástroj Winch, rozšířený o podporu dvou nových databázových systémů a jeho implementace upravná tak, že přidávání dalších databází je snažší (rychlejší). Výsledný kód již neobsahuje takové množství duplicit, což má za následek zlepšení přehlednosti a udržovatelnosti zdrojových kódů nástroje Winch.

Kapitola 2

Analýza

2.1 Představení nástroje Winch

Winch je nástroj, který umožňuje provádět datové řezy a transformace dat v databázích, v nichž také automaticky upozorňuje na výskyt osobních dat, která poté umožní anonymizovat. V minulosti bylo na téma anonymizace osobních údajů, i Winch samotný, vypracováno již několik bakalářských prací, například [1, 2, 3], které se zabývaly dílčími částmi nástroje Winch nebo teorií a myšlenkami, na kterých je vybudován. Souběžně s touto prací vznikají i další práce, které Winchi přinesou novou funkcionalitu a možnosti. Aktuální verze nástroje umí pracovat s databázovými systémy Oracle, Microsoft SQL Server, DB2 a PostgreSQL. Skládá se ze dvou částí, tou první je Winch Addin, který má formu pluginu do aplikace Enterprise Architect a mimo jiné slouží k pohodlnému plánování a nastavování parametrů anonymizace. Také poskytuje uživatelské rozhraní pro ovládání druhé části nástroje Winch - konzolové aplikace Winch Connector[3], který na základě řídícího souboru, vygenerovaného Addinem, připravuje SQL skripty a spouští proces anonymizace přímo v databázi. V rámci této práce se zabývám pouze anonymizací osobních dat, konzolové aplikace Winch Connector.V dalším textu se omezím hlavně na části konzolové aplikace Winch Connector, které slouží k anonymizaci. Winch Addin, ani zbylé části Connectoru, nejsou předmětem této práce a nebudu se jimi tedy podrobněji zabývat. Zde je seznam některých funkcí nástroje Winch:

- Vyhledávání osobních údajů v databázi
- Nastavení anonymizace
- Plánování procesu anonymizace a jeho spuštění v DB
- Datové řezy - výběr pouze omezené podmnožiny dat
- Transformace dat

[3] Dále v textu jen Connector

2.1.1 Vyhledávání osobních údajů v databázi

Vyhledávání osobních údajů je proces prozkoumávání databáze, při kterém se podle typických možností uložení a pomocí různých validačních funkcí identifikují sloupce v databázových tabulkách, které s velkou pravděpodobností obsahují nějaký osobní údaj. Na ty poté Winch uživatele upozorní, což může pomoci při následném plánování a konfiguraci anonymizace dat v těchto sloupcích. Vyhledáváním osobních údajů v databázích se dříve zabývala bakalářská práce Davida Skalského [3], kde se o použitých postupech a myšlenkách můžeme dočíst více.

2.1.2 Nastavení anonymizace

Samotný proces anonymizace těchto nalezených, nebo i ručně zvolených dat umožňuje Winch kofigurovat dle potřeb a požadavků uživatele. Ten má možnost anonymizaci omezit pouze na některé tabulky či sloupce a ke každému sloupci přiřadit některou z nabízených anonymizační funkcí. Některým z těchto funkcí lze přidat další parametry, například konstantu nebo hodnotu z jiného sloupce, kterými můžeme výstup ovlivnit. Toho lze využít například pro zlepšení funkce na anonymizaci křestního jména, které informace o pohlaví dané osoby zjednoduší výběr anonymizovaného jména.

2.1.3 Plánování procesu anonymizace a jeho spuštění v DB

Po uživatelském nastavení následuje plánování procesu anonymizace, které provede Winch Connector na základě předchozí konfigurace, načež vygeneruje řídící soubor s naplánovaným postupem anonymizace ve formě SQL skriptu pro danou databázi. Connector tento skript umí do databáze také automaticky nasadit a v databázi spustit. Anonymizaci samotnou tedy neprovádí Winch, ale celá probíhá uvnitř databáze s pomocí Winchem připraveného postupu a vygenerovaných procedur. Ve vygenerovaných procedurách pro anonymizaci Winch využívá databázových funkcí, které mají za úkol anonymizaci konkrétního osobního údaje přímo v databázi, validačních funkcí sloužících ke kontrole správnosti původních i nových anonymizovaných dat a různých dalších pomocných funkcí a tabulek, které je nutné v databázi předem připravit. K tomu slouží inicializační skript, který je součástí distribuce nástroje Winch a který je třeba v databázi spustit, než se začne Winch používat.

2.1.4 Datové řezy

Objem anonymizovaných dat může být uživatelem omezen pomocí datového řezu, při kterém Winch zachovává vazby mezi entitami a nenarušuje tak konzistenci výsledné podmnožiny získaných dat. Datové řezy umožňuje Winch provádět i bez anonymizace. V kombinaci s anonymizováním osobních dat jich lze využít například pro přípravu databáze do testovacího prostředí, kde

nepotřebujeme takový objem dat, jako se nachází v produkčním prostředí, a kde není žádoucí mít skutečná osobní data. Více o datových řezech a anonymizaci si můžeme přečíst zde [1].

2.2 Architektura a implementace

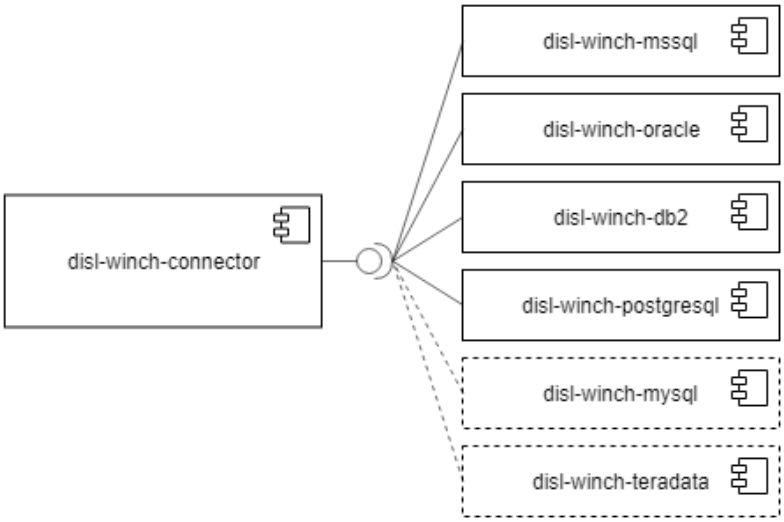
V této části se omezím na architekturu konzolové aplikace Winch Connector, jelikož ta obsahuje veškerou logiku potřebnou k anonymizaci a spolu s konkrétním databázovým systémem vykonává potřebné operace. Plugin pro Enterprise Architect se cíle této práce netýká a pro rozšíření Connectoru není třeba do jeho implementace zasahovat, takže implementací a architekturou Winch Addinu se není třeba detailně zabývat.

Pro implementaci aplikace Winch Connector byla použita kombinace programovacích jazyků Java a groovy. Aplikace má zvláštní implementaci pro každý databázový systém a je také jednotlivě distribuována. Součástí distribuce je i inicializační SQL skript pro danou databázi, pomocí kterého se do databáze nahrají pomocné a anonymizační funkce, tabulky a slovníky, které Winch používá pro anonymizaci přímo v databázi. Implementace pro každý databázový systém je umístěna ve zvláštním modulu s odpovídajícím jménem *disl-winch-dbname*. Jednotlivé specializované moduly jsou odvozeny od jednoho základního modulu *disl-winch-connector*, který osahuje balíčky s třídami implementujícími sdílenou funkcionalitu a dodává abstraktní třídy a rozhraní, čímž definuje výslednou architekturu aplikace. Jednotlivé moduly pak obsahují potomky abstraktních tříd, implementace daných rozhraní z modulu *disl-winch-connector* a další třídy, které přidávají novou funkcionalitu nebo přetěžují metody, kde základní implementace nevyhovuje dané databázi - například generují kód v dialektu SQL specifickém pro daný databázový systém.

V aktuální implementaci jsou podporovány databáze Oracle, MS SQL Server, DB2 a PostgreSQL a mimo základního modulu *disl-winch-connector* jsou tedy implementovány další čtyři moduly odpovídající jednotlivým databázovým systémům, konkrétně *disl-winch-oracle*, *disl-winch-mssql*, *disl-winch-db2* a *disl-winch-postgresql*. Jelikož tato práce zahrnuje implementaci nástroje Winch pro databáze MySQL a Teradata, přibily v průběhu jejího vypracování odpovídající moduly *disl-winch-mysql* a *disl-winch-teradata*. Rozdělení modulů (včetně nově přidaných) je vidět na obrázku 2.1

2.3 Obsah jednotlivých modulů

Jak již bylo řečeno v sekci výše, základní modul *disl-winch-connector* udává rozhraní, abstraktní třídy a implementaci sdílé funkcionality a pomocných tříd, tedy představuje základ, na kterém jsou postaveny ostatní moduly, které už jsou specifické pro konkretní databázový systém.



Obrázek 2.1: Diagram modulů s implementacemi WinchActor

Tyto specifické moduly již implementují jen relativně malou část funkcionality. Hlavní rozhraní, jejichž implementace je třeba do jednotlivých modulů přidat a další třídy a soubory, kterými je potřeba doplnit nebo změnit funkcionalitu, jsou tyto:

1. WAbstractDatabaseHelper (balíček com.gem.winch.database)
2. Dekorátory kódu (balíček com.gem.winch.decorator)
3. SqlDictionaryBuilder (balíček com.gem.winch.dictionary)
4. Jednotlivé abstraktní generátory zdrojového SQL kódu procedur pro provedení anonymizace (balíček com.gem.winch.pattern.codegenerator)
5. Jednotlivé třídy představující konkrétní anonymizační funkce (balíček com.gem.winch.database.anonymization)
6. Zdrojové soubory s SQL kódy anonymizačních a pomocných funkcí

2.3.1 WAbstractDatabaseHelper

WAbstractDatabaseHelper je abstraktní třída definovaná v základním modulu, sloužící jako „prostředník“ pro získávání a předávání instancí tříd specifických pro daný modul, hlavně instance tříd generujících zdrojový kód databázových procedur, které v databázi provedou anonymizaci. Každá implementace Winche musí obsahovat třídu, která z *WAbstractDatabaseHelper* dědí

a implementuje chybějící metody. Jednotliví potomci této třídy se napříč databázemi příliš neliší, fungují ale částečně jako „továrna“ poskytující konkrétní instance potřebných tříd.

2.3.2 Generátory SQL procedur

Sada tříd sloužících ke generování SQL kódu pro vytvoření a mazání tabulek, procedur provádějících anonymizaci jednotlivých tabulek databáze a kódu jednotlivých CRUD operací. Používají se při generování skriptů, které se po nasazení a spuštění starají o samotný průběh anonymizace dat přímo uvnitř databáze.

2.3.3 Dekorátory kódu

Třídy LowerDecorator, NotNullDecorator a další „dekorátory“, tedy třídy sloužící k získání SQL příkazu pro úpravu výstupu anonymizačních funkcí, zapsaném ve specifickém dialektu jazyka SQL příslušného databázového systému. Dekorátory je třeba do modulu zařadit pouze pokud nestačí jejich základní implementace a je třeba ji přetížit. Tyto třídy obsahují pouze metodu *decorate*, která vrátí vstupní řetězec (SQL příkaz) „obalený“ příslušnou funkcí, například *LOWER(input)*.

2.3.4 SqlDictionaryBuilder

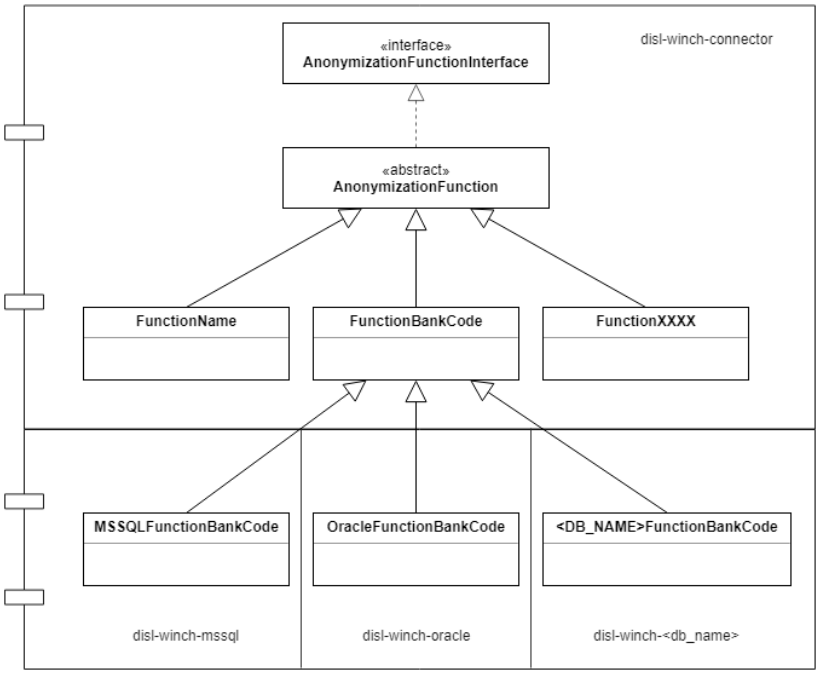
SqlDictionaryBuilder je třída, jejímž úkolem je vygenerovat SQL kód pro vytvoření tabulek, představujících slovníky, a jejich naplnění daty, které Winch v některých anonymizačních funkcích využívá. Data, kterými slovníky naplní, jsou poskytnuta v podobě zdrojových souborů[4] v modulu *disl-winch-connector*. Kód vygenerovaný touto třídou, nebo jejími potomky, je součástí inicializačního skriptu pro prvotní přípravu schématu a funkcí, které Winch ke svému běhu potřebuje.

2.3.5 Třídy představující anonymizační funkce

Anonymizačních funkcí je momentálně 48[5] a implementace přslušných tříd, kterými jsou reprezentovány, je velmi podobná. Hierarchii funkčních tříd naznačuje obrázek 2.2. Všechny třídy z prostřední úrovně(Function*) rozšiřují abstraktní třídu AnonymizationFunction, ale liší se pouze vnitřními konstantami a slouží jako základ pro další třídy, reprezentující již danou funkci v konkrétní databázi (např. OracleFunction*). Tyto třídy z poslední úrovně většinou opět nepřidávají mnoho nové logiky, jejich hlavním účelem je držet informace o dalších anonymizačních funkcích a pomocných funkcích, které

[4] resources

[5] platí pro mou vývojovou větev, v jiných větvích můžou být přidávány další



Obrázek 2.2: Diagram hierarchie anonymizačních tříd

potřebují mít pro správné fungování v databázi předinstalované, tedy seznam závislostí. Tyto informace jsou každé anonymizační třídě „předány“ pomocí speciální anotace *@RequiredResources*, která obsahuje seznam cest k souborům obsahujícím SQL kód jednotlivých potřebných funkcí. V aktuální implementaci tak funkční třída v anotaci dostává i cestu ke své vlastní SQL implementaci.

V této hierarchii vzniká velké nožství tříd s malou přidanou hodnotou, které navzájem sdílí velké množství duplikovaného kódu, což není dobré pro snadnou orientaci ve zdrojovém kódu nástroje Winch a jeho udržovatelnost. Mezi těmito třídami je tedy mnoho duplicitního kódu s minimem přidané funkcionality. Zdrojový kód by se dal pročistit a implementace těchto částí urychlit například přidáním třídy, která bude sloužit jako továrna[6] a instance anonymizačních funkcí vytvářet za běhu z jedné základní třídy. Rozdílné hodnoty a menší části kódu lze pak instancím doplňovat pomocí getterů, setterů a groovy closures[7]. Duplicity mezi těmito funkcemi jsou však problém hlavně kvůli přehlednosti a udržovatelnosti kódu, v balíčcích „překáží“ a zvyšují počet souborů, které se musí vytvořit při implementaci Winche pro nový databázový

[6] návrhový vzor Factory

[7] Konstrukt podobný lambda funkcím v jazyce Java

systém. Z hlediska času pro jejich vytvoření však nepředstavují výraznou zátěž, ve většině případů je lze zkopírovat z již implementovaného modulu a upravit závislosti, jména souborů a samotných tříd a v anotacích změnit cesty ke zdrojovým souborům s funkcemi pro právě implementovanou databázi.

2.3.6 Zdrojové soubory SQL funkcí

Zdrojové soubory[8] obsahují SQL kód databázových funkcí (anonymizačních i pomocných) a skripty pro vytvoření pomocných tabulek pro logování, konfiguraci a velikosti slovníků. Právě tyto zdrojové soubory představují při vytváření nových implementací nástroje Winch největší časovou zátěž. Je jich mnoho a kód každé z nich je potřeba připravit v procedurálním jazyce konkrétního databázového systému a patřičně odladit a otestovat. Tato práce představuje většinový podíl pracnosti implementace nového modulu Connectoru a proto se v dalších částech této práce budu věnovat hlavně pokusu o zlepšení a zefektivnění práce při přidávání těchto zdrojových SQL kódů do nového modulu.

2.4 Anonymizační funkce

Winch Connector pro provedení anonymizace jednotlivých osobních údajů používá databázové funkce. Pro každý osobní údaj má Winch implementovanou funkci, jejíž SQL kód pro konkrétní databázi je uložen ve zdrojovém souboru v příslušném modulu Connectoru.

Anonymizační funkce využívají k anonymizaci různé techniky, jako jsou výpočty s ASCII hodnotami jednotlivých znaků původní hodnoty, modulární aritmetika, maskování hodnot, nahrazování původních hodnot novými hodnotami z předpřipravených slovníků atp. Některé části kódů těchto funkcí se často opakují, například:

- kontrola validity vstupních dat
- kontrola nenulovosti vstupů a parametrů
- načtení konstant a parametrů z pomocné konfigurační tabulky
- volání pomocných funkcí a zpracování jejich výstupu

V některých případech jsou si však velmi podobné i celé funkce. To je hlavně případ funkcí, které k anonymizaci dat využívají hodnoty ze slovníků. Slovníkové funkce mají podobnou strukturu, několik z nich se mezi sebou liší dokonce jen konstantami, jako je název použitého slovníku, klíč k záznamu v konfigurační tabulce atd. Zbylé slovníkové funkce přidávají další specifické kontroly, případně v sobě kombinují jiné anonymizační funkce nebo skládají data z více než jednoho slovníku.

[8] resources

Příkladem takových funkcí může být funkce pro anonymizaci celého jména, která kombinuje funkce pro jméno a příjmení (obě slovníkové), nebo funkce pro anonymizaci čísla účtu, která pro zpracování kódu banky používá slovník, ale pro zbytek čísla má jiný postup.

Základ slovníkových funkcí je velmi podobný, struktura jednoduché slovníkové funkce vypadá následovně:

1. kontrola validity vstupu
2. zjištění počtu řádků ve slovníku
3. výpočet řádku slovníku, ze kterého bude vybrán výstup
4. kontrola a vrácení výstupu

Tyto podobnosti, jak mezi slovníkovými funkcemi navzájem, ale i opakujících se částí v kódech ostatních funkcí, jsou zohledněny v návrhu řešení, které je podrobněji rozebráno v následujících částech textu.

2.5 Návrh řešení

Jako první možnost řešení mě napadlo implementovat jednotlivé anonymizační funkce přímo v aplikaci Winch Connector a anonymizaci provádět tak, že data budou nahrána do aplikace, kde budou anonymizována a následně vrácena zpět do databáze. Tímto způsobem by potřeba zdrojových souborů s SQL kódy odpadla úplně, ovšem toto řešení bylo již dříve zamítnuto kvůli jeho značné nevýhodě, kterou je doba běhu anonymizace za takových podmínek. V databázích s velkým objemem dat by proces anonymizace trval příliš dlouho na to, aby bylo toto řešení prakticky použitelné, což je zapříčiněno nutností všechna data přenášet mezi aplikací a databází a trvání celého procesu tak mnohonásobně prodlužuje.

Místo toho se pokusím zjednodušení dosáhnout pomocí generování SQL kódu funkcí dynamicky, pomocí jazyka groovy, přímo v aplikaci Connector. Vzhledem k tomu, že na funkce je kladen požadavek identického chování v každém implementovaném databázovém systému, dá se předpokládat, že i jejich kód bude podobný, což potvrzuje již hotová implementace nástroje Winch pro Oracle, MSSQL, PostgreSQL a DB2. Implementace konkrétní funkce v různých databázích se liší hlavně syntaxí jednotlivých SQL dialektů a v možnostech procedurálních jazyků jednotlivých databázových systémů. Zároveň jsou si navzájem podobné i některé konkrétní funkce, což je zohledněno v návrhu šablon, které budou ke generování kódu využity a budou moci být mezi těmito podobnými funkcemi sdíleny. Aspoň část zdrojových souborů tedy půjde nahradit generovaným kódem, což programátorovi, pracujícímu na implementaci nástroje pro další databázový systém ušetří práci s jejich ruční implementací. V dalších kapitolách je tento návrh rozebrán podrobněji.

Kapitola 3

Architektura

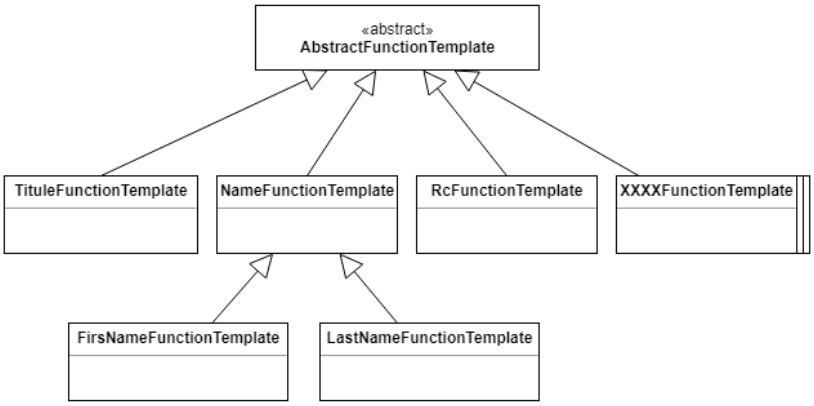
V této kapitole je vysvětlena struktura navrhovaného řešení. Josu ukázány hierarchie tříd, které se budou starat o generování potřebných SQL funkcí, jejich konkrétní úkoly i jejich provázanost se zbytkem systému.

Aby se vyřešila vzájemná podobnost některých jednotlivých anonymizačních funkcí, i podobnost jejich implementací napříč databázovými systémy, bude úkol generování rozdělen na dvě části. Vzájemnou podobu funkcí řeší šablony a podobnosti implementace v jednotlivých databázových systémech vyřeší generátory, které generují logické podcelky jednotlivých funkcí v příslušné SQL syntaxi a s využitím šablon tak sestaví výsledné zdrojové kódy funkcí pro konkrétní databázi.

3.1 Šablony funkcí

Šablony jsou implementovány jako třídy, které pomáhají vygenerování těla konkrétních anonymizačních funkcí a drží o nich potřebné informace, jako je „tvar“ funkce, její parametry, proměnné a návratové typy. Slouží tak v podstatě jako „pseudokód“ dané funkce. Jedna šablona může být ale využita i ke generování několika funkcí, čehož je při implementaci několikrát využito. Toho je docíleno předáním některých potřebných informací až za běhu programu, přímo mezi danou funkční třídou a jí příslušející šablonou. Šablony jsou navrženy tak, aby byly nezávislé na databázi, pro kterou jsou použity a tedy i na konkrétní implementaci Winche. Můžou tak být umístěny v základním modulu *disl-winch-connector*. Všechny třídy představující šablony rozšiřují abstraktní třídu AbstractFunctionTemplate, jejíž rozhraní musí každá šablona dodržet a z které podědí implementaci společné části vlastností a funkcionality.

Ukázka hierarchie šablon je vidět na obrázku 3.1



Obrázek 3.1: Hierarchie tříd představujících šablony anonymizačních funkcí

3.2 Generátory funkcí

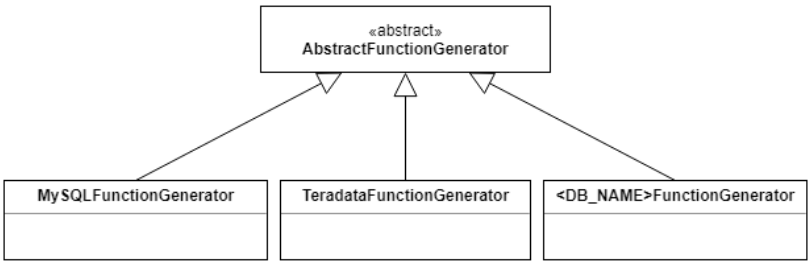
Generátory funkcí jsou třídy, které generují zdrojový kód jednotlivých potřebných konstruktů jazyka SQL v syntaxi konkrétní databáze, starají se o přiřazení správných datových typů a generují skript pro vytvoření funkce uvnitř databáze. Na rozdíl od šablon, které souvisí s konkrétní funkcí a jsou nezávislé na databázi, generátory jsou od konkrétní podoby funkcí úplně odstíněny. Kvůli odlišnostem v syntaxi a v možnostech jednotlivých databázových systémů musí být generátor implementován zvlášť pro každý databázový systém. Podobu tohoto generátoru udává abstraktní třída *AbstractGenerator*, která je umístěna v modulu *disl-winch-connector*. Do každého odvozeného modulu *disl-winch-*dbname**, kde chceme generátor použít, je nutné vytvořit jeho implementaci jako potomka této abstraktní třídy a doplnit funkcionalitu specifickou pro konkrétní databázový systém.

Ukázka hierarchie generátorů je vidět na obrázku 3.2

3.3 Provázanost se zbytkem systému

Každé funkční třídě, jejíž kód chceme generovat, je v konstruktoru vytvořena instance příslušné šablony, do které jsou třídou doplněny potřebné informace, jako je název výsledné vygenerované funkce, použitý slovník, konfigurační konstanty a další, v závislosti na konkrétní šabloně.

S instancí generátoru pak šablona komunikuje prostřednictvím abstraktní třídy StaticFunctionGenerator, která slouží k oddělení konkrétní implementace generátoru od použité šablony a zároveň svým rozhraním pro komunikaci s generátorem zlepšuje přehlednost kódu v šabloně. Tato statická třída tedy



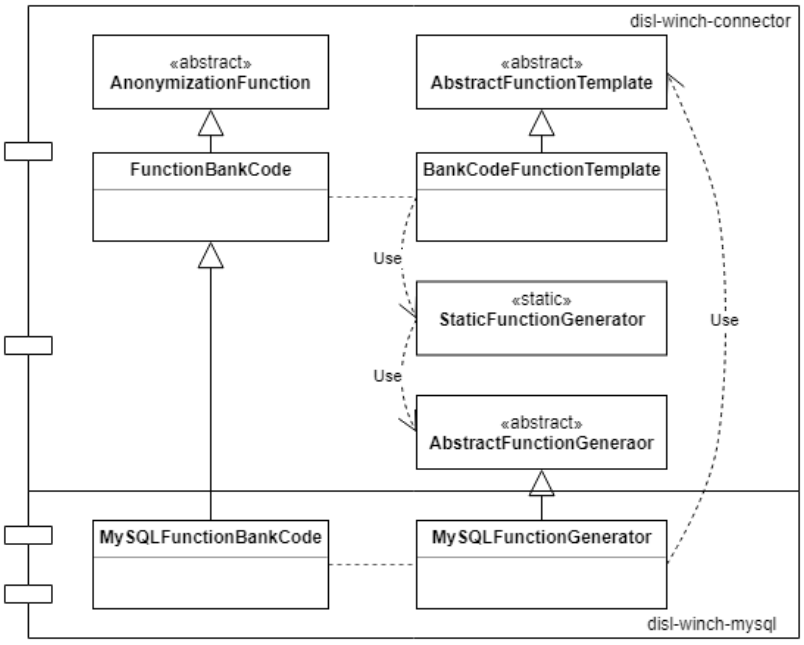
Obrázek 3.2: Hierarchie tříd představujících generátory kódu pro jednotlivé DB

figuruje jako fasáda [9], získává konkrétní implementaci generátoru od potomka třídy WAbstractDatabaseHelper, který je implementován v každém modulu Winch Connector a již v současné implementaci podobným způsobem vytváří a poskytuje instance dalších tříd specifických pro jednotlivé moduly. Tito potomci implementují návrhový vzor jedináček [10], jsou tedy v kódu přístupní odkudkoli (každý v rámci svého modulu) a nevytváří se neustále dokola nové instance.

Proces generování začíná tak, že funkce získá od příslušného *DatabaseHelperu* instanci generátoru a zavolá metodu *generateCreateQuery()*, které jako parametr předá odkaz na svou šablonu. Generátor pak od šablony získá tělo funkce, pomocí kterého poté vytvoří skript, sloužící k vytvoření funkce v dané databázi. Tento skript je poté generátorem vrácen volající funkční třídě.

Obrázek 3.3 ukazuje zapojení generátorů a šablon do původní struktury nástroje Winch.

[9] návrhový vzor Facade
[10] Singleton



Obrázek 3.3: Zapojení generátorů a šablon do zbytku systému

Kapitola 4

Implementace

Většina kódu aplikace Winch Connector je napsána v jazyce Groovy, místy kombinovaným s jazykem Java. V částech zdrojových kódů, kterých se implementované řešení dotýká, se využívá hlavně jazyk Groovy, ve kterém je implementováno i přidané řešení, využívající generátorů a šablon. Postupně probereme detailněji implementaci šablon a generátorů, včetně několika ukázek kódu a nakonec budou popsány problémy, které se v průběhu implementace objevily a jejich řešení a důsledky.

4.1 Šablony

Šablony tvoří hierarchii tříd vycházejících z abstraktní třídy AbstractFunctionTemplate. Část její implementace je ukázána zde:

```
abstract class AbstractFunctionTemplate {

    String END_LINE = COMMAND_SEPARATOR + "\r\n"
    String FUNCTION_NAME = "defaultFunctionName"
    String SCHEMA_NAME = "WINCH."
    String CONST_SETTING_NAME = "N/A"
    String RETURN_VARIABLE = "vresult"
    SQLDataType RETURN_TYPE = VARCHAR(4000)

    protected HashMap<String, SQLVariable> paramsMap = [:]
    protected HashMap<String, SQLVariable> varsMap = [:]

    abstract String getFunctionBody()

}
```

Textové proměnné v ukázce jsou používány pro formátování kódu a doplňování hodnot do šablon. Některé hodnoty je nutné v dědících třídách změnit, jiné jsou mnoha třídamy sdíleny, například návratová hodnota a její datový typ, které se opakují téměř u všech.

Návratový typ je uložený v proměnné typu *SQLDataType*, což je třída, která reprezentuje datové typy proměnných v databázi a pomáhá vygenerování správného kódu pro jejich deklaraci v různých databázových systémech. Implementace této třídy bude rozebrána dále v textu.

Šablona dále obsahuje dvě mapy pro uložení proměnných a parametrů a metody pro jejich nastavování. Tyto mapy jsou předávány generátoru, kteý podle nich vytvoří části SQL kódu pro deklaraci proměnných a definici parametrů pro generovanou funkci. Nakonec abstraktní třída *AbstractFunctionTemplate* obsahuje deklaraci metody *getFunctionBody*, kterou musí dědící třídy s šablonami implementovat. Ta slouží k získání zdrojového kódu těla funkce a je tedy také volána generátorem.

Jednotlivé šablony pak musí abstraktní třídu *AbstractFunctionTemplate* rozšířit a nastavit jednotlivé hodnoty na hodnoty odpovídající funkci, kterou generují. Implementace jednotlivých šablon se z hlavní části skládá z přetížení abstraktní metody *getFunctionBody()*, ve které je definováno tělo generované funkce. V konstruktoru se vytváří a ukládají obalovací objekty pro parametry a proměnné a nastavují se konfigurační konstanty, název funkce a další potřebné hodnoty. Některé šablony dále obsahují přetížení dalších metod z potomka, nebo proměnné s kousky SQL kódu a opakovaně používanými hodnotami. Pro demonstraci je přidána ukázka implementace šablony pro funkce na anonymizaci titulů, třídy *TituleFunctionTemplate*.

```
    TituleFunctionTemplate(String functionName, String dictName,
                           String dictColumnName) {
        super(functionName, dictName, dictColumnName)
        addParam(new SQLVariable("puvodni", VARCHAR(4000)))
        addParam(new SQLVariable("param", VARCHAR(4000)))
        addVar(new SQLVariable("vresult", VARCHAR(4000)))
    }

    String getFunctionBody() {
"""\
    ${CHECK_DEF_VALUES(var("puvodni"))}
    ${RETURN_ON_CONDITION(AND(CONDITION(var("puvodni"), IS_NULL()),
                              CONDITION(var("param"), IS_NULL())),
                  NULL)}
    ${SET_VAR(var("vresult"),
              SELECT_RANDOM_COLS_FROM(DICT_COLUMN_NAME, DICT_NAME))}
"""
    }
```

Tato šablona slouží k vygenerování hned dvou funkcí. Jsou to funkce na anonymizaci titulu před a za jménem, reprezentované třídami *FunctionTituleBefore* a *FunctionTituleAfter*. Tyto funkce patří mezi nejjednodušší, pouze vybírají náhodnou hodnotu z předpřipraveného slovníku. Mají tedy velmi krátký kód, což je vidět v ukázce, a zároveň se obě funkce liší pouze slovníkem, ze kterého hodnoty vybírají. Šablona se tak dá velmi snadno použít pro obě funkce. V ukázce je vidět, že funkce používá metody *CHECK_DEF_VALUES*, *RETURN_ON_CONDITION*, *SET_VAR*. Toto jsou metody statické třídy *StaticFunctionGenerator*, která je do šablony staticky importována a metody tak lze volat bez použití prefixu s názvem třídy, které patří. Tohoho přístupu bylo použito kvůli přehlednosti šablon. Tyto statické metody dále volají odpovídající metody konkrétní implementace Generátoru funkcí, jejíž instanci získají z příslušného *DatabaseHelperu*.

4.2 Generátory

Generátory jsou implementovány jako třídy s mnoha metodami, které slouží pro generování jednotlivých konstruktů jazyka SQL a jeho procedurálních nadstaveb. Pro použití těchto generátorů je v současné implementaci počítáno se znalostí jazyka SQL a jeho použití. Jak vstupy, tak i výstupy metod těchto generátorů jsou ve většině případů ve formě jednoduchých řetězců, tedy datového typu *String*. Generátor tedy nehlídá, zda jako parametr dostává skutečně hodnotu, kterou očekává, což může při nesprávném použití vést k nesmyslným výstupům. V budoucnu by bylo vhodné některé části zabalit do zvláštních objektů, což by způsobilo lepší kontrolu typů a bezpečnější používání. V rámci mé práce jsem se však snažil vyzkoušet, zda je takový přístup vůbec použitelný, k čemuž tento způsob implementace docela postačuje.

4.2.1 AbstractFunctionGenerator

Jako základ pro jednotlivé implementace generátorů, sloužících ke generování dílčích částí kódu pro konkrétní databázový systém, slouží abstraktní třída *AbstractFunctionGenerator*. Ta definuje funkcionalitu výsledných generátorů a je zde implementováno generování částí SQL, které mají společnou podobu ve více databázových systémech. Některé části, které jsou implementované už v této abstraktní třídě, jsou vidět v následujících ukázkách:

```
String INDENT = "\t\t"
String END_LINE = "\r\n"
String COMMAND_SEPARATOR = ";"
String EQUALS() { "=" }
String HIGHER_THAN() { ">" }
String HIGHER_OR_EQUAL() { ">=" }
String LESS_THAN() { "<" }
```

```
String LESS_OR_EQUAL() { "<=" }
String NOT_EQUALS() { "<>" }
String NOT_NULL() { "IS NOT NULL"}
String IS_NULL() { "IS NULL" }
String LIKE() { "LIKE" }
String getNULL() { "NULL" }

String var(String v) { v }
String str(String s) { "'${s}'"}

HashMap<SQLDataTypeEnum, String> DATA_TYPES

String translateDataType(SQLDataTypeEnum type) {
    DATA_TYPES[type]
}
```

Zde jsou vidět konstanty a metody sloužící k získávání SQL kódu jednotlivých logických operátorů a řetězců, sloužících k formátování generovaného kódu. Jsou společné pro většinu databází a proto je možné implementovat je již v základní třídě *AbstractFunctionGenerator*. Dále jsou zde metody *var()* a *str()*, které jsou v šablonách používány k „dekoraci“ proměnných a přidání správných uvozovek kolem řetězcových konstant. Některé databáze například vyžadují před názvem proměnné znak '@' (@nazev_proměnné), což má za úkol řešit právě funkce *var*. Nakonec je zde deklarace mapy *DATA_TYPES* a metody, která se stará o „překlad“ datových typů, tedy pro vstupní hodnotu typu *SQLDataTypeEnum* vrátí jeho správný zápis v konkrétní databázi. Pro správné fungování musí tuto mapu každá třída, dědící z *AbstractFunctionGenerator*, inicializovat páry hodnot, odpovídajícími příslušnému databázovému systému.

V další ukázce jsou vidět metody, generující dva hlavní konstrukty používané v procedurách, *WHILE* cyklus a podmíněný příkaz *IF*. Opět jsou použitelné ve více databázových systémech. Třetí metoda *BLOCK* slouží v generátoru a šablonách hlavně jako pomocná. Její úkol je *zabalit* více jednotlivých SQL příkazů do jednoho řetězce, což umožňuje v dalších metodách používat blok více příkazů jako jediný parametr.

```
String WHILE(String condition, String ... statements) {
    String str = """\
    While ${condition} DO
    """
    statements.each {
        str += indent(it.toString())
    }

    str += """    END WHILE${COMMAND_SEPARATOR}"""
    str
}

String IF_STATEMENT(String condition, String bodyTrue,
                    String bodyFalse = "") {
    String sql = """\
    IF ${condition} THEN
    ${indent(bodyTrue)}"""
    if (! bodyFalse.equalsIgnoreCase("")) {
        sql += """\
        ELSE
        ${indent(bodyFalse)}"""
    }
    sql += "END IF${COMMAND_SEPARATOR}"
    sql
}

String BLOCK(String ... statements) {
    String ret = ""
    statements.each{
        ret += it.toString() + END_LINE
    }
    ret
}
```

4.2.2 Generátory pro jednotlivé databáze

.

Každý modul s implementací nástroje Winch pro nějaký databázový systém potřebuje vlastní generátor, implementovaný na míru dané databázi. Tento generátor musí rozšiřovat abstraktní třídu *AbstractFunctionGenerator*, která je popsána v předchozí sekci. Pro implementaci podpory databázových systémů MySQL a Teradata bylo tedy do příslušných modulů třeba přidat odpovídající generátory *MySQLFunctionGenerator* a *TeradataFunctionGenerator*.

V případě databáze Teradata jsou pro účel anonymizace místo funkcí s

klasickou návratovou hodnotou použít procedury, jelikož SQL funkce v Teradata nepodporují „složitější“ konstrukty, jako jsou cykly, podmínky apod, které jsou v anonymizačních funkcích potřeba. Návratová hodnota je pak simulována pomocí výstupního parametru dané procedury. Rozhodl jsem se pro jejich generování použít stejné generátory, jako pro funkce ostatních databázových systémů. Bylo však potřeba několik úprav a zobecnění některých metod generátoru. Potřebné úpravy spočívaly například v úpravě podoby metody *RETURN()*, která musí být v Teradata řešena přířazením hodnoty do výstupního parametru a skokem na konec těla funkce. Další podstatný rozdíl je ve způsobu volání procedur a funkcí. Proceduru, narozdíl od funkce, není možné volat v rámci přiřazovacího příkazu, ani v podmínce, ale je třeba ji zavolat samostatně pomocí klíčového slova *call* a až poté získat a zpracovat výstup. Rozdíl ve volání můžeme vidět v následujících pseudokódech

```
-- Teradata
    call procedure_foo(tmp, in);
    if tmp = 1 then
        ...
    end if;

    call procedure_bar(output, input);
    set x = output;

-- MySQL
    if function_foo(in) = 1 then
        ...
    end if;

    set x = function_bar(input);
```

Rozdíl ve volání pomocných funkcí v přiřazovacím příkazu částečně řeší metoda generátoru *GET_FUNCTION_OUTPUT()*, už hotové podmínky v šablonách však bylo nutné předělat. Volání procedur v podmínkách je nahrazeno pomocnými proměnnými *hlp_str* a *hlp_int*, do kterých se jejich výstup musí uložit ještě před podmíněným příkazem. Tyto pomocné proměnné, stejně jako dříve zmíněný výstupní parametr, jsou generátorem so SQL kódu anonymizační funkce přidány automaticky metodami *prepareParams()* a *prepareVars()*, není tak třeba je přidávat zvlášť v každé šabloně, jako se přidávají ostatní proměnné. V generátorech i šablonách proběhly úpravy i v dalších metodách, které bylo potřeba více zobecnit, nejen kvůli databázi Teradata, ale potenciálně i pro ostatní databáze. Ve většině případů však stačilo přetěžovat původní metody a provést potřebné úpravy tam.

Některé odlišnosti mezi generátory pro MySQL a Teradata jsou vidět v následujících ukázkách. Hned v první ukázce vidíme rozdíl v již zmíněné me-

todě *RETURN()* pro ukončení funkce (u Teradata procedury) a navrácení výstupu.

```
// Teradata
@Override
String RETURN(String variable) {
"""\
    ${this.VARIABLE_ASSIGNMENT(var("voutput"), variable)}
    leave fc_body${COMMAND_SEPARATOR}"""
}

// MySQL
@Override
String RETURN(String variable) {
    "RETURN ${variable}${COMMAND_SEPARATOR}"
}
```

Následuje taktéž zmíněná metoda *GET_FUNCTION_OUTPUT()*, která řeší rozdíl v získávání výstupu z pomocných funkcí (v MySQL) a procedur (v Teradata).

```
// Teradata
@Override
String GET_FUNCTION_OUTPUT(String var,
                           String function,
                           String... params) {
    "call ${function}(${var}, ${params.join(", ")});"
}

// MySQL
@Override
String GET_FUNCTION_OUTPUT(String var,
                           String function,
                           String... params) {
    "${VARIABLE_ASSIGNMENT(var,
                           FUNCTION_CALL(function, params))}"
}
```

V další ukázce vidíme, že v Teradata bylo pro porovnání řetězce s regulárním výrazem nutné použít zabudovanou funkci *REGEXP_SIMILAR()*, jelikož v podmínce zde nelze použít logický binární operátor (přesto že jeho obdoba v Teradata také existuje), podobně jako *RLIKE* u MySQL.

```
// Teradata
@Override
String REGEX_LIKE(String what, String regex) {
    "REGEXP_SIMILAR(${what}, ${regex}) = 1"
}

// MySQL
@Override
String REGEX_LIKE(String what, String regex) {
    "(${what} RLIKE ${regex})"
}
```

Metoda *SELECT_RANDOM_COLS_FROM()*, sloužící k získání náhodného řádku z databázové tabulky, ukazuje, že rozdíly jsou i v samotných *SELECT* dotazech. V databázi Teradata je problém řešen pomocí výběru náhodného vzorku dat velikosti 1, naproti tomu v MySQL je účel splněn výběrem prvního řádku z náhodně seřazené tabulky.

```
// Teradata
@Override
String SELECT_RANDOM_COLS_FROM(String columns, String from) {
    "(SELECT ${columns} FROM "
            + "(SELECT * FROM ${from} SAMPLE 1) rnd_smpl)"
}

// MySQL
@Override
String SELECT_RANDOM_COLS_FROM(String columns, String from) {
    "(SELECT ${columns} FROM ${from} ORDER BY RAND() LIMIT 1)"
}
```

V poslední ukázce je příklad metody, která generuje SQL pro přetypování proměnné na celé číslo. Zde obě databáze používají funkci *CAST*, ovšem v databázi Teradata se pro určení výstupního datového typu používá konkrétní datový typ (včetně rozsahu, pokud by šlo o jiný typ, než INT), kdežto u databáze MySQL se používá obecnější *SIGNED*, označující číslo se znaménkem.

```
// Teradata
@Override
String TO_NUMBER(String value) {
    "CAST(${value} as INT)"
}

// MySQL
@Override
```

```
String TO_NUMBER(String value) {
    "CAST(${value} as SIGNED)"
}
```

Rozdílů mezi databázemi bylo samozřejmě mnohem více a některé z nich způsobovaly při takovémto použití v generátorech, kde bylo potřeba jednotného způsobu volání, problémy, kvůli kterým bylo třeba podobu jednotlivých metod generátoru v průběhu implementace měnit.

4.3 Pomocné třídy

4.3.1 StaticFunctionGenerator

Třída *StaticFunctionGenerator* slouží jednotlivým šablonám jako rozhraní pro komunikaci s generátorem pro konkrétná databázi. Obsahuje pouze statické metody, které jsou volány jednotlivými šablonami a které požadavek šablony předají instanci generátoru. Správná instance generátoru SQL funcí pro daný databázový systém je získána z odpovídajícího *DatabaseHelperu* takto:

```
AbstractFunctionGenerator gen =
    WDatabaseFactory.getDatabaseHelper().getFunctionGenerator()
```

Na této instanci je poté zavolána příslušná metoda a výsledek vrácen, opět skrze *StaticFunctionGenerator*, zpět šabloně. Jelikož jsou všechny metody *StaticFunctionGeneratoru* statické, nemusí se při jejich použití v šablonách používat prefix s instancí (tzv. dot notace), ale při použití statického importu

```
import static
    com.gem.winch.database.generators.StaticFunctionGenerator.*
```

je lze volat přímo. Rozdíl je ukázán zde:

```
    // volání s použitím instance generátoru
    generator.CHECK_DEF_VALUES(var("v")

        =>

    // volání se statickým importem StaticFunctionGenerator.*
    CHECK_DEF_VALUES(var("v")
```

Tento způsob volání pak způsobuje kratší zápis a výrazně zpřehledňuje kód v šablonách.

4.3.2 SQLVariable

Třída *SQLVariable* reprezentuje proměnnou nebo parametr databázové anonymizační funkce. Obsahuje informace o názvu proměnné, datovém typu (vnitřní proměnná typu *SQLDataType*), počáteční hodnotě a v případě použití pro definování parametru i o tom, jestli parametr slouží jako vstupní, výstupní nebo vstupně-výstupní (v databázích většinou IN, OUT, INOUT). Poslední informace se hodí především u databáze Teradata, kde je pomocí výstupních parametrů simulována návratová hodnota. Implementace této třídy je velice jednoduchá, nemá žádné vlastní metody. Její použití bude ukázáno v sekci 4.3.4 a její kód můžeme vidět zde:

```
class SQLVariable {

    SQLVariable(String name, SQLDataType type,
                String value = "", String io = "IN") {
        this.name = name
        this.type = type
        this.value = value
        this.io = io
    }

    public String io = "IN"
    public String name
    public SQLDataType type
    public String value
}
```

4.3.3 SQLDataTypeEnum

Zde se jedná o definici výčtového typu, obsahující jednotlivé datové typy v databázových systémech. Definice *SQLDataTypeEnum* obsahuje všechny datové typy, které byly použity při implementaci:

```
public enum SQLDataTypeEnum {
    VARCHAR, CHAR, INT, NUMBER, DECIMAL, FLOAT, BOOLEAN, DATE
}
```

4.3.4 SQLDataType

Třída, která představuje databázový datový typ. Tato třída má privátní konstruktor, místo kterého nabízí statické metody, které se starají o vytvoření instance, její správnou inicializaci a navrácení. Ty jsou pojmenovány dle datového typu, který vytváří a s kterým souvisí i parametry dané funkce. Stejně jako v databázích i v těchto metodách lze specifikovat rozsah a přesnost,

resp. délku v případě datového typu *VARCHAR*. Jelikož třída *SQLDataType* slouží k „obalení“ více různých datových typů, má každá instance informaci o konkrétním datovém typu, který představuje, uloženou ve vnitřní proměnné typu *SQLDataTypeEnum*, kam byla uložena již konstruktorem, volaným jednou z veřejných statických metod. Vytváření instancí pomocí statických metod má v tomto případě výhodu v tom, že částečně nahrazuje potřebu ručního nastavení konkrétního typu, ale také umožňuje vytvářet instance třídy *SQLDataType* podobným stylem, jako přímo v jazyce SQL. Použití této třídy je vidět u „deklarace“ proměnných v části šablony *RcFunctionTemplate* v následující ukázce, kde je použito hned několik implementovaných statických metod k vytváření instancí *SQLDataType*.

```
vars = [
    new SQLVariable(var("v_female"), NUMBER(5,0)),
    new SQLVariable(var("v_rc"), VARCHAR(4000), var("iv_rc")),
    new SQLVariable(var("v_year"), NUMBER(10,0)),
    new SQLVariable(var("v_month"), NUMBER(10,0)),
    new SQLVariable(var("v_day"), NUMBER(10,0)),
    new SQLVariable(var("v_a"), DATE, NULL),
    new SQLVariable(var("v_result"), NUMBER(19,0)),
    new SQLVariable(var("v_slash"), BOOLEAN, "false"),
    new SQLVariable(var("v_result_char"), VARCHAR(11))
]
```

4.4 Problémy a jejich řešení

V průběhu implementace se objevilo mnoho komplikací, jejichž řešení způsobovalo velké zdržení a které ve dvou případech vedly dokonce k nutnosti přepracovat velkou část již hotové implementace.

V prvním případě, kdy bylo nutné již částečně implementované řešení přepracovat, byla zvolena příliš málo obecná podoba šablon a generovaných částí SQL, nebylo by tak možné jednoduše pokrýt více funkcí a ukázalo se, že by takové řešení ve výsledku způsobovalo spíše více práce, kvůli nutnosti mnoho metod přetěžovat (a tedy znovu vytvářet potomky tříd v každém modulu). Původní podoba řešení by ušetřila práci hlavně u navzájem podobných anonymizačních funkcí, kterých ale není mnoho a ušetřená práce (nebo čas) by tak nevynahradila práci strávenou implementací generátoru a přetěžováním metod v šablonách.

Druhý případ nastal ve chvíli, kdy už byly šablony hotové a funkční pro databázi MySQL, na které byly šablony a celá funkčnost generování laděna a testována. Na začátku implementace generátoru pro databázi Teradata se ukázalo, že zde nebude možné pro účel anonymizace použít funkce. Teradata podporuje pouze velice jednoduchou podobu funkcí, ve kterých nelze

používat cykly, skoky, ani podmíněné příkazy a nelze v nich tedy implementovat složitější logika, kterou je nutno použít pro splnění požadavků, které jsou na anonymizační funkce Winche kladeny. Řešení tohoto problému již bylo podrobněji vysvětleno v kapitole 4.2.2 a znamenalo zásah téměř do všech již hotových šablon a několik úprav generátorů, s čímž souvisí i opakované testování a ladění funkcí, které změny narušily, a tedy i velkému nárůstu času sráveném u implementace.

Implementaci generování dále komplikovalo velké množství rozdílů v chování jednotlivých databázích za podobných okolností, ať už u funkcí a procedur, tak v některých případech i u syntaxe. Pro generování funkcí bylo třeba co nejuniverzálnějšího rozhraní pro generování jednotlivých částí SQL, a ne vždy tak mohl být použit nejjednodušší SQL kód, řešící daný problém, ale musel být zvolen takový, který bude vhodnější pro použití v generátoru a šablonách.

Tyto rozdíly umocňovala i samotná logika uvnitř anonymizačních funkcí, hlavně použití modulární aritmetiky a výpočtů s ASCII hodnotami jednotlivých znaků vstupního řetězce. Aby anonymizační funkce vracely správné výsledky, muselo se brát ohled i na kódování vstupních i výstupních řetězců a jejich různé konverze, ale i na velká a malá písmena apod. Práce s řetězci a chování funkcí, které je zpracovávají, se napříč databázemi různí. Příklady mohou být funkce *ASCII()* a *CHR()*, pro konverzi mezi znakem a jeho ASCII hodnotou, ale také jednoduché porovnávání řetězců operáorem „=“, který, dle typu databáze, rozlišuje nebo nerozlišuje velká a malá písmena.

Podobných menších rozdílů bylo velice mnoho a v průběhu implementace byly neustále objevovány další a generovátory i šablony se s nimi musejí být schopny vypořádat. Do budoucna se tedy v řešení počítá i s tím, že některé problémy nepůjdou vyřešit pouhou drobnou úpravou šablony či generátoru a proto je zachována zpětná kompatibilita se starým způsobem přidávání anonymizačních funkcí (pomocí zdrojových souborů).

Kapitola 5

Testování

V této kapitole je rozebráno testování a ladění jak jednotlivých generátorů a šablon, tak i výsledných anonymizačních funkcí.

Testování databázových funkcí a procedur zahrnovalo i ruční testování a ladění přímo na obou přidávaných databázích. Databázi MySQL jsem měl při testování spuštěnou přímo na počítači použitém k vývoji a pro práci s ní jsem použil prostředí MySQL Workbench 14.0. Databáze Teradata je distribuována v podobě obrazu linuxového systému spustitelného pomocí virtualizačního nástorje WMVare, kde je databáze vytvořena, nastavena a připravena k provozu ihned po spuštění. K připojení k Teradata databázi jsem použil prostředí Teradata Studio Express. Na použitém počítači však virtualizace, společně s ostatními potřebnými nástroji, způsobovala výrazné zpomalení běhu, což se také podepsalo na trvání celého testování. Když byla možnost, testovací databáze byla spouštěna na jiném počítači v lokální síti, což testování anonymizačních funkcí v databázi Teradata velice usnadnilo, bohužel to ale šlo málokdy.

Z celé práce představovalo testování a ladění největší podíl času. Celkový čas zabraný u testování a ladění mnohonásobně předčil počáteční očekávání. Tato část práce byla komplikována problémy s rychlostí vývojového prostředí po spuštění všech potřebných nástrojů, kterých je mnoho a jsou výkonově náročné, ale i celková složitost testovacího prostředí. Hlavní část času si pak vyžádo testování a ladění databázových funkcí a procedur, ať už těch, které byly přidávány ručně v podobě zdrojových souborů s SQL kódem daných funkcí, tak i nově generovaných, k jejichž otestování byly částečně použity unit testy. Jelikož jsou anonymizační funkce navrženy tak, aby se z anonymizované hodnoty „nedala“ zjistit hodnota původní, je v nich často používána například modulární aritmetika, počítání s ASCII hodnotami znaků anonymizovaných řetězců, maskování hodnot apod.

Všechny tyto techniky, společně s nedostatkem mých předchozích zkušeností s databázemi Teradata a MySQL, měly za důsledek obtížné ladění databázových funkcí v případě nesprávných výstupů, ale také implementovaných generátorů

a šablon, které bylo potřeba v průběhu celé implementace stále přizpůsobovat a opravovat vznikající chyby. Tato část práce - testování, prolínající se částečně s implementací - tak zabrala nejvíce času z celé práce.

5.1 Pomocné SQL funkce

Při implementaci některých pomocných databázových funkcí jsem jako vzor použil zdrojové SQL kódy odpovídajících funkcí z již hotových implementací nátroje Winch, další byly implementovány rozdílným způsobem, vhodnějším v konkrétní databázi. Tyto funkce byly testovány ručně hned po jejich implementaci přímo v prostředí odpovídající databáze, tedy MySQLWorkbench nebo Teradata Studio Express. Znovu pak byly testovány při ladění anonymizačních funkcí, které je používají, což pomohlo objevit další chyby, na které se nepřišlo už při implementaci.

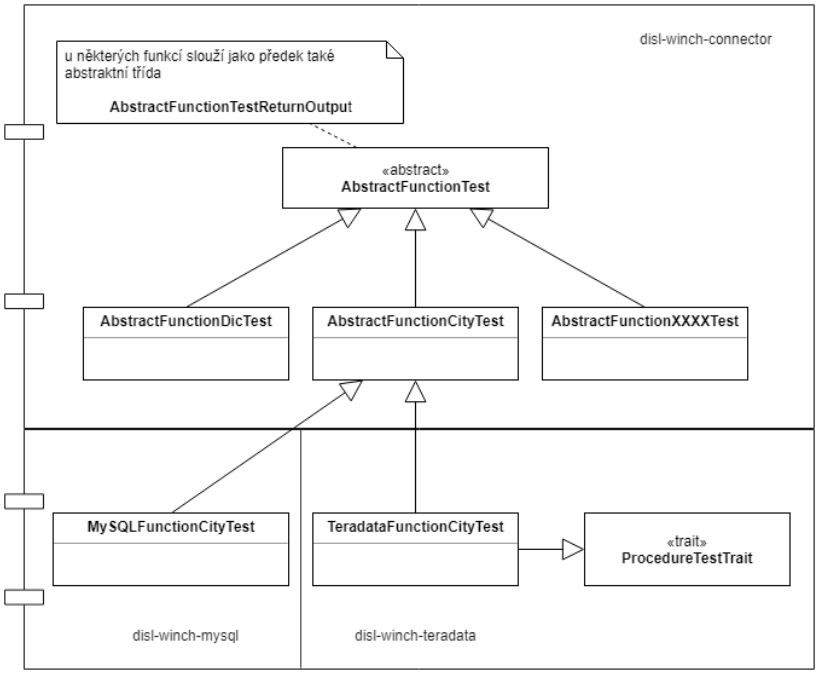
5.2 Generátory, šablony a anonymizační funkce

Generátory anonymizačních funkcí i jejich šablony byly testovány průběžně při jejich implementaci. Ze začátku byly implementovány obecněji a většina z generovaných funkcí v databázi nefungovala, takže šablony bylo třeba doopravit při testování generovaných anonymizačních funkcí. Stejně tak bylo třeba průběžně upravovat jednotlivé metody generátorů, které často bylo třeba zobecnit nebo jinak změnit pro větší univerzálnost v použití pro více databázových systémů.

K testování jednotlivých vygenerovaných funkcí byly použity jednotkové testy, pro jejichž vytvoření sloužily jako základ abstraktní testovací třídy z modulu disl-winch-connector. Ty obsahují vzorové vstupy, očekávané výstupy a napříč databázemi sdílené testovací metody, sloužící k nasazení SQL kódu do databáze, volání testovaných funkcí a porovnání výsledků. Hierarchii testů můžeme vidět na obrázku 5.1

Základem je abstraktní třída, která dodává logiku pro volání testovaných funkcí v databázi a vyhodnocení výsledků. Z ní jsou odvozeny další abstraktní třídy pro anonymizační funkce jednotlivých osobních údajů. Ty slouží jako základ pro další třídy, které již náleží konkrétní databázi a patří tedy do příslušného modulu.

V případě databáze Teradata však bylo třeba změnit způsob volání anonymizačních funkcí a získání jejich výstupu. V případě Teradata, kde jsou anonymizační funkce ve skutečnosti implementovány (generovány) jako procedury, bylo třeba je i jinak testovat. Procedury se v Teradata volají jinak a návratovou hodnotu je možné pouze simulovat pomocí výstupního parametru. Jelikož metody, které se o toto starají, jsou již v úplném základí abstraktní třídě hierarchie testovacích tříd a počítá pouze s funkcemi a návrtovými hodnotami, použil jsem k nahrazení této funkcionality Groovy Trait (něco mezi



Obrázek 5.1: Diagram hierarchie testovacích tříd

rozhraním a abstraktní třídou) *ProcedureTestTrait*, kterou implementují jednotlivé testovací třídy pro Teradata a jejímž kódem se potřebné zděděné funkce překryly.

Nejvíce chyb bylo způsobováno samotnou složitostí anonymizační logiky a jejich opravování zabralonejvíce času. Mohlo za to hlavně použití modulární aritmetiky a také počítání s ASCII hodnotami, které může být ovlivněno kódováním, případně různými konverzemi uvnitř databáze. V případě nesprávných výstupů pak bylo složtié dopočítat, kde přesně chyba vznikla chyba vznikla.

Kapitola 6

Zhodnocení a návrh budoucího rozvoje

6.1 Zhodnocení

Implementace Teradata i MySQL za pomoci generování SQL kódu anonymizačních funkcí ukázala, že je skutečně reálné pomocí šablon funkce generovat najednou pro více databázových systémů, ještě lépe pak generovat více podobných funkcí pomocí jediné šablony, jako v případě slovníkových funkcí. Implementace se však neobešla bez průběžných úprav generátorů i šablon při řešení problémů, způsobených rozdílnými možnostmi a omezeními obou databází. Přesto, že Teradata se od ostatních již implementovaných databází liší více, než zbylé implementace navzájem (a to nejen kvůli nutnosti použít procedury místo funkcí), implementovat společnou podobu šablon bylo možné. Nedá se ale říci, že použití řešení v příštích implementacích nástroje pro další databázové systémy by neodhalilo další rozdíly, s kterými generátor ani šablony zatím nepočítají a které mohou vynutit další úpravy v šablonách.

Jelikož šablony jsou sdíleny mezi jednotlivými implementacemi, každá jejich úprava může rozbít generování v již hotové a odladěné implementaci a je tak nutné každou takovou změnu otestovat ve všech databázových systémech, pro které je daná šablona použita. Jednotlivým třídám, reprezentujícím anonymizační funkce, však lze SQL kód stále doplnit starým způsobem pomocí zdrojových souborů (resources) a tak může generování pomoci, i když se nepodaří vygenerovat kompilovatelný zdrojový kód. Díky šabloně by takový kód měl mít alespoň správný „tvar“ a je tedy stále možné ho ručně doupravit a použít, aniž by bylo nutné upravovat a znovu testovat a ladit danou šablonu, případně generátor.

Na základě podobnosti SQL kódu funkcí napříč ostatními již implementovanými databázovými systémy a jejich podobnosti vzhledem k MySQL, kde generování funguje, se dá soudit, že by podobné řešení šlo bez velkých změn aplikovat i pro ně. Jelikož ty už mají SQL kódy hotové, dává zde implemen-

tace generování smysl pouze v případě, že by docházelo k častějším úpravám podoby funkcí nebo požadavků na ně kladených, případně mělo-li by přibýt dostatečné množství nových funkcí na anonymizaci dalších typů osobních dat.

Výhoda řešení pomocí šablon a generátorů je, že přidávání nové funkce do modulů, kde je generování již implementováno, znamená pouze přidání jedné šablony. Dále ušetří mnoho zdrojových souborů, které je třeba udržovat a každou změnu v nich promítat do všech modulů. Místo toho je nyní třeba upravit jen příslušnou šablonu a změna se projeví ve všech modulech, je pouze nutné v nich změnu otestovat, což však bylo třeba v obou případech.

Nevýhodami pak může být již zmíněné nebezpečí, že úpravou jedné šablony rozbijeme již hotové a odladěné funkce. Použití jednotného rozhraní generátoru a jím nabízených metod pro generování částí SQL, ale také společné šablony pro jednolivé funkce, způsobují, že možnosti, jak vygenerovaný kód optimalizovat a zefektivnit, jsou velice omezené. Optimální kód v jednom databázovém systému nemusí být optimální v ostatních. Kvůli šablonám je ale „tvar“ vygenerovaných funkcí daný a bez implementace nové šablony, nebo ručních úprav výsledného SQL kódu, ho nelze snadno změnit.

6.2 Budoucí rozvoj

Implementovanému řešení by v budoucnu prospěl drobný refaktoring kódu. Velice by také pomohlo přepracování implementací generátorů způsobem, který by generátory udělal robustnější a méně náchylné k chybám. Aktuální implementace generátoru ve svých metodách přijímá jako vstup pouze stringy, stejně tak vrací pouze stringové výstupy. Programátor tak může dát jakýkoli vstup jakékoli funkci, což generátor nehlídá a programátor tak musí vědět, co dělá. V případě předání špatných vstupů, nebo vstupů ve špatném pořadí, by se generovaly funkce generovaly špatně a nešly by ani zkompilovat. Celkové spolehlivosti generování, ale i „uživatelské“ přívětivosti by tak pomohl návrh nějaké sady tříd, které vstupy a výstupy (řetězce s částmi SQL kódu) zaobalí, což by zacházení s generátorem usnadnilo a zmenšilo by tak pravděpodobnost nesprávného použití a chybného generování.

U generování funkcí bych se zaměřil více na podobnost funkcí mezi sebou, než podobnosti stejných funkcí napříč databázovými systémy. Generování funkcí pro více databází pomocí jediné šablony sice možné je, ale takový postup je zbytečně složitý, zabere více času a omezuje možnosti optimalizace generovaných funkcí, navíc s sebou nese riziko rozbití funkcí při úpravě šablony.

Dále by orientaci v kódu, ale i jeho údržbě a spolehlivosti, pomohl celkový refaktoring Connectoru. Ten stále obsahuje velké množství zbytečných tříd, které nepřináší nic nového a liší se pouze drobnostmi, daly by se tak snadno nahradit například pomocí „továrny“, která by jejich instance vytvářela z jedné společné třídy a odlišnosi (často jde jen o konstanty apod.) doplňovala dynamicky. Konstanty by se daly přesunout do zdrojových souborů a malé kousky

kódu je v Groovy možné přidávat pomocí *Closures*, případně je implementovat uvnitr *Traits*[11]a objekt „navěsit“.

[11] n

Závěr

Cílem práce bylo zefektivnit vytváření implementací nástroje Winch pro nové databázové systémy a zároveň snížit množství vznikajících duplicit v kódu. Zvolený postup se ukázal jako v praxi realizovatelný.

Podařilo se implementovat generátory a šablony, které jsou schopné jednotným způsobem generovat validní kód pro databáze MySQL i Teradata. Navržený postup tak vede minimálně ke snížení duplicit.Konkrétně se výrazně snížilo množství potřebných souborů s SQL kódy, které dříve musely být opakovaně implementovány pro každou databázi zvlášť. Odladění takto „univerzálního“ generátoru však zabere dost času, v dalším vývoji bych se proto zaměřil spíše na větší využití vzájemné podoby funkcí (jako u slovníkových) v rámci jedné databáze, než na jejich podobnost mezi moduly. To by vedlo ke snadnější implementacia kratšímu ladění.Zároveň by se minimalizovala rizika, uvedená v kapitole 6.

I přes všechny zmiňované rozdíly mezi databázemi a komplikace při jejich ladění se podařilo většinu anonymizačních funkcí úspěšně vygenerovat a odladit. I kdyby se kvůli časově náročné implementaci samotného generátoru neušetřil čas oproti dřívějšímu způsobu přidávání anonymizačních funkcí, stále je zde zřejmá výhoda, že se stejná funkce neopakuje v každém modulu, ale je jen na jednom místě. To usnadňuje údržbu i orientaci v kódu. V případě, že se nepodaří generování odladit úplně, pořád je možné vygenerovaný kód upravit ručně a přidat ho původním způsobem. Proto jsou generátory užitečné i v případě dílčího neúspěchu při generování.

Vzhledem k tomu, že postup funguje u databází s tak rozdílnými možnostmi, jako jsou Teradata a MySQL, lze se oprávněně domnívat, že bude možno jej s relativně malými modifikacemi aplikovat i na další typy databázových systémů.

Literatura

[1] Smítka, P.: *Datové řezy a anonymizace*. Bakalářská práce, Fakulta informačních technologií, České vysoké učení technické v Praze, 2013.

[2] Schuh, M.: *Implementace datové vrstvy pro anonymizační nástroj*. Bakalářská práce, Fakulta informačních technologií, České vysoké učení technické v Praze, 2018.

[3] Skalský, D.: *Vyhledávání osobních údajů v relačních databázích*. Bakalářská práce, Fakulta informačních technologií, České vysoké učení technické v Praze, 2017.

[4] Apache: *Groovy Language Documentation [online]*. 2019-05-07 [cit. 2019-06-27]. Dostupné z: `http://docs.groovy-lang.org/docs/latest/html/documentation/`

[5] baeldung: *Closures in Groovy [online]*. 2019-03-23 [cit. 2019-06-27]. Dostupné z: `https://www.baeldung.com/groovy-closures`

[6] baeldung: *An Introduction to Traits in Groovy [online]*. 2019-03-06 [cit. 2019-06-27]. Dostupné z: `https://www.baeldung.com/groovy-traits`

[7] *Traits [online]*. 2019-05-31 [cit. 2019-06-27]. Dostupné z: `http://docs.groovy-lang.org/next/html/documentation/core-traits.html`

[8] Klein, H.: *Groovy Goodness: 'String', "Strings", /Strings/ and GStrings [online]*. 2009-08-20 [cit. 2019-06-27]. Dostupné z: `https://mrhaki.blogspot.com/2009/08/groovy-goodness-string-strings-strings.html`

[9] *Teradata Documentation [online]*. ©2019 [cit. 2019-06-27]. Dostupné z: `https://docs.teradata.com/landing-page/`

[10] *Teradata Tutorial [online]*. ©2019 [cit. 2019-06-27]. Dostupné z: `https://www.tutorialspoint.com/teradata`

[11] *Teradata String Functions [online]*. ©2018 [cit. 2019-06-27]. Dostupné z: https://dbmstutorials.com/teradata/teradata_string_functions.html

[12] *MySQL Functions [online]*. ©1999-2019 [cit. 2019-06-27]. Dostupné z: https://www.w3schools.com/sql/sql_ref_mysql.asp

[13] *MySQL 8.0 Reference Manual [online]*. ©1999-2019 [cit. 2019-06-27]. Dostupné z: https://dev.mysql.com/doc/refman/8.0/en/

Příloha **A**

Seznam použitých zkratek

DB databáze

DBMS Database Management System - systém řízení báze dat

SQL Structured Query Language - strukturovananý dotazovací jazyk

MSSQL databáze Microsoft SQL Server

ASCII American Standard Code for Information Interchange (znaková sada)

DB2 databázový systém firmy IBM

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `BP_Brychta_Ondrej_2019.pdf` ..... bakalářská práce ve formátu pdf
- `Zadani.pdf` ..... zadání bakalářské práce
- `src` ..... zdrojové kódy implementace
  - `disl-winch-connector`
  - `disl-winch-mysql`
  - `disl-winch-teradata`
- `thesis` ..... zdrojová forma práce ve formátu LaTeX