ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Kontrola kvality konceptuálního modelu
Student: Jan Novák
Vedoucí: Ing. Jiří Hunka
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2020/21

Pokyny pro vypracování

V rámci portálu dbs.fit.cvut.cz je prováděno testování studentů ze znalosti konceptuálního modelování. Realizujte proto nástroj na automatické nalezení rozdílů mezi dvěma konceptuálními modely, který bude využit pro tuto automatickou opravu. Při realizaci je třeba vzít v úvahu, že výsledek automatického porovnání musí být možné následně upravit manuálně, což provede vyučující v případě, že obdobné nedostatky nevykázalo podobné předchozí řešení. Při realizaci je třeba klást důraz na správnost výstupu automatické opravy.

Postupujte v těchto krocích:
1. Analyzujte současný stav a možnosti starého nástroje pro konceptuální modelování, "Kreslítka", v portálu v souvislosti s opravou.
2. Analyzujte možnosti nově vznikajícího nástroje DSM.
3. Na základě analýzy proveďte návrh konkrétního řešení.
4. Proveďte prototypovou implementaci.
5. Navrhněte vhodné testovací scénáře a implementujte je pomocí automatických testů.
6. Realizujte výsledné řešení pro nástroj DSM.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 11. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Kontrola kvality konceptuálního modelu

Jan Novák

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

17. září 2020

Poděkování

Rád bych poděkoval za trpělivost a rady při vedení této práce a týmu v předmětu BI-SP2. Dále bych rád poděkoval Bc. Filipu Dolníkovi za jeho konzultace a rady ohledně implementace práce. V neposlední řadě bych chtěl poděkovat rodině, přítelkyni a přátelům za jejich podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při 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 předpisu, 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 citovaného zákona.

V Praze dne 17. září 2020 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Jan Novák. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Novák, Jan. *Kontrola kvality konceptuálního modelu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce se zabývá návrhem a implementací modulu automatické opravy testových otázek, zaměřených na schopnost návrhu konceptuálního modelu podle zadání. Práce vzniká jako součást projektu DSM, který má za cíl nahradit stávající aplikaci Kreslítko jakožto nástroj pro podporu tvorby diagramů konceptuálních modelů. Projekt DSM vzniká jako multiplatformní projekt v jazyce Kotlin, který následně přeložen do jazyka JavaScript pro klientskou (webovou) část a jazyka Java pro serverovou část projektu.

Klíčová slova databázové systémy, automatická oprava, konceptuální model, přiřazovací problém, maďarský algoritmus, Kotlin

Abstract

This thesis describes design and implementation of a module for automatic correction of exam questions focussed on the ability to design a conceptual model according to an assignment. This thesis has been created as a part of the DSM project, which aims to replace the current application Kreslitko as a tool for drawing conceptual model diagrams. The DSM project is being implemented as mutliplatform project in Kotlin programming language which is then compiled into JavaScript for the client (web) side or into Java for the server side.

Keywords database systems, automatic correction, conceptual model, assignment problem, hungarian algorithm, Kotlin

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza **5**
2.1 Používané pojmy 5
2.2 Analýza současného stavu Kreslítka 5
2.2.1 Řešení automatické opravy 6
2.2.2 Výstup automatické opravy v případě nalezení chyb . . 6
2.3 Database schema modeller 6
2.3.1 Technologie 6
2.3.1.1 Datová třída 8
2.3.1.2 Rozšiřující funkce 8
2.3.2 Model 9

3 Návrh řešení **13**
3.1 Doménový model automatické opravy 13
3.2 Mapování odevzdané odpovědi na vzor 13
3.2.1 Definice 14
3.3 Přiřazovací problém 15
3.4 Maďarská metoda 16
3.4.1 Nezávislé nuly a krycí čáry 16
3.4.2 Postup 17
3.4.3 Transformace M:N vazby 19

4 Vývoj a implementace **21**
4.1 Vývoj 21
4.2 Model automatické opravy 21
4.3 Třída AssignmentSolver 23

4.4 Třída DBModelCorrector 26
4.5 Pomocné třídy a rozšiřující funkce 26
4.5.1 Třída Decomposition 26
4.5.2 Třída ScoreValues 26
4.5.3 Rozšiřující funkce 27
4.5.3.1 Funkce *compareAll()* 27
4.5.3.2 Funkce *createInstance()* 27

5 Testování **29**

Závěr **31**

Bibliografie **33**

A Seznam použitých zkratek **35**

B Obsah přiloženého CD **37**

Seznam obrázků

2.1 Diagram se zvýrazněnými chybnými úseky 7
2.2 Vzorové řešení 8
2.3 Doménový model - canvas [4] 10
2.4 Doménový model - entita [4] 11
2.5 Doménový model - relace [4] 12

3.1 Doménový model - database model corrector 14
3.2 Příklad M:N relace 20
3.3 Dekompozice M:N relace - slabá entita 20
3.4 Dekompozice M:N relace - silná entita 20

4.1 Diagram tříd - Database model corrector 24

Seznam tabulek

3.1 Příklad přiřazovací úlohy 15

4.1 Přiklad nastavení malusů pro rozdíly v atributech 26

Úvod

V rámci předmětu Databázové systémy jsou každoročně testovány stovky studentů. Testování probíhá na portálu dbs.fit.cvut.cz ve formě semestrální práce, zápočtového testu a závěrečné zkoušky. Pro zajištění plynulosti a kvality oprav zápočtových testů a závěrečných zkoušek jsou pro portál vyvíjeny aplikace podporující automatickou opravu testových otázek a případné předzpracování studentovo odpovědi v případě, že je její hodnocení postoupeno vyučujícímu.

Tato bakalářská práce se zabývá návrhem a implementací modulu automatické opravy testových otázek, zaměřených na schopnost navrhnout konceptuální model podle zadání, a dále návrhem testovacích scénářů a jejich implementací jako automatických testů. Samotná implementace modulu vzniká jako součást projektu DSM.

Projekt DSM je nově vznikající aplikace pro portál dbs.fit.cvut.cz, která je vyvíjena v rámci předmětů BI-SP 1 a BI-SP 2. Cílem projektu je nahradit stávající aplikaci, na podporu kreslení diagramů, Kreslítko, která již nevyhovuje požadavkům předmětu a z vývojového hlediska se stal dlouhodobě neudržitelným. Účast na projetku v rámci předmětu BI-SP 2 pod vedením Ing. Jiřího Hunky a Bc.Filipa Dolníka a možnost podílet se na reálně využívaném projektu byla hlavní motivací autora pro vypracovaní této práce.

Hlavním přínosem této práce je zdokonalení portálu dbs.fit.cvut.cz, zjednodušení a zkvalitnění práce vyučujících při opravování odpovědí.

V 1. kapitole jsou popsány cíle práce.

2. kapitola popisuje aktuální stav automatické opravy diagramů a aplikace Kreslítko na portálu dbs.fit.cvut.cz a jeho aktuální nedostatky. Dále kapitola obsahuje popis nově vznikajícího projektu DSM, který má v budoucnu nahradit aplikaci Kreslítko. Analýza obsahuje popis nového doménového modelu, který byl rozšířen pro rostoucí potřeby a ambice portálu dbs.fit.cvut.cz, dále jsou zde uvedeny příklady výhod jazyka Kotlin, který nahradil JavaScript, jako hlavní vývojový jazyk aplikace pro podporu kreslení diagramů konceptuálních modelu.

3. kapitola popisuje návrh řešení modulu automatické opravy pro projekt DSM. Je zde povrchně popsán jednoduchý doménový model, se kterým algoritmus automatické opravy pracuje a dále jsou uvedeny důvody jeho vzniku v porovnání s předchozí verzí modulu opravy. Následuje popis problému mapování částí diagramu odevzdaného studentem na odpovídající části vzorové řešení. Následuje krátká diskuze nad možnými řešeními, které jsou implementačně i teoreticky jednodušší, ale nevyhovují potřebám předmětu Databázové systémy. Dále jsou zadefinovány a popsány pojmy z teorie potřebné k pochopení přiřazovacího problému a jsou uvedeny dva pohledy na daný problém, první pohled problém popisuje spíše z praktického hlediska, druhý uvedený pohled využívá teorii grafů.

4. kapitola se věnuje popisu samotné implementace. Obsahuje detailnější popis modelu automatické opravy včetně diagramu tříd. Dále je popsán způsob kvantifikace ohodnocení jednotlivých porovnávaných částí diagramů a stěžejních tříd a funkcí modulu. Součástí popisu implementace metody *getScore()* a funkce *createInstance()* jsou i krátké diskuze popisující úskalí implementace.

V 5. a poslední kapitole je popsán postup při testování modulu.

Kapitola **1**

Cíl práce

Hlavním cílem práce je návrh a implementace řešení automatické opravy testových otázek zaměřených na schopnost navrhnout konceptuální model. Samotné řešení je součástí projektu DSM, který v budoucnu nahradí aplikaci Kreslítko, jako nástroje na tvorbu diagramů.

Práce se bude skládat z popisu aktuálního řešení automatické opravy, analýzy projektu DSM a technologií použitých při jeho vývoji. Následovat bude návrh nového robustnějšího řešení pracujícího s novým upraveným modelem a popis samotné implementace, který by měl částečně sloužit i jako dokumentace projektu.

Kapitola 2

Analýza

Následující kapitola popisuje aktuální stav aplikace Kreslítko v souvislosti s automatickou opravou diagramů konceptuálních modelů na portálu dbs.fit.cvut.cz.

2.1 Používané pojmy

- **portál DBS** - podpůrný portál předmětu DBS dbs.fit.cvut.cz
- **Kreslítko** - aplikace aktuálně nasazená na portálu DBS
- **DSM** - *Database schema modeller* - aktuálně vyvíjený projekt nahrazující Kreslítko
- **newDBS** - projekt samotného portálu DBS, napsaný v jazyce PHP, který zajišťuje tvoru HTML stránek a zastřešuje všechny aplikace a moduly
- **Tralex** - nástroj na podporu tvorby zjednodušeného relační modelu na portálu DBS

2.2 Analýza současného stavu Kreslítka

Kreslítko ve své aktuální podobě na portálu DBS neobsahuje funkcionalitu automatické opravy konceptuálního modelu. Automatické testování všech testových otázek na portálu DBS zajišťuje třída *AutoCorrect*, která je součástí projektu newDBS. Samotné porovnání vzorového a studentem odevzdaného modelu zajišťuje třída *DiagramComparator*, přesněji řečeno její statická metoda *compareDiagrams*.

2.2.1 Řešení automatické opravy

Automatická oprava studentova diagramu probíhá následovně. Po ukončení testu je studentovo řešení postupně porovnáno se všemi vzorovými odpověďmi pomocí funkce *compareDiagrams*. Funkce následně vrátí seznam nalezených rozdílů. V případě, že existuje vzorové řešení identické s opravovaným diagramem, je studentovy automaticky přidělen plný počet bodů. V opačném případě je vybrána dvojice studentovo řešení – vzor s největší vzájemnou shodou. A oprava je dále postoupena vyučujícímu k manuální opravě.

2.2.2 Výstup automatické opravy v případě nalezení chyb

V případě, že automatická oprava nalezla chyby ve studentově řešení a hodnocení je postoupeno k manuální opravě, jsou vyučujícímu prezentovány dva diagramy: vzor a odevzdané řešení s vyznačenými chybnými úseky, viz diagramy 2.1 a 2.2

Kreslítko v tomto případně nabízí opravujícímu následující funkce pro manipulaci s opraveným diagramem:

- zvýraznit chybný úsek diagramu
- smazat zvýraznění úseku diagramu
- manipulace s rozložením diagramu
- přidat komentář k úseku diagramu

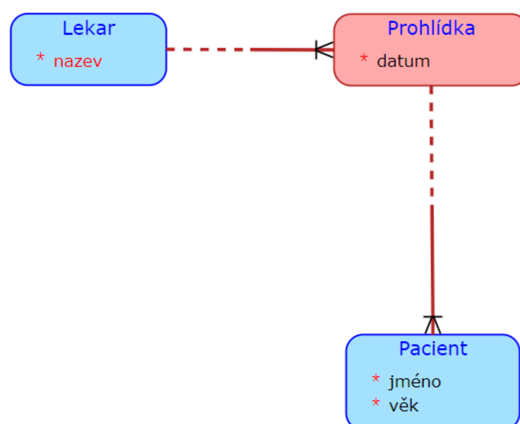
2.3 Database schema modeller

Projekt DSM je nově vznikající aplikace pro portál DBS v rámci předmětů BI-SP 1 a BI-SP 2. Cílem projektu je nahradit stávající aplikaci na podporu kreslení diagramů, Kreslítko, která již nevyhovuje požadavkům předmětu a z vývojového hlediska se stala dlouhodobě neudržitelným.

2.3.1 Technologie

Výstup projektů DSM a Kreslítko je v zásadě stejný, oba projekty generují JavaScriptovou aplikaci. Programovací jazyky, použité pro vývoj je však zásadně liší. Zatímco projekt Krelítko je vyvíjen v JavaScriptu, projekt DSM je postaven na jazyce Kotlin, který je až následně přeložen do JavaScriptové aplikace. Důvodem této zásadní změny technologií je zkvalitnění a zrychlení práce vývojářů a zjednodušení budoucí údržby projektu.[1]

Kotlin, který je odvozen od jazyka Java a je považován za jeho nástupce, nabízí všechny funkcionality svého předchůdce, avšak v mnohem čitelnější



Obrázek 2.1: Diagram se zvýrazněnými chybnými úseky

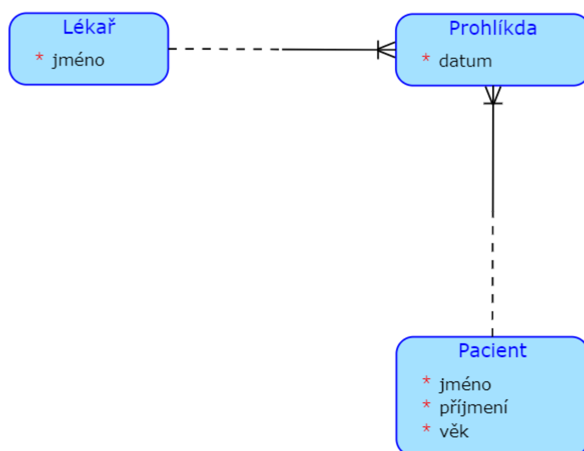
a úspornější formě. Toho Kotlin dosahuje díky své úspornější syntaxi, v porovnání s Javou, a svých specifických konstruktů, jakými jsou například: datová třída, rozšiřující funkce, nebo přetěžování operátorů, z nichž první dva příklady jsou detailněji popsány v následujících podkapitolách.

Dalším cílem jazyka Kotlin je omezení výskytu runtime výjimek.

> Jednou z nejčastějších nástrah řady programovacích jazyků, Javu nevyjímaje, je skutečnost, že pokus o přístup ke členu reference ukazující na hodnotu null skončí tzv. null reference výjimkou. V Javě by se jednalo o NullPointerException, nebo zkráceně NPE.

[2]

Kotlin tento problém řeší zdvojením všech definovaných datových typů. Například u třídy *Integer* se jedná o *Int* a *Int?*, kde *Int?* může obsahovat referenci na *null* a jakýkoliv neošetřený přístup k jeho členům skončí chybou při kompilaci. Jelikož je nullabilní typ implicitně nepřevoditelný na svůj nenullabilní protějšek, stejnou chybou skončí i případ použití hodnoty typu *Int?* při volání funkce očekávající parametr typu *Int*.[2]



Obrázek 2.2: Vzorové řešení

2.3.1.1 Datová třída

Jak již název napovídá, hlavním účelem datové třídy je držení dat. Kompilátor na základě parametrů konstruktoru vygeneruje metody *equals()*, která umožňuje strukturální porovnání dvou objektů, *hashCode()*, *toString()* a *copy()* pro vytvoření hluboké kopie objektu. Na příkladu 2.1 je vidět nejenom ukázka definice datové třídy, ale i úspornost zápisu, dosažené sloučením definice konstruktoru třídy a deklarací atributů.

```
1 data class Square(
2     var xCoordinate: Double,
3     var yCoordinate: Double,
4     val edgeSize: Double,
5     var label: String?
6     )
```

Zdrojový kód 2.1: Ukázka datové třídy

2.3.1.2 Rozšiřující funkce

Rozšiřující funkce je další z konstruktů jazyka Kotlin, pomáhající docílit větší čitelnosti zdrojového kódu, rozšiřovat funkcionalitu nejenom námi definovaných tříd, ale i tříd z knihoven třetích stran, do kterých nelze jinak zasahovat,

bez nutnosti použití dekorátoru, nebo vytvořením podtřídy a omezení viditelnosti rozšířené funkcionality pouze na nezbytně velký rozsah.[3]

Na příkladu 4.1 funkce *getAllBiggerThan* rozšiřuje funkcionalitu všech objektů typu *Collection<Rectangle>*, do jehož implementace nelze nijak zasahovat, zároveň je funkcionalita přístupná pouze uvnitř třídy *ShapeService*.

```
1  class ShapeService {
2
3      public fun foo(rectangles: Collection<Rectangle>, size: Int) {
4          rectangles.getAllBiggerThan(size)
5      }
6
7      private fun Collection<Rectangle>.getAllBiggerThan(size: Double)
           : Collection<Rectangle> {
8          return this.filter { r -> Math.pow(r.edgeSize, 2) > size}
9      }
10
11 }
```

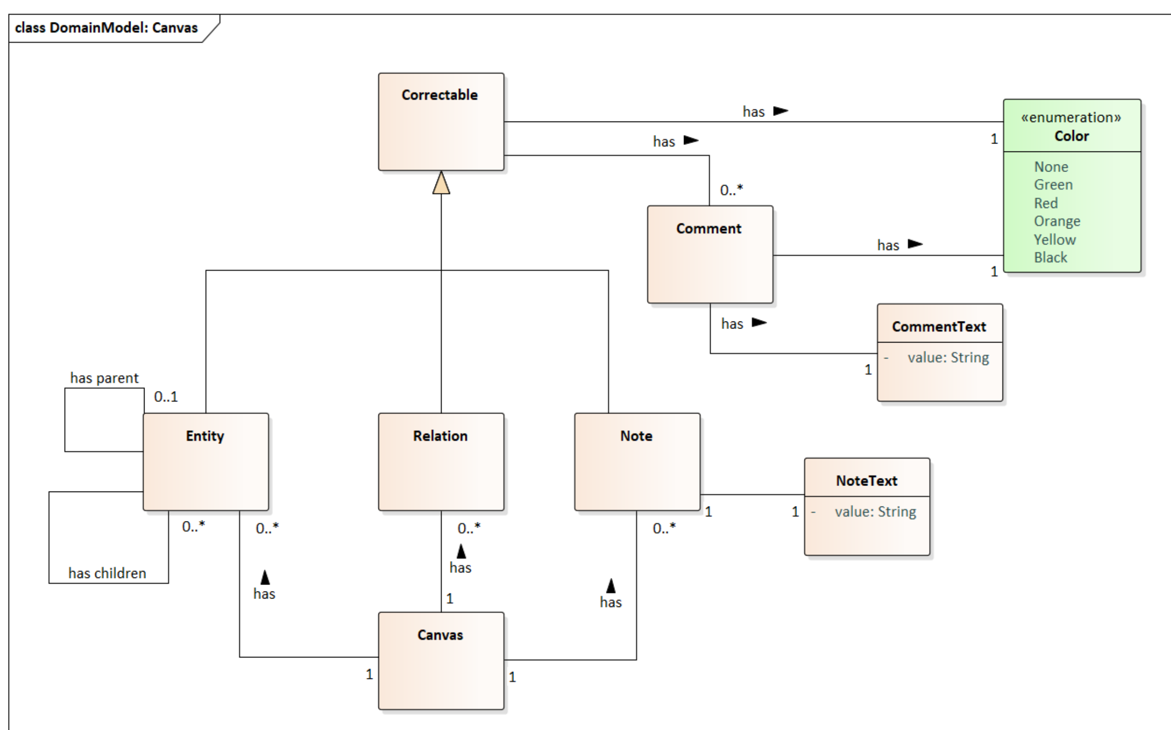
Zdrojový kód 2.2: Ukázka rozšiřující funkce

2.3.2 Model

Značnými úpravami prošel i samotný model diagramu, jelikož již nevyhovoval novým požadavkům kladeným na projekt DSM. Nový model byl navržen robustněji a obsahuje detailnější informace o diagramu. Třídou reprezentující celý diagram je třída *Canvas*, která obsahuje seznam všech entit, relací a poznámek viz. diagram 2.3.

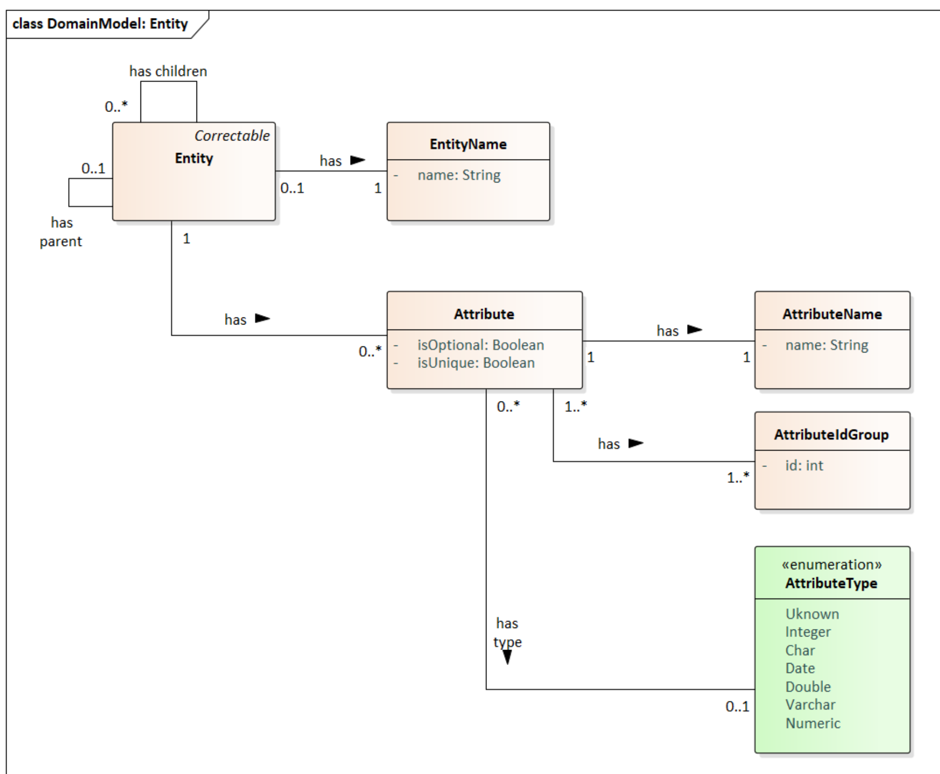
Instance třídy Entity v sobě obsahuje informaci o jejím názvu ve formě datové třídy *EntityName*, množinu (Set) všech svých atributů, pro potřeby reprezentace IS-A hierarchie dále obsahuje odkaz na svého rodiče a množinu všech potomků. V neposlední řadě je v instanci třídy uložena informace o všech relacích ve formě množiny odkazů na příslušící třídu *RelationEnd* dané relace, tato informace slouží primárně pro potřeby funkce importu a exportu diagramu do formátu SQL Developeru. V případě třídy entity se nejedná o datovou třídu z důvodu vytvoření cyklu ve vygenerované metodě *equals* způsobeného vztahem potomek-rodič. Pro účely automatické opravy implementuje třída Entity rozhraní *Correctable*, které jí umožní přiřadit barvu a případné komentáře vyučujícího, nebo vygenerované automatickou opravou.

Instance třídy *Relace*, zobrazené na diagramu 2.5 obsahuje informaci o svém názvu ve formě datové třídy *RelationName*, podobně jako třída *Entity*. Dále obsahuje množinu instancí *RelationAttribute*, tato množina reprezentuje přenos cizích klíčů v případě identifikační závislosti entity. Tuto informaci aktuálně využívá aplikace Tralex pro reprezentaci zjednodušeného relační zápisu. Instance si dále udržuje informaci o směru relace a reference na oba své konce, připojené k Entitám, ve formě instancí třídy *RelationEnd*. Podobně jako třída Entity i třída *Relation* implementuje rozhraní *Correctable* pro účely opravy.

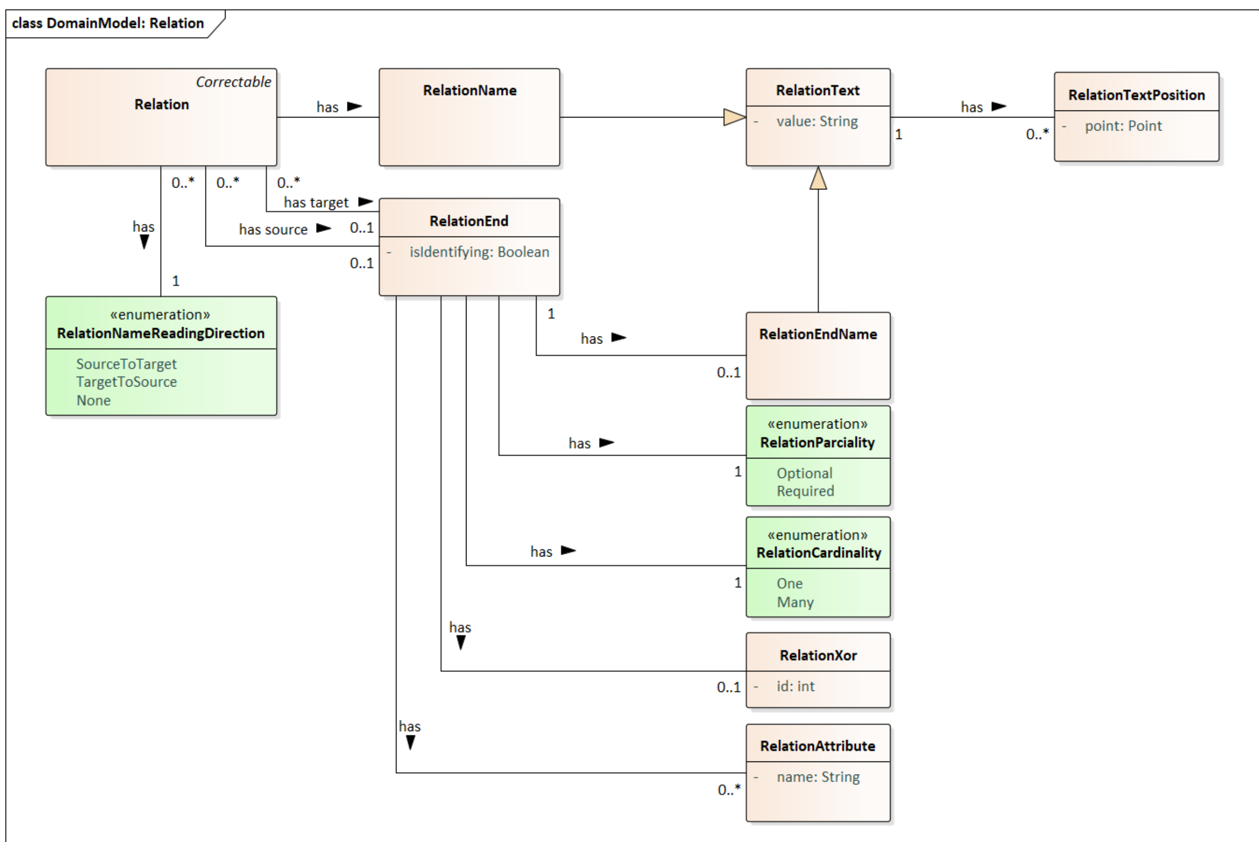


Obrázek 2.3: Doménový model - canvas [4]

Třída *RelationEnd* v sobě zaobaluje informace určující vztah jednotlivých cílových entity v relaci. Obsahuje název konce relace ve formě datové třídy *RelationEndName*, dále obsahuje informaci o parcialitě a kardinalitě příslušné strany relace, zda se jedná o slabou či silnou entitu v dané relaci a zda patří do výlučného vztahu s jinou relací. Třída *RelationEnd* implementuje rozhraní *Correctable*.



Obrázek 2.4: Doménový model - entita [4]



Obrázek 2.5: Doménový model - relace [4]

Kapitola 3

Návrh řešení

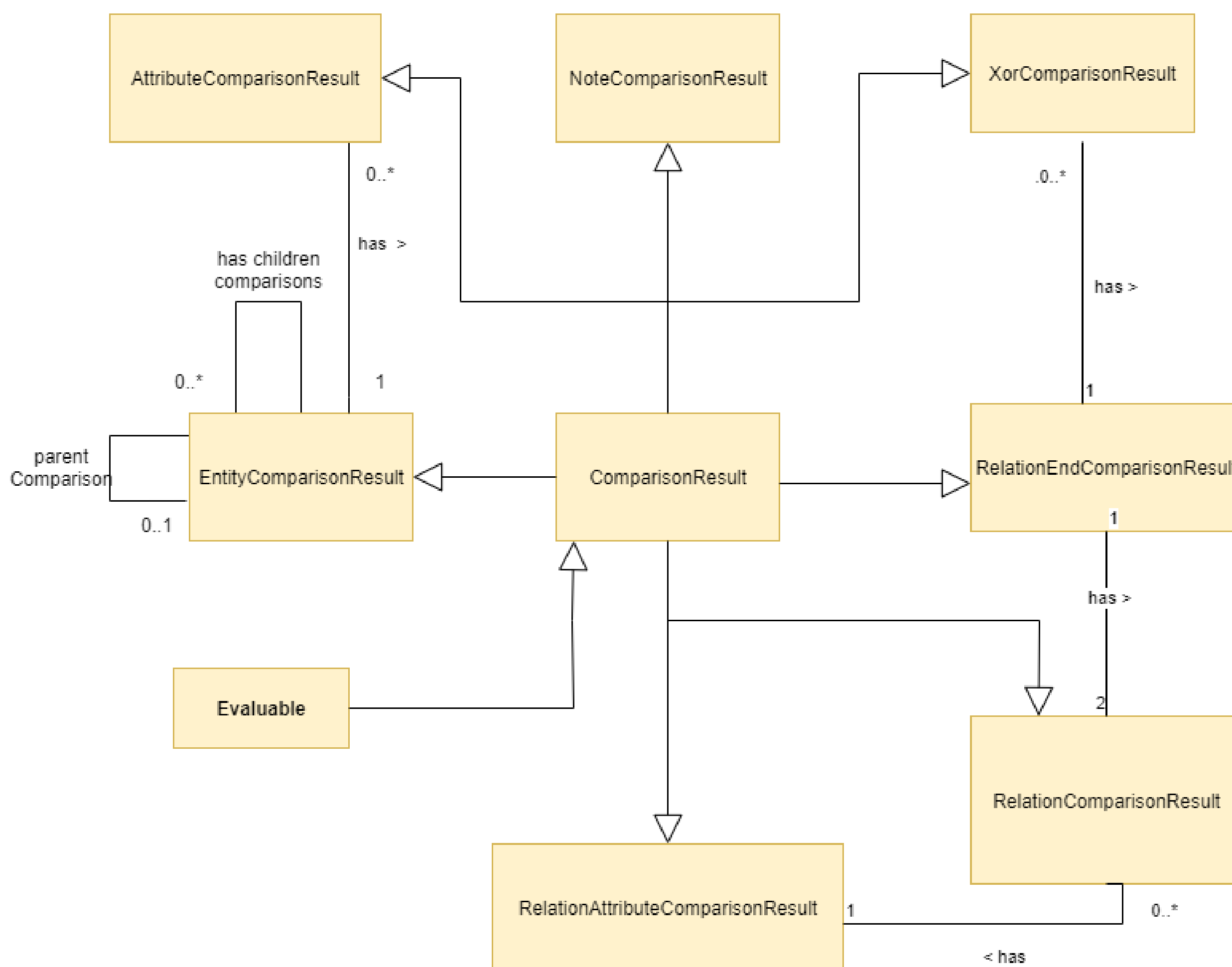
Následující kapitola popisuje návrh řešení automatické opravy. Nejprve je popsán doménový model, se kterým automatická oprava pracuje. Dále je definován a popsán problém mapování částí diagramu, odevzdaného ke korekci, na své protějšky ze vzorového diagramu. Následuje samotný algoritmus mapování. Poslední část kapitoly se věnuje dekompozici vztahů M:N, jakožto rozdílným přesto ekvivalentním zápisům, na které je nutné brát zřetel při opravě.

3.1 Doménový model automatické opravy

Předchozí řešení automatické opravy bylo velice jednoduché a pracovalo přímo s modelem diagramu. Pro potřeby nového robustnějšího algoritmu bylo potřeba navrhnout jednoduchý model tříd viz. diagram 3.1, sloužící k uložení informací o podobnosti jednotlivých porovnávaných dvojic mezi kroky algoritmu a generování diagramu s vyznačenými chybami pro účely manuální opravy. Samotný model je podrobněji popsán v kapitole *Implementace*.

3.2 Mapování odevzdané odpovědi na vzor

Zřejmě největším problémem automatické opravy je postup mapování částí odevzdaného diagramu na odpovídající protějšky ve vzorovém řešení. Nejjednodušší přímočaré řešení je průchod všech možných řešení. To by však znamenalo vytvoření všech možných permutací, což odpovídá časové složitosti $O(n!)$, takové řešení je však nepřípustné, protože by srovnání i relativně jednoduchých diagramů bylo příliš výpočetně náročné. Jako další možné řešení se nabízí forma hladového algoritmu, který by postupně procházel studentovo diagram a v každém kroku vybírat aktuálně nejlépe ohodnocenou dvojici. Tento postup by našel řešení v lineárním čase, ale nalezené řešení by se nemuselo shodovat s celkově nejlepším řešením. Při použití toho postupu by docházelo



Obrázek 3.1: Doménový model - database model corrector

nejenom k znevýhodňování některých studentů při hodnocení, ale v krajních případech by mohl výsledek kontroly naopak ztížit manuální opravu vyučujícímu svým absurdním výsledkem. Jednotlivé části diagramů je tedy nutné porovnávat v souvislosti s celkem. Postup, který tento problém řeší je popsán v následujících podkapitolách.

3.2.1 Definice

Neorientovaný graf je uspořádaná dvojice (V, E), kde V je neprázdná množina vrcholů a E je množina hran. Hrana je dvouprvková podmnožina V.

Úplný bipartitní graf $K_{n_1}, K_{n_2})$, kde $n_1 > 0$ a $n_2 > 0$ tvořený dvěma partitami o n_1 a n_2 je graf $(A \cup B, \{\{a, b\} | a \in A \wedge b \in B\})$, kde $A \cap B = \emptyset$ $|A| = n_1$ a $|B| = n_2$.

Ohodnocený graf vznikne přiřazením reálného čísla každé hraně $e \in E$. **Párování** v grafu G je množina všech hran $M \subseteq E(G)$ takových, že každý vrchol grafu G patří do nejvýše jedné hrany z M. Párování se nazývá **perfektní**, pokud má právě $|V(G)|/2$ hran, neboli každý vrchol patří právě do jedné hrany z M.

3.3 Přiřazovací problém

Přiřazovací problém je optimalizační úloha, ve které máme na vstupu dvě množiny A a B a a ohodnocení jednotlivých párů například z množiny reálných čísel $(\{\{a,b\},c\}|a \in A \wedge b \in B \wedge c \in R)$ a cílem je nalezení navzájem disjunktních párů $\{a,b\}$ tak, aby součet jejich ohodnocení byl nejnižší možný. Množiny A a B nemusí být stejně velké, tento problém lze však doplněním menší množiny pracovními prvky, jejichž páry později z výsledku odstraní, proto se bude dále v textu předpokládat stejná velikost obou vstupních množin.

Problém si lze ukázat na příkladu plánování práce: Na vstupu máme množinu pracovníků a množinu úkolů, které je potřeba splnit v co nejkratším čase. Každý z pracovníků je jinak zkušený a různé úkoly mu tím pádem trvají různou dobu. Ohodnocení párů je v tomto případě doba, kterou potřebuje daný pracovník pro dokončení úkolu viz. tabulka 3.1.

úkol \ Prac.	**Bob**	**Adam**	**Helmut**	**Petr**
úkol 1	10	15	50	30
úkol 2	50	35	25	30
úkol 3	15	10	30	25
úkol 4	40	15	30	10

Tabulka 3.1: Příklad přiřazovací úlohy

Řešení by v tomto případě vypadalo následovně:

úkol	**pracovník**
úkol 1	Bob
úkol 2	Helmut
úkol 3	Adam
úkol 4	Petr

Na samotný přiřazovací problém existuje další pohled tentokrát z teorie grafů. V tomto případě reprezentujeme množinu vstupů jako úplný ohodnocený bipartitní graf, kde množiny vrcholů reprezentují pracovníky a úkoly. Výsledek je v tomto případě reprezentován množinou hran, která pokrývá všechny vrcholy obou partit a jejíž součet vah všech hran je nejnižší možný. [5]

3.4 Maďarská metoda

Tato sekce popisuje maďarskou metodu řešící problém přiřazení. Podstatnou částí maďarské metody je hledání nezávislých nul a tvorba krycích čar. Před samotným představením metody jsou definovány tyto pojmy a je popsán způsob jakým hledat nezávisle nuly. Pro větší přehlednost budou ohodnocení jednotlivých párů reprezentováno maticí namísto tabulkou.

3.4.1 Nezávislé nuly a krycí čáry

Mějme matici o rozměrech $n \times n$, která obsahuje nulové prvky. Jednotlivé nulové prvky vybereme tak, aby v každém řádku a sloupci byla vybrána nejvýše jedna nula. Takto nalezené nuly nazýváme nezávislými nulami. Postup pro nalezení nejvyššího počtu nezávislých nul je následující:

1. Vyškrtneme všechny řádky matice, ve který se nevyskytuje žádná nula

2. Ve zbývajících řádcích najdeme ty, ve kterých se vyskytuje jedna nula. Vybereme jeden z těchto řádků. Nulu v něm označíme za nezávislou a řádek vyškrtneme. Ve zbývajících řádcích opakujeme 1. a 2. krok, dokud je to možné.

3. Jestliže je v každém řádku a sloupci více než jedna nula, najdeme řádek s nejmenším počtem nul. Jednu z nul v tomto řádku označíme za nezávislou a řádek vyškrtneme. Opakujeme 2. a 3. krok dokud je to možné.

Pro ověření, zda jsme našli maximální počet nezávislých nul v matici použijeme Königovu větu, která popisuje vztah mezi minimálním počtem krycích čar pokrývajících všechny nuly v matici a maximálním počtem nezávislých nul v matici. Krycí čárou se rozumí vodorovná, nebo svislá úsečka, která pokrývající řádek matice.

Königova věta: Maximální počet nezávislých nul, které je možné v dané matici vybrat, se rovná minimálním počtu krycích čar, jimiž je možno pokrýt všechny nuly v matici.[6]

Postup pro konstrukci minimálního počtu krycích čar je následující:

1. Vybereme řádek, který neobsahuje nezávislé nuly. Přes sloupce, ve kterých se tyto nezávislé nuly nachází, vedeme vertikální úsečky. Tento krok opakujeme, dokud se v matici takovéto řádky vyskytují.
2. Pokud v matici zůstaly nepokryté nuly, vedeme krycí čáry přes řádky, ve kterých se tyto nepokryté nuly vyskytují.

3.4.2 Postup

V této podkapitole jsou popsány jednotlivé kroky algoritmu na konkrétním příkladu.

$$\begin{pmatrix} 9 & 14 & 11 & 17 \\ 4 & 7 & 12 & 6 \\ 8 & 6 & 10 & 13 \\ 14 & 12 & 13 & 13 \end{pmatrix}$$

1. krok: V prvním kroku provedeme redukci řádků a sloupců matice. Pro každý řádek matice najdeme jeho minimum a odečteme jej od celého řádku.

   $$\begin{pmatrix} 0 & 5 & 2 & 8 \\ 0 & 3 & 8 & 2 \\ 2 & 0 & 4 & 7 \\ 2 & 0 & 1 & 1 \end{pmatrix}$$

   Dále upravíme sloupce matice podle stejného principu jako řádky.

   $$\begin{pmatrix} 0 & 5 & 1 & 7 \\ 0 & 3 & 7 & 1 \\ 2 & 0 & 3 & 6 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

2. krok: V druhém kroku nalezneme maximální počet nezávislých nul a nakreslíme krycí čáry podle postupu uvedeného v sekci 3.4.1. Pokud zjistíme, že matice obsahuje stejný počet nezávislých nul jako má matice řádků, algoritmus je u konce. Výsledný tvar dvojic zjistíme podle souřadnic nezávislých nul. V případě, že počet nezávislých nul neodpovídá počtu řádků, pokračujeme 3. krokem. (Nezávislé nuly jsou podtrženy.)

$$\begin{pmatrix} \underline{0} & 5 & 1 & 7 \\ 0 & 3 & 7 & 1 \\ 2 & \underline{0} & 3 & 6 \\ 2 & 0 & \underline{0} & 0 \end{pmatrix}$$

3. krok: Ve třetím kroku pracujeme s krycími čarami, které jsme nakreslili v druhém kroku. V matici nejprve nalezneme minimální hodnotu nepřekrytou žádnou čarou, tuto hodnotu označíme x. Poté odečteme x od každého nepřekrytého řádku a naopak přičteme ke každému překrytému sloupci. Kombinací těchto dvou kroků může pro každý prvek matice nastat jedna z následujících možností:

   - Pokud se jedná o nepřekrytý prvek, jeho hodnota se sníží o x.
   - Pokud je prvek jednou překrytý, jeho hodnota se nezmění.
   - Pokud je prvek překrytý dvakrát, jeho hodnota se zvýší o x.

$$\begin{pmatrix} 0 & 5 & 1 & 7 \\ 0 & 3 & 7 & 1 \\ 2 & 0 & 3 & 6 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Výsledná matice má po těchto úpravách následující tvar:

$$\begin{pmatrix} 0 & 4 & 0 & 6 \\ 0 & 2 & 6 & 0 \\ 3 & 0 & 3 & 6 \\ 3 & 0 & 0 & 0 \end{pmatrix}$$

Výslednou matici dále upravujeme podle druhého kroku.

Dokončení příkladu: Vrácením se k druhému kroku dostaneme:

$$\begin{pmatrix} \underline{0} & 4 & 0 & 6 \\ 0 & 2 & 6 & \underline{0} \\ 3 & \underline{0} & 3 & 6 \\ 3 & 0 & \underline{0} & 0 \end{pmatrix}$$

Počet nalezených nezávislých nul odpovídá počtu řádků matice. Nalezli jsme tedy optimální řešení pro dané zadaní.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Pro příklad uvedený výše existuje pouze jedno optimální řešení. Může však nastat situace, kdy těchto řešení existuje více. V tom případě algoritmus vrátí pouze jedno z nich.

3.4.3 Transformace M:N vazby

Předešlé řešení nebralo v potaz možné ekvivalentní zápisy vazby M:N. Při vytváření zadaní museli vyučující hlídat a případně explicitně určit, zda mají být M:N vazby dekomponovány a jakým způsobem. Nově navržené řešení automatické opravy se tento nedostatek snaží vyřešit. Nutno podotknout, že výsledek manuální dekompozice vazby se může, i přes použití stejného postupu, u jednotlivých řešitelů lišit např.: v názvu pomocné entity.

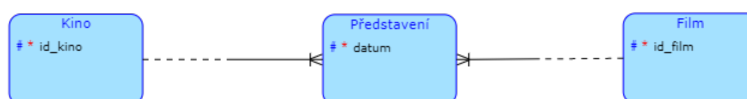
M:N vazbu, která je uvedena na příkladu 3.2 je možné dekomponovat třemi různými způsoby [7]:

1. využitím silné entity Představení s vlastním primárním atributem *datum* a dvou 1:N vazeb viz. diagram 3.4

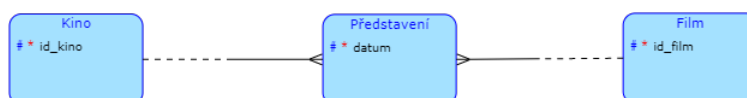
2. využitím slabé entity identifikačně závislé na cizích primárních atributech *id_kino id_film* a svého primárního atributu *datum* viz diagram 3.3

3. využitím slabé entity bez vlastního primárního atributu, v tomto případě však může jedno kino hrát daný film nejvýše jednou.

Obrázek 3.2: Příklad M:N relace



Obrázek 3.3: Dekompozice M:N relace - slabá entita



Obrázek 3.4: Dekompozice M:N relace - silná entita

Kapitola 4

Vývoj a implementace

V této kapitole je popsána metodika vývoje a detaily implementace automatické opravy. Dále je uvedeno několik příkladů testovacích scénářů, které byly implementovány v rámci unit testů pro kontrolu konzistence a správnosti implementace i při budoucím vývoji na projektu DSM.

4.1 Vývoj

Pro vývoj aplikace byla použita vodopádová metoda. Pro použití vodopádové metody je nutné, aby bylo zadání práce zcela jasně definováno již na počátku vývoje. Výhodou této metody je časová a rozsahová predikovatelnost. Nevýhodou tohoto postupu je pomalá rychlost dodání funkčního programu. Toto je způsobeno tím, že jednotlivé fáze vývoje, tj. analýza–design–implementace–testování se zabývají celou aplikací najednou a každá fáze může započít až ve chvíli, kdy jsou hotové a uzavřené všechny ji předcházející fáze. Ze stejného důvodu není vhodné používat vodopádovou metodu na projekty, u kterých změny zadaní uprostřed vývoje. Vzhledem k tomu, že se neočekávají úpravy doménového modelu DSM ani zadaní a s přihlédnutím k samotné podstatě problému automatické opravy, kdy první funkční verze řešení až po dokončení celé implementace, se vodopádová metoda jeví jako nejlepší volba.[8]

Jak již bylo zmíněno v předchozích kapitolách, vývoj modulu probíhal v jazyce Kotlin kompilovaného do JavaScriptu. Diky tomu bude budoucí nasazení tohoto modulu s celou aplikací DSM na portál dbs.fit.cvut.cz, psaný v jazyce PHP, jednodušší. Pro verzování modulu je použit verzovací systém Git, běžící na fakultním portále gitlab.fit.cvut.cz.

4.2 Model automatické opravy

Předešlá implementace automatické opravy využívala k porovnávání modelů jejich zápisy ve formátu JSON a mohla tudíž pracovat přímo s entitami da-

ných diagramů. Nebyl však již schopný určit, zda se v opravovaném diagramu nachází přebytečná část (entita, relace, atribut,...), nebo naopak, zda v něm část chybí. Za tímto účelem byl navržen model, zobrazený na diagramu 4.1, který ukládá informace o jednotlivých porovnávaných párech.

Hlavní částí navržené modelu je generická abstraktní třída *ComparisonResult*, třída obsahuje dva atributy *toEvaluate* pro uložení části odpovědi a *master* pro uložení jejího protějšku ze vzorového diagramu. Třída dále obsahuje tři abstraktní metody *generateCorrectedModel()*, pro vygenerování opravené části diagramu, *isCorrect()*, která vrací hodnotu *true*, pokud se porovnávaná dvojice shoduje, a *isWrong()*, která vrací hodnotu *true* pokud se dvojice zcela neshodují, nebo jeden z páru chybí (v případě přebývajícího, nebo chybějícího protějšku). Poslední metoda *getNotNullModel()* není abstraktní. Její účel je zjednodušení a zpřehlednění generování opraveného diagramu. Metoda provede kontrolu, zda atribut *toEvaluate* neobsahuje referenci na *null*, pokud atribut referenci na *null* neobsahuje, jeho hodnota je funkcí vrácena, v opačném případě se jako výstupní hodnota vrátí atribut *master*. Třída dále implementuje rozhraní *Evaluable*, která obsahuje jedinou metodu *getScore()*. Toto rozhraní je využíváno mapovacím algoritmem pro získání číselného ohodnocení jednotlivých párů na základě jejich podobnosti. I když by bylo možné třídu *ComparisonResult* s rozhraním *Evaluable* sloučit, jsou ponechány odděleně pro kompletní izolování mapovacího algoritmu od porovnávacího modelu. Další třídy modelu jsou navrhnuty a implementovány jako podtřídy *ComparisonResult*. Tyto třídy slouží k ukládání informací o podobnost jednotlivých dvojic, ke generování opravené části modelu. Dále každá ze tříd implementuje metody *isCorrect()*, *isWrong()* a *getScore()*.

Metoda *getScore()* porovnávané dvojci přiřadí celočíselné skóre na základě podobnost. Maximální hodnota, které může dvojice dosáhnout je 0. Za každý nalezený rozdíl je dvojici udělen malus. Hodnoty malusů jsou uloženy jako konstanty ve statické třídě *ScoreValues*.

```
1  override fun getScore(): Int = this.getScoreWithOutChildren() - this
       .childrenComparisons.map { it.getScore() }
2          .reduce { acc, value -> acc + value }
3
4      private fun getScoreWithOutChildren(): Int {
5          var score = ScoreValues.PerfectScore
6
7          if (nameMismatch) {
8              score -= ScoreValues.EntityNameMismatch
9          }
10         score -= attributeComparisons.map { it.getScore() }.reduce {
               acc, value -> acc + value }
11         score -= parentComparison?.getScoreWithOutChildren() ?: 0
12         return score;
13     }
```

Zdrojový kód 4.1: Ukázka metody *getScore()* třídy *EntityComparisonResult*

Porovnávací model obsahuje třídu pro uložení informací o porovnání pro všechny třídy z modelu DSM, které implementují rozhraní *Correctable*. Výjimkou je třída *RelationAttributeComparisonResult*, která toto rozhraní neimplementuje, zároveň je její model natolik jednoduchý na to, aby vyžadovala dedikovanou třídu porovnání. Důvod existence této třídy se nachází ve tříde *RelationEnd*, která obsahuje kolekci *RelationAttribute*. Aby bylo možné přesně kvantifikovat podobnost dvou tříd *RelationEnd*, je nutné na sebe jednotlivé relační atributy namapovatu. K tomu je potřeba zaobalující třída, která implementuje rozhraní *Evaluable*.

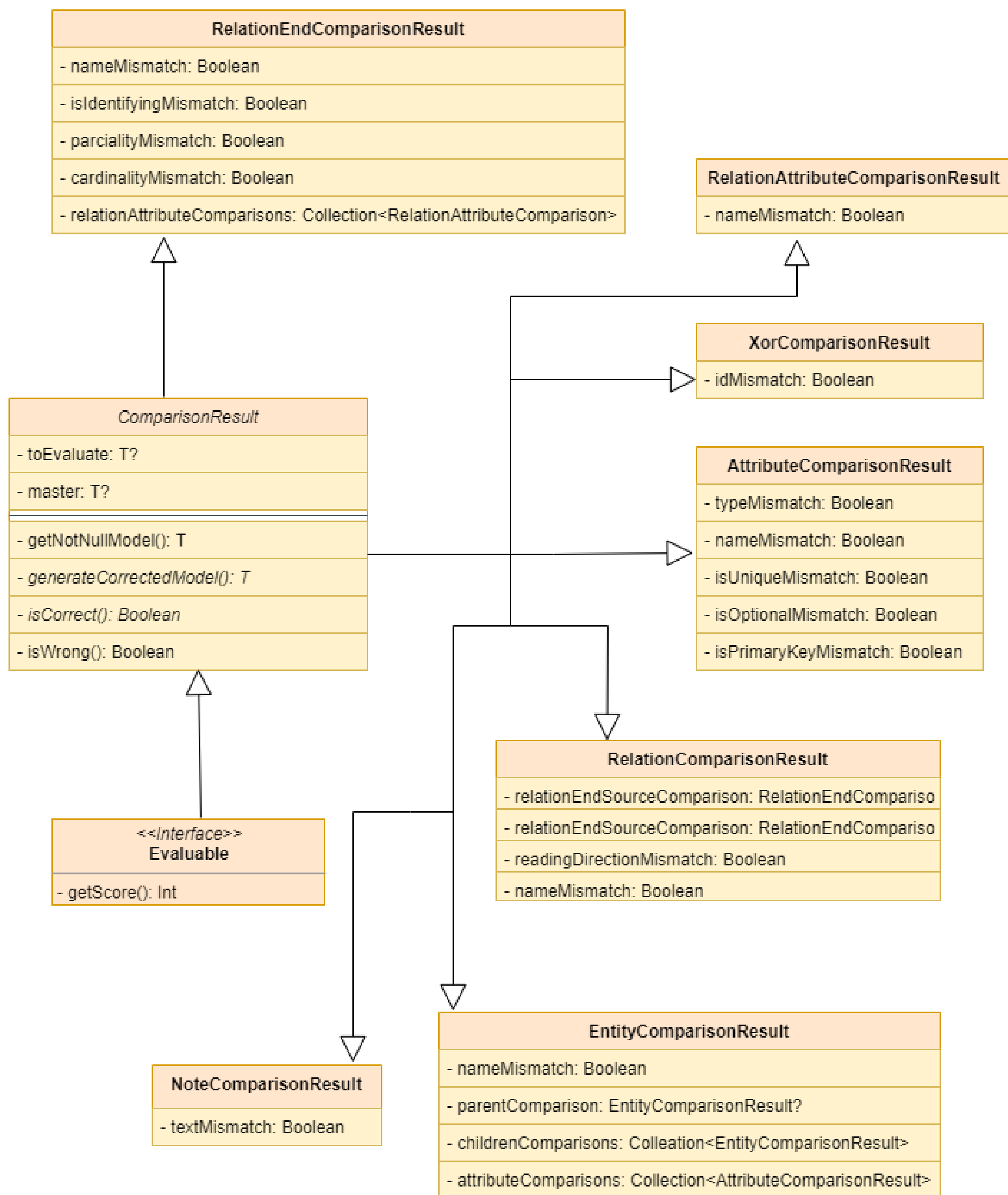
```
1  override fun generateCorrectedModel(): Entity =
2          super.getNotNullModel().also {
3              it.color = determineColor(this)
4              it.attributes = this.attributeComparisons.map { it.
                   generateCorrectedModel() }.toMutableSet()
5              it.relationEnds = this.relationEndsComparisons.map { it.
                   generateCorrectedModel() }.toMutableSet()
6              it.children = this.childrenComparisons.map { child ->
                   child.generateCorrectedModelWithParent(it) }.
                   toMutableSet()
7              val correctionComment = this.generateComment();
8              if (correctionComment != null) {
9                  it.comments.add(correctionComment)
10             }
11         }
```

Zdrojový kód 4.2: Ukázka metody *generateCorrectedModel()*

4.3 Třída AssignmentSolver

Třída *AssignmentSolver* obsahuje pouze jednu generickou statickou metodu *solve()*. Metoda přijímá jeden parametr, kterým je dvourozměrný seznam tříd typu T, reprezentující tabulku hodnot podobně jako u příkladu 3.1. Typ T je dále omezen pouze na třídy implementující rozhraní *Evaluable*. Výstupem této metody je seznam instancí třídy T, které tvoří optimální řešení přiřazovacího problému. V případě, že dimenze vloženého parametru neodpovídají čtvercové matici, skončí metoda výjimkou *InvalidDimensionsException*.

Na ukázce metody 3.1, je na první pohled patrné, že počet kroků algoritmu se liší v implementaci a v návrhu. Nejedná se o chybu. Implementovaný algoritmus optimalizovaný a rozdělen do vícero jednodušších kroků. [**munkresAlgorithm**]



Obrázek 4.1: Diagram tříd - Database model corrector

```
1  fun <T: Evaluable> solve(allPossibleAssignments: List<List<T>>):
       MutableList<T> {
2
3          checkDimensions(allPossibleAssignments)
4          init(allPossibleAssignments.size)
5          val scoreMap = extractAndPreProcessScores(
               allPossibleAssignments)
6          var isDone = false
7          var step = Step.One
8
9          while (!isDone) {
10
11             step = when (step) {
12                 Step.One -> stepOne(scoreMap)
13                 Step.Two -> stepTwo(scoreMap)
14                 Step.Three -> stepThree(scoreMap)
15                 Step.Four -> stepFour(scoreMap)
16                 Step.Five -> stepFive(scoreMap)
17                 Step.Six -> stepSix(scoreMap)
18                 else -> Step.Seven
19             }
20             isDone = step == Step.Seven
21         }
22
23         return generateResult(scoreMap, allPossibleAssignments);
24     }
```

Zdrojový kód 4.3: Ukázka metody *solve()*

Samotný algoritmus předpokládá, že skóre jednotlivých dvojic jsou nezáporná. Jelikož ale metoda *getScore()* vrací pouze hodnoty menší nebo rovné 0, musí před započetím samotného výpočtu proběhnout korekce těchto hodnot. Vzhledem k tomu, že při výpočtu skóre platí: čím nižší hodnota, tím horší shoda a při samotném výpočtu optimálního přiřazení platí přesný opak: čím vyšší hodnota, tím horší shoda, stačí jednotlivé skóre nahradit jeho absolutní hodnotou.

Jedním z možných úskalí algoritmu je generování výsledku. Pokud nastane situace, že správných řešení existuje vícero, neexistuje způsob, kterým by algoritmus našel to nejvhodnější z pohledu porovnávání databázových modelů. Tomuto lze částečně předejít přiřazením unikátních hodnot k jednotlivým malusům ve třídě *ScoreValue*, které ale dále musí splňovat podmínku, že kombinací vícero méně závažných chyb nelze získat stejné skóre, jako jednou závažnější chybou. Uvedeno na příkladu třídy *Attribute*: Pokud nastavíme malus za chybný název na hodnotu 1 a malus za chybně nastavenou unikátnost na 2, žádná jiná chyba již nesmí odpovídat malusu 3. Aktuálně odpovídají nastavené hodnoty mocninám čísla 2. Pro názornější ukázku jsou v tabulce 4.1 uvedeny hodnoty pro porovnávání entitních atributů.

Mapovací algoritmus není použit pouze na vzájemné mapování entit, relací a poznámek, ale také například entitních atributů.

chyba	malus
AttributeNameMismatch	1
AttributeUniqueMismatch	2
AttributeIsOptionalMismatch	4
AttributeTypeMismatch	8
AttributePrimaryKeyMismatch	16

Tabulka 4.1: Příklad nastavení malusů pro rozdíly v atributech

4.4 Třída DBModelCorrector

Jedná se o hlavní třídu celého modulu zastřešující veškerou logiku pro ohodnocení studentova řešení. Třída přijímá dva parametry konstruktoru, jedná se o samotnou odpověď, určenou k ohodnocení a seznam všech přípustných řešení úlohy. Při inicializaci třídy je studentovo řešení porovnáno se všemi vzorovými řešeními. Následně je vybrána dvojce odpověď-vzor, které jsou si nejvíce podobné a na základě výsledku porovnání těchto dvou modelu je úloha ohodnocena a je vygenerován opravený model.

Před samotným porovnáním je vzorový diagram zkontrolován, zda neobsahuje M:N vazbu. Pokud ano, jsou pomocí metody *decomposeAllManyToMany()* třídy *Decomposition* dekomponovány všechny M:N vazby a výsledný diagram je přiřazen do seznamu možných řešení.

Třída obsahuje dva atributy *correctedModel* a *numberOfMistakes*, u kterých je zveřejněná pouze metoda *get()*. Atribut *CorrectedModel* vrací již opravený model pro zobrazení k manuální opravě. *NumberOfMistakes* reprezentuje počet chyb, kterých se student dopustil. Dále třída obsahuje metodu *isCorrect()*, jejíž návratová hodnota je typu *Boolean* a vrací *true*, pokud je opravovaný diagram korektní.

4.5 Pomocné třídy a rozšiřující funkce

4.5.1 Třída Decomposition

Třída *Decomposition* obsahuje pouze jednu statickou metodu *auxDecomposeMany()*, metoda přijíma parametr typu *Canvas* a vrací *List* instancí třídy *Canvas*, které obsahují všechny způsoby dekompozice M:N vazeb v diagramu.

4.5.2 Třída ScoreValues

Statická třída *ScoreValue* obsahuje hodnoty malusů pro výpočet skóre porovnání jako konstant.

4.5.3 Rozšiřující funkce

Většina funkcionalit rozšiřující model DSM, nebo porovnávací model (vyjma metod z *ComparisonResult* a *Evaluable*) je implementována formou rozšiřujících funkcí.

4.5.3.1 Funkce *compareAll()*

Generická funkce *compareAll()* rozšiřující rozhraní *MutableList<T>* přijímá dva generické parametry T, E a dva vstupní parametry, *master* typu *MutableList<T>* a *comparisonClassType* typu *JsClass<E>*. Výstupem této metody je *List<E>*. Účelem této metody je porovnání dvou kolekcí, obsahující instance tříd stejného typu, a s použitím algoritmu mapování vytvořit seznam nejpodobnějších dvojic. Jelikož je funkce generická, je nutné ji předat typ třídy reprezentující porovnání prvků, pro zpřístupnění konstruktoru dané třídy.

4.5.3.2 Funkce *createInstance()*

Tuto funkci je možné považovat za nerizikovější část celého modulu. Jelikož se Kotlinovský kód překládá do JavaScriptu a protože samotný projekt Kotlinu překládaného do JavaScritpu od autorů jazyka je stále v rané fázi vývoje, není možné přistupovat ke všem funkcím, které nabízí API reflexe například v Javě. Aktuálně je možné tento problém obejít. A je velice doporučeno tuto funkci nahradit, jakmile to samotný jazyk dovolí.

Samotná funkce využívá faktu, že jména proměnných uvnitř těla funkce se při překlad do JavaScriptu nezmění, proto je možné přistupovat k těmto proměnným i uvnitř bloku funkce *js()*, která obsahuje kód v čistém JavaScriptu.[9]

```
1 fun <T: Any> JsClass<T>.createInstance(vararg args: dynamic): T {
2     val argsArray = (listOf(null) + args).toTypedArray()
3     val ctor = this
4     return js("new (Function.prototype.bind.apply(ctor, argsArray))"
         ) as T
5 }
```

Zdrojový kód 4.4: Ukázka metody *solve()*

Kapitola 5

Testování

Jelikož není samotný projekt DSM připravený na nasazení na portál DBS, nebylo možné provést uživatelské testy automatické opravy, proto bylo nutné spoléhat se pouze na vytvořené automatické testy. Hlavními cíli testování byly třídy *AssignmentSolver*, *Decomposition* a *DBModelCorrector*. Návrh testovacích scénářů a samotné testování probíhalo až na samém konci vývoje. Testovací scénáře byly navrženy tak, aby pokryly co největší množinu všech situací, které mohou nastat. Testování probíhalo heuristicky, tj. všechny scénáře byly nejdříve vyřešeny ručně a výstupy se porovnávaly s výsledky z ručního řešení. Testy byly implementovány jako automatické unit testy, které se spouštějí v prostředí NodeJS při kompilaci projektu. [10] Testování odhalilo několik chyb týkajících se návrhu doménového modelu automatické opravy a algoritmu dekompozice.

Závěr

Tato práce se zabývá především návrhem a implementací automatické opravy konceptuálních modelů pro projekt DSM. Tento projekt si klade za cíl nahrazení aktuální aplikace Kreslítko, která již nevyhovuje rostoucím požadavkům předmětu BI-DBS.

V práci je popsána aktuální metoda automatické opravy konceptuálních modelů na portále dbs.fit.cvut.cz. Dále je v práci popsán projekt DSM a technologie na kterých je vyvíjen. Následuje popis navrženého a uvedení pro problémů optimální přiřazování částí porovnávaných diagramů do dvojic studenvzor. O přiřazovacím problému je dále vedena diskuze popisující několik možných řešení a jejich nedostatky. Maďarský algoritmus, který se z možných řešení jeví jako vyhovující pro mapování dvojic, je dále představen na konkrétním příkladu. Následuje popis samotné implementace maďarského algoritmu, dekompozice vztahů M:N a hlavní třídy zastřešující logiku automatické opravy diagramů *DBModelCorrector*. K popisu implementace jsou také připojené další diskuze popisující úskalí a potenciální slabá místa implementovaného řešení. Poslední část práce obsahuje popis testování modulu.

Hlavním cílem práce bylo dodat funkční řešení automatické opravy konceptuálních modelů, které je postaveno na technologiích projektu DSM, a které je schopné pracovat s novým robustnějším modelem diagramu. Nový modul, který je aktuálně připravený na porovnávání konceptuálních modelů, by měl být do budoucna propojen s aplikací Tralex, která umožňuje tvorbu zjednodušených relačních modelů. Propojením těchto dvou modulů umožní vyučujícím modelovat řešení úloh, zaměřených na transformaci konceptuálního modelu do relačního, pouze jako konceptuálního modelu. Všechna přípustná řešení úlohy se vygenerují automaticky. Aby ale bylo toto propojení možné, je nutné navrhnout a implementovat nový modul pro automatickou transformaci konceptuálního modelu na zjednodušený relační.

Bibliografie

1. *Why Kotlin?* Rusko: JetBrains, 2020. Dostupné také z: https://kotlinlang.org/%5C#why-kotlin.
2. *Kotlin Nullability.* Rusko: JetBrains, 2020. Dostupné také z: https://kotlinlang.org/docs/reference/null-safety.html.
3. *Kotlin Extensions.* Rusko: JetBrains, 2020. Dostupné také z: https://kotlinlang.org/docs/reference/extensions.html.
4. *Doménový model DSM.* Praha, 2020. Dostupné také z: https://gitlab.fit.cvut.cz/dbs/dsm/wikis/Domain-Model.
5. SUCHÝ ONDŘEJ, VALLA Tomáš. *Párování, hledání párování v bipartitních grafech, Hallova věta a její důsledky* [online] [cit. 2020-09-17]. Dostupné z: https://courses.fit.cvut.cz/BI-AG2/media/lectures/bi-ag2-p05.pdf.
6. TLAPÁK, Martin. *Přiřazovací problém s aplikací ve zdravotnictví.* Praha, 2018.
7. VALENTA, Michal; POKORNÝ, Jaroslav. *Databázové systémy.* 1. vyd. Praha: ČVUT, 2013. ISBN 987-80-01-05212-9.
8. HLAVATÝ, Martin. *Softwarový proces* [online] [cit. 2020-09-17]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/171216/course/section/28533/1_SoftwareProcess.pdf.
9. *Kotlin and JavaScript Interoperability.* Rusko: JetBrains, 2020. Dostupné také z: https://kotlinlang.org/docs/reference/js-interop.html.
10. *Testování* [online] [cit. 2020-09-17]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/308910/course/section/46035/06_Testing.pdf.

Příloha A

Seznam použitých zkratek

DSM Database schema modeller

JSON JavaScript Object Notation

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
├── impl ......................... zdrojové kódy implementace
└── thesis ....................... zdrojová forma práce ve formátu LaTeX
thesis.pdf ....................... text práce ve formátu PDF
```