

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5220-72273

Bc. Jakub Ondik

INTERAKTÍVNA VIZUALIZÁCIA ZMIEN
V SOFTVÉROVÝCH MODELOCH

Diplomová práca

Študijný program: Softvérové inžinierstvo

Študijný odbor: 9.2.5 Softvérové inžinierstvo

Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového
inžinierstva, FIIT STU v Bratislave

Vedúci práce: Ing. Karol Rástočný, PhD.

máj 2018

Zadanie diplomovej práce

Meno študenta: **Bc. Jakub Ondik**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Interaktívna vizualizácia zmien v softvérových modeloch**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:

Všeobecný cieľ:

Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.

Špecifický cieľ:

Vytvorte riešenie zodpovedajúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.

Riadte sa pokynmi Vášho vedúceho.

Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti, alebo o priebežné výsledky Vášho riešenia, alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnete zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU Bratislava

Vedúci práce: **Ing. Karol Rástočný, PhD.**

Termíny odovzdania:

Podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu

Predmety odovzdania:

V každom predmete dokument podľa pokynov na www.fiit.stuba.sk v časti: home > Informácie o > štúdiu > organizácia štúdia > diplomový projekt.

V Bratislave dňa 13. 2. 2017

SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE

Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

1

prof. Ing. Pavol Návrat, PhD.
riaditeľ Ústavu informatiky, informačných systémov
a softvérového inžinierstva

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce ¹

Študent:

Meno, priezvisko, tituly: Jakub Ondik, Bc.
Študijný program: Softvérové inžinierstvo
Kontakt: jakubondik@gmail.com

Výskumník:

Meno, priezvisko, tituly: Karol Rástočný, Ing. PhD.

Projekt:

Názov: Interaktívna vizualizácia zmien v softvérových modeloch
Názov v angličtine: Interactive visualization of modifications in software models
Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky: modelovanie softvéru, vizualizácia softvéru

Text návrhu zadania²

Počas vývoja softvérových systémov, vo fáze návrhu, sa vytvárajú rôzne softvérové modely. Tieto modely sa následne upravujú podľa meniacich sa požiadaviek na systém napríklad zo strany zákazníka, ale aj podľa zmien v návrhových rozhodnutiach týkajúcich sa práve riešených problémov. Tieto zmeny sú niekedy ťažko viditeľné bez predchádzajúceho poznania pôvodných verzií modelov, čo môže spôsobiť zníženie efektivity práce softvérových analytikov. Riešením týchto problémov je detekcia a interaktívna vizualizácia vykonaných zmien v softvérových modeloch.

Analyzujte problematiku detekcie a vizualizácie zmien v softvérových modeloch. Navrhnite metódu detekcie a vizualizácie zmien v softvérových modeloch v podobe interaktívneho softvérového prototypu, ktorý bude riešiť problém detekcie a vizualizácie zmien medzi verziami softvérových modelov. Metódu overte prostredníctvom implementovaného prototypu na množine dát pozostávajúcej z viacerých softvérových modelov zapísaných v jazyku UML spolu s ich históriou.

¹ Vytlačiť obojstranne na jeden list papiera

² 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Literatúra³

- KESSENTINI, Marouane, et al. Search-based detection of model level changes. Empirical Software Engineering, 2016, 1-46.
- SAISIM, Pattamaporn; SENIVONGSE, Twittie. Verifying software change requirements through conceptual class diagrams comparison. In: Proceedings of the World Congress on Engineering and Computer Science. 2014.

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Jakub Ondik, konzultoval(a) a osvojil(a) si ho Ing. Karol Rástočný, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 23.1.2017



Podpis študenta



Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / nie⁴

Dňa: 13. 2. 2017



Podpis garanta predmetov

³ 2 vedecké zdroje, každý v samostatnej rubrike a s údajmi zodpovedajúcimi bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uvedte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

⁴ Nehodí sa prečiarknite

ČESTNÉ PREHLÁSENIE

Čestne prehlasujem, že som predloženú diplomovú prácu vypracoval samostatne, na základe konzultácií a s použitím uvedenej literatúry a ďalších informačných zdrojov.

V Bratislave, 3. 5. 2018

Bc. Jakub Ondik

POĎAKOVANIE

Chcem sa poďakovať môjmu vedúcemu práce, Ing. Karolovi Rástočnému, PhD., za cenné a užitočné rady a za dlhé diskusie nielen o realizácii tejto práce. Tak-tiež ďakujem členom výskumnej skupiny AdvanSD za cenné rady, ktoré prispeli k výsledku tejto práce.

Anotácia

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLOGIÍ

Študijný program: Softvérové inžinierstvo

Autor: Bc. Jakub Ondík

Diplomová práca: Interaktívna vizualizácia zmien v softvérových modeloch

Vedúci diplomovej práce: Ing. Karol Rástočný, PhD.

máj 2018

Počas vývoja softvérových systémov, vo fáze návrhu, sa vytvárajú rôzne softvérové modely. Tieto modely sa následne upravujú podľa meniacich sa požiadaviek na systém napríklad zo strany zákazníka, ale aj podľa zmien v návrhových rozhodnutiach týkajúcich sa práve riešených problémov. Tieto zmeny sú niekedy ťažko viditeľné bez predchádzajúceho poznania pôvodných verzií modelov, čo môže spôsobiť zníženie efektivity práce nielen pre softvérových analytikov, ale aj pre programátorov, ktorí tieto zmeny premietajú do zdrojového kódu. Riešením týchto problémov je detekcia a interaktívna vizualizácia vykonaných zmien v softvérových modeloch.

Navrhujeme metódu detekcie zmien medzi verziami UML softvérových modelov, ktorá slúži ako základ pre vizualizáciu týchto zmien. Samotnú metódu vizualizácie zmien navrhujeme vzhľadom na podporu skupinovej diskusie.

Overenie metódy detekcie zmien sme vykonali nad poskytnutými modelmi a nad vlastnými modelmi, pričom sme sledovali granularitu a správnosť detekcie zmien. Na overenie sme použili najčastejšie používané typy diagramov, a to diagramy aktivít a tried, stavové diagramy a sekvenčné diagramy. V prípade sekvenčných diagramov sme zaznamenali nedostatok v detekcii presunov jednotlivých správ, ktorý vychádza z povahy reprezentácie správ v XMI formáte.

Vizualizáciu zmien sme overili prostredníctvom používateľského testovania, počas ktorého sme porovnávali potrebný čas na identifikáciu zmien a počet neidentifikovaných zmien. Po porovnaní výsledkov našej metódy s výsledkami existujúceho prístupu môžeme usúdiť, že naša metóda má významne lepšie výsledky.

Annotation

Slovak University of Technology in Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree course: Software engineering

Author: Bc. Jakub Ondik

Master's Thesis: Interactive visualization of modifications in software models

Supervisor: Dr. Karol Rástočný

2018, May

During the development of software systems, various software models are created in the software design phase. The models are being further modified not only by customers' changes in system requirements, but by design decisions based on problems being solved as well. These changes are sometimes not visible without previous familiarity with original versions of software models, which can cause loss of efficiency not only for software analysts, but for programmers, who must reflect these modifications into source code as well. The solution to these problems can be found in detection and interactive visualization of modifications in software models.

We propose the method for modifications detection in UML software models, which serves as the basis for their visualization. We propose the visualization method with support of group discussion in mind.

Evaluation of difference detection method was performed on existing provided and our models. We observed granularity and precision of detected differences. We performed the evaluation on the most frequently used types of diagrams, namely activity, state, class and sequence diagrams. In case of sequence diagrams, we observed lack of precise detection of message movements, which arises due to the character of message representation in XMI format.

To evaluate the visualization method, we carried out a user testing, during which we compared needed time to identify differences between two model versions and number of missed differences. After comparing the results of our method with the existing approach we conclude that our method achieved significantly better results.

Obsah

1	Úvod	1
2	Detekcia zmien v softvérových modeloch	3
2.1	Metódy detekcie zmien	3
2.1.1	UMLDiff	3
2.1.2	Detekcia zmien na základe zhôd	5
2.1.3	Detekcia zmien na báze vyhľadávania	6
2.1.4	Detekcia zmien pomocou produkčného systému	8
2.2	Zhrnutie a diskusia	8
3	Vizualizácia zmien v softvérových modeloch	11
3.1	Vizualizácia zmien	11
3.1.1	Fujaba Tool Suite	11
3.1.2	UMLDiff _{cll}	13
3.1.3	Rozšírenie pre nástroj PISET	15
3.1.4	Sparx Enterprise Architect	16
3.1.5	IBM Rational Rhapsody DiffMerge	18
3.2	Upriamanie pozornosti	20
3.2.1	Rybie oko	21
3.2.2	Hĺbka poľa	22
3.2.3	Teplotná mapa	23
3.2.4	Mapa nápadnosti	24
3.3	Zhrnutie a diskusia	25
4	Metóda detekcie zmien v softvérových modeloch	29
4.1	Vytvorenie upraveného Merkleho stromu	29
4.2	Prechádzanie vytvorenej stromovej štruktúry	32
4.3	Spracovanie detegovaných zmien	32
5	Metóda vizualizácie zmien v softvérových modeloch	35
5.1	Statické časti modelov	36
5.2	Dynamické časti modelov	40

5.3	Vizualizácia veľkých softvérových diagramov	40
6	Overenie metódy detekcie zmien v softvérových modeloch	43
6.1	Prototyp	43
6.2	Vyhodnotenie	45
7	Overenie metódy vizualizácie zmien v softvérových modeloch	47
7.1	Prototyp	47
7.2	Vyhodnotenie	49
8	Zhodnotenie a záver	53
	Literatúra	57
	Príloha A: Technická dokumentácia	A-1
	Príloha B: Inštalačná príručka	B-1
	B1: Inštalácia rozšírenia	B-1
	B2: Inštalácia vizualizačného nástroja	B-1
	Príloha C: Používateľská príručka	C-1
	C1: Rozšírenie UmlDiff	C-1
	C2: Vizualizačný nástroj UmlViz	C-3
	Príloha D: Použité modely	D-1
	D1: Diagram 1	D-2
	D2: Diagram 2	D-4
	D3: Diagram 3	D-6
	D4: Diagram 4	D-8
	D5: Diagram 5	D-10
	Príloha E: Plán práce	E-1
	E1: Plán práce po prvej priebežnej správe	E-1
	E2: Zhodnotenie zimného semestra	E-1
	E3: Plán práce po druhej priebežnej správe	E-2
	E4: Zhodnotenie letného semestra	E-2

Príloha F: Obsah elektronického média	F-1
Príloha G: Príspevok 1 prezentovaný na IIT.SRC	G-1
Príloha H: Príspevok 2 prezentovaný na IIT.SRC	H-1

1 Úvod

Počas vývoja softvérových systémov sú vo fáze návrhu vytvárané rôzne softvérové modely. Tieto modely sa ďalej menia podľa meniacich sa požiadaviek na systém, ale aj podľa zmien v návrhových rozhodnutiach týkajúcich sa riešených problémov. Tieto zmeny sú niekedy ťažko viditeľné bez predchádzajúceho poznania pôvodnej verzie modelu, čo môže spôsobiť zníženie efektivity práce analytika, ktorý tento model schvaľuje a navrhuje ďalšie úpravy, ale aj práce programátora, ktorý má tieto zmeny pre premietnuť do kódu. Je preto vhodné používať nástroj, ktorý na tieto zmeny automaticky a viditeľne upozorní.

V tejto práci sa sústredíme na vizualizáciu zmien v UML modeloch, pričom spôsob vizualizácie prispôbujeme tímom analytikov. Cieľom tejto práce je poskytnúť metódu určenú na detekciu zmien medzi dvomi verziami modelu vytvoreného pomocou UML. Výstup tejto metódy má slúžiť ako podklad pre navrhovanú metódu vizualizácie zmien v UML modeloch — má teda poskytovať dáta o zmenách. Metódu vizualizácie zmien navrhujeme samostatne pre statické a dynamické typy UML diagramov, keďže vplyv a význam zmien je medzi týmito typmi odlišný.

Overenie metódy na detekciu zmien v UML modeloch plánujeme prostredníctvom porovnávania výstupov existujúcich implementovaných metód s výstupmi našej metódy na detekciu zmien, ktorá bude implementovaná v podobe prototypu. Overenie metódy na vizualizáciu zmien plánujeme prostredníctvom používateľského testovania, čím chceme dokázať jednu z určených hypotéz. Prototyp metódy na vizualizáciu zmien plánujeme implementovať ako externý nástroj, ktorý bude integrovaný s nástrojom Sparx Systems Enterprise Architect v podobe rozšírenia.

V kapitole 2 analyzujeme existujúce metódy na detekciu zmien v softvérových modeloch, pričom sa neobmedzujeme iba na metódy určené pre UML modely. Uvádzame aj krátke vyjadrenie k nedostatkom a použiteľnosti analyzovaných metód.

V kapitole 3 analyzujeme metódy a nástroje podporujúce vizualizáciu zmien v softvérových modeloch. Prevažná časť analyzovaných metód je vhodná pre UML modely, avšak to nevylučuje použitie aj pre iné typy softvérových

modelov. Taktiež uvádzame metódy a techniky určené na upriamanie pozornosti na určité časti vizualizovanej plochy. Analyzujeme aj metódy určené na poskytnutie prehľadu, ktoré plánujeme použiť pri poskytnutí prehľadu vykonaných zmien v rozsahovo veľkých softvérových modeloch. K týmto analyzovaným nástrojom, metódam a technikám sa vyjadrujeme, pričom uvádzame ich nedostatky a postrehy, ktoré by bolo vhodné použiť aj pri návrhu našej metódy.

V kapitole 4 predstavujeme návrh metódy na detekciu zmien v UML modeloch. Predstavujeme nami upravený algoritmus Merkleho stromov vrátane reprezentácie uzla tohto stromu a odôvodnením našej úpravy zameranej na softvérové modely. Taktiež uvádzame pseudokód algoritmu metódy detekcie zmien a predstavujeme jednotný prenosný objekt reprezentujúci zmeny v modeloch, ktorý je určený na prenos zmien na ďalšie spracovanie, v našom prípade na vizualizáciu daných zmien.

V kapitole 5 uvádzame návrh metódy na vizualizáciu zmien v UML modeloch. Uvádzame spôsob vizualizácie zmien v diagramoch patriacich do kategórie štruktúrálnych diagramov. Taktiež uvádzame potrebu rôzneho spôsobu vizualizácie pre behaviorálne (dynamické) diagramy UML. Spomíname aj možnosť pohľadu poskytujúceho prehľad zmien v rozsiahlych modeloch použitím teplotnej mapy.

V kapitole 6 uvádzame vykonané overenie metódy na detekciu zmien v UML modeloch. Uvádzame úspešnosť našej metódy na skupine diagramov a taktiež uvádzame problém s detekciou presunov správ v sekvenčných diagramoch.

V kapitole 7 uvádzame vykonané overenie metódy na vizualizáciu zmien v UML modeloch. Opisujeme vykonaný experiment vrátane porovnania dosiahnutých výsledkov našej metódy v porovnaní s metódou "vedľa seba".

V kapitole 8 zhodnocujeme stav tejto práce, uvádzame potvrdené hypotézy a vyjadrujeme sa k možnostiam ďalšej práce.

2 Detekcia zmien v softvérových modeloch

Základným prvkom pre vizualizáciu zmien v softvérových modeloch sú informácie o zmenách. Na získanie týchto informácií je nutné použiť nástroj alebo metódu, ktorá podporuje daný typ softvérového modelu, resp. jeho verzií. Takáto metóda alebo nástroj môžu zmeny zisťovať či už iteratívne, t. j. v čase ich vzniku zaznamenávaním akcií (pridanie, odstránenie, presun a pod.) tvorcu modelu alebo až po vytvorení modelu na základe porovnávania jeho verzií. V tejto kapitole uvádzame niekoľko takýchto metód.

2.1 Metódy detekcie zmien

2.1.1 UMLDiff

Jednou z metód, ktorá sa zameriava na modely tried je metóda s názvom UMLDiff (Xing a Stroulia, 2005). Táto metóda sa zameriava na modely tried, ktoré sú vygenerované zo zdrojového kódu jazyku Java, pričom deteguje štrukturálne zmeny medzi dvoma verziami takýchto modelov.

Metamodel, s ktorým táto metóda pracuje, sa zakladá na UML modeli statickej štruktúry. Tento metamodel je reprezentovaný ako graf $G_D(V_D, E_D)$, pričom v prípade tejto metódy množina vrcholov V_D pozostáva z entít typov podporovaných jazykom Java, akými sú napríklad balík, trieda a primitívny typ. Každá z týchto entít obsahuje informácie o svojom type, názve, modifikátore viditeľnosti a prístupu a pozícií v zdrojovom kóde. Taktiež obsahuje informáciu, či patrí priamo modelu, ktorý sa bude porovnávať alebo knižnici a informáciu o verzii systému. Množina hrán E_D pozostáva zo štandardných typov vzťahov jazyka UML, akými sú generalizácia, realizácia a pod.

Pri samotnom porovnávaní verzií modelov UMLDiff paralelne prechádza obe verzie modelov pričom postupuje od koreňového uzla, ktorý predstavuje celý systém, postupne cez balíky, triedy a rozhrania, polia a bloky. Identifikácia rovnakých, resp. podobných entít prebieha na základe porovnávania ich názvov s využitím heuristiky podobnosti názvu a porovnávania vzťahov k ostatným entitám pomocou heuristiky štrukturálnej podobnosti.

Heuristika podobností názvu, ktorú táto metóda používa, je založená na porovnávaní párov okolitých znakov v reťazci a prideluje skóre na základe prieniku párov okolitých znakov podľa vzorca 2.1

$$\frac{velkostMnozinyPrieniku * 2}{velkostMnozinyZjednotenia} \quad (2.1)$$

Tento prístup je použitý z dôvodu nepresnosti bežných porovnávacích LCS (angl. *Longest Common Subsequence*, najdlhšia spoločná podpostupnosť) algoritmov – porovnávané reťazce, ktoré sa líšia iba v poradí podreťazcov sú hodnotené nižšie (sú menej podobné) ako reťazce, ktoré majú nahradený jeden podreťazec. Príkladom môže byť metóda s názvom *PridajNovýUzol()*, ktorú LCS algoritmus vyhodnotí ako menej podobnú metóde s názvom *NovýUzolPridaj()* ako metódu *VykresliNovýUzol()*. Práve toto predstavuje veľký nedostatok z dôvodu, že pri modeloch tried je bežné presúvať či už podreťazce v názve metód alebo poradie parametrov metód.

Heuristika štrukturálnej podobnosti funguje na základe prieniku množín entít, v ktorom sú porovnávané entity vo vzťahu, ktorý je špecifikovaný konkrétnym typom. Prienik je pritom určený špecifikovanou porovnávacou metódou, ktorou môže byť aj metóda heuristiky podobnosti názvu. Výsledné skóre je dané vzorcom 2.2

$$\frac{minRovnakych}{maxRovnakych + pocetRozdielnych1 + pocetRozdielnych2}, \quad (2.2)$$

kde *minRovnakych* predstavuje minimum z počtu entít, ktoré sú vo vzťahu s porovnávanými entitami a boli detegované ako rovnaké a *maxRovnakych* predstavuje maximum z týchto hodnôt. Podobne hodnota *pocetRozdielnych1* predstavuje počet entít, ktoré sú vo vzťahu s prvou porovnávanou entitou a *pocetRozdielnych2* predstavuje počet entít, ktoré sú vo vzťahu s druhou porovnávanou entitou. V prípade, že sú tieto množiny prázdne, je použitá heuristika podobností názvu, ktorá je postupne umocňovaná podľa typu porovnávaných reťazcov – *podobnostNazvu*¹ pre typy parametrov, *podobnostNazvu*² pre prístup k poliam a pod.

Výsledné skóre, na základe ktorého táto metóda vyhodnotí, či sú dve entity

podobné, je dané vzorcom 2.3

$$\frac{\textit{podobnostNazvu} + \textit{strukturalnaPodobnost}}{\textit{podobnostNazvu} + N}, \quad (2.3)$$

kde *podobnostNazvu* je hodnota heuristiky podobnosti názvu, *strukturalnaPodobnost* je súčet hodnôt skóre pre všetky typy vzťahov medzi porovnávanými entitami a *N* je počet všetkých možných vzťahov, ktoré môžu medzi dvoma entitami existovať. Táto hodnota je následne porovnávaná s úrovňou podobnosti, ktorá je definovaná používateľom – ak je daná hodnota vyššia ako definovaná úroveň podobnosti, dané entity sú považované za rovnaké.

2.1.2 Detekcia zmien na základe zhôd

V prípade porovnávania viacerých verzií modelov nie je k dispozícii zoznam iteratívnych zmien (angl. *changelist*), napr. ak sú takéto verzie vytvárané spôsobom duplikácie a následného menenia. Tento prípad rieši metóda detekcie zmien na základe zhôd (angl. *correspondences*) (Gerth a Engels, 2008).

Táto metóda sa zmeriava na procesné modely, pričom zmeny deteguje v troch krokoch, a to (i) detekcia zmien v dvoch verziách procesného modelu na základe zhôd a SESE fragmentov (angl. *Single-Entry-Single-Exit*, jediný vstup a jediný východ), pričom pre každý detegovaný rozdiel je vygenerovaná súvisiaca operácia, (ii) zoradenie vygenerovaných operácií podľa štruktúry procesného modelu a (iii) vyriešenie rozdielov medzi verziami procesného modelu podľa priority danej používateľom.

SESE fragmenty predstavujú podgrafy grafu, v tomto prípade procesného modelu, ktoré obsahujú práve jeden vstupný a práve jeden výstupný bod (Vanhatalo et al., 2007). Tieto fragmenty sú využiteľné na detekciu rozdielov medzi dvoma verziami modelu na základe porovnávania podgrafov (Gerth a Engels, 2008).

Zhody predstavujú spôsob prepojenia prvkov nachádzajúcich sa v oboch verziách modelov, pričom za takéto prvky sa považujú prvky obsahujúce rovnakú funkcionality. Táto metóda pracuje v tomto prípade s početnosťou 1 k 1. V prípade, že niektorý z prvkov sa nenachádza v jednej z verzií porovnávaných modelov, je použitá početnosť 1 k 0, resp. 0 k 1 v závislosti od referenčného

modelu. To znamená, že v prípade počtosti 1 k 1 zmena detegovaná nie je a v prípade počtosti 1 k 0, resp. 0 k 1 musí byť prvok z modelu odstránený, resp. do modelu pridaný.

Táto metóda pracuje s tromi typmi zmien pre prvky typu *akcia*, resp. operácií, a to: (i) vloženie $InsertAction(V, x, a, b)$, ktoré predstavuje vloženie prvku x do modelu V medzi prvky a a b a následné prepojenie týchto prvkov s vloženým prvkom x , (ii) odstránenie $DeleteAction(V, x)$, ktoré predstavuje odstránenie prvku x z modelu V a obnovenie riadiaceho toku a (iii) presun $MoveAction(V, x, a, b)$, ktoré predstavuje presun prvku x do modelu V medzi prvky a a b a následné prepojenie týchto prvkov s presunutým prvkom x . V prípade zmien riadiacich uzlov a prepojení, metóda pracuje s analogickými názvami operácií (napr. $DeleteFragment(V, f_1)$, ktorý predstavuje odstránenie fragmentu (prepojenia) f_1 z modelu V), pričom používa aj dodatočné typy na konverziu fragmentu ($ConvertFragment(V, f_1, f_2)$) a konkrétne podtypy operácie $InsertFragment(V, a, b, f_2)$ napr. na vloženie paralelných a cyklických fragmentov.

Samotné zoradenie týchto operácií prebieha na základe dekompozície modelu do SESE fragmentov. Toto umožňuje prepojenie každej operácie do konkrétnej časti (podgrafu) modelu, v ktorom sa nachádza daná operácia nachádza. Týmto procesom sa vytvára hierarchický zoznam zmien, ktorý je možné využiť aj na následnú vizualizáciu častí modelu, v ktorých boli detegované zmeny. Toto umožňuje používateľovi (či už analytikovi, ktorý zmeny prehliada a hodnotí, prípadne spája dve verzie modelov, alebo programátorovi, ktorý na základe týchto zmien upravuje kód) sústrediť sa len na zmenené miesta a teda nezaťažovať sa časťami modelu, ktoré sú pre neho v danej chvíli nerelevantné.

2.1.3 Detekcia zmien na báze vyhľadávania

Zmeny medzi verziami modelov je možné detegovať aj na základe vyhľadávania sekvencií refaktorizácií (Kessentini et al., 2016). Táto metóda k takémuto vyhľadávaniu používa viac-cieľovú optimalizáciu (angl. *Multi-Objective Optimization*) využitím genetického algoritmu NSGA-II (Deb et al., 2002).

Vstupom metódy založenej na NSGA-II sú dve verzie modelu a vyčerpávajúci zoznam refaktorizačných operácií. Výstupom je zoznam aplikácií týchto

refaktorizačných operácií, ktoré vedú od pôvodnej verzie modelu na upravenú verziu modelu. V princípe teda táto metóda hľadá takú postupnosť aplikácií refaktorizačných operácií, aby bola zabezpečená zhoda verzie upraveného modelu s pôvodnou verziou po aplikácii refaktorizačných operácií aplikovaním čo najmenšieho počtu takýchto operácií.

Keďže táto metóda využíva genetický algoritmus, je nutný výber vhodnej počiatočnej populácie, genetických operátorov a hodnotiacej (fitness) funkcie. Dĺžka vektora jedinca v počiatočnej populácii je zvolená v závislosti od počtu refaktorizácií (teda v intervale, ktorého horná hranica je počet refaktorizácií), ktoré budú použité na detekciu zmien medzi verziami modelov. Konkrétna dĺžka je zvolená buď náhodne alebo používateľom. Takýto spôsob zvolenia vektora jedinca je použitý z toho dôvodu, že vyššia dĺžka vektora jedinca nezabezpečuje presnejšiu detekciu sekvencie operácií – niekedy stačí aplikovať malý počet refaktorizačných operácií.

Ako genetické operátory táto metóda používa selekciu, kríženie a mutáciu. Selekcia využíva stochastické univerzálne vzorkovanie – pravdepodobnosť postupu jedinca do nasledujúcej populácie je priamo úmerná jeho fitness hodnote, pričom do nasledujúcej populácie postúpi vždy polovica jedincov, ktorá krížením vytvorí nových jedincov do veľkosti pôvodnej populácie. Kríženie prebieha zvolením náhodného bodu rezu vektorov jedincov, pričom sa časti vektorov, dané zvoleným rezom, medzi sebou spoja, čím vznikne nový jedinec. Mutácia prebieha tak, že z vektora jedinca sa náhodne vyberie aspoň jeden prvok (gén), ktorý predstavuje refaktorizačnú operáciu, a ten je nahradený inou refaktorizačnou operáciou. Po aplikovaní genetických operátorov kríženia a mutácie, metóda ešte odstráni duplikované refaktorizačné operácie nachádzajúce sa vo vektore jedinca.

Na určenie kvality jedincov táto metóda používa dve fitness funkcie – na určenie kvality refaktorizačných operácií, ktorej hodnoty sú z intervalu $[0, 1]$ a na určenie počtu refaktorizačných operácií. Fitness funkcia na určenie refaktorizačných operácií hodnotí jedincov na základe modelu, ktorý vznikne po aplikácii refaktorizačných operácií z vektora daného jedinca na pôvodnú verziu modelu. Táto funkcia teda hodnotí podobnosť výsledného modelu s upraveným modelom. Fitness funkcia na určenie počtu refaktorizačných operácií predstavuje iba dĺžku vektora jedinca. Zmeny medzi modelmi sú teda určené ako

minimálne sekvencie refaktorizačných operácií, čo v konečnom dôsledku predstavuje zoznam zmien prevedených na pôvodnom modeli, prevedením ktorých vznikla modifikovaná verzia daného modelu.

2.1.4 Detekcia zmien pomocou produkčného systému

Jedným zo spôsobov detekcie zmien v modeloch je využitie detekcie sémantickej ekvivalencie pomocou nástroja Rainbow (Costa et al., 2014), nakoľko zmeny v modeloch nemusia byť iba syntaktické. Tento nástroj, okrem detekcie zmien, podporuje aj detekciu konfliktov medzi verziami modelu. Detekciu konfliktov podrobne neuvádzame, keďže táto práca sa konfliktami medzi softvérovými modelmi nezaoberá. Nástroj Rainbow podporuje iba diagramy tried a využíva tri fázy, a to: (i) preklad, (ii) sémantické obohatenie a (iii) detekciu konfliktov.

Fáza *preklad* zabezpečuje konverziu troch verzií modelu (základnej, aktuálnej a vývojovej) na Prolog fakty, pričom každý Prolog fakt predstavuje prvok alebo vzťah z diagramu tried. Príkladom Prolog faktu je *attribute(className, attributeName)*, ktorý vyjadruje fakt, že trieda s názvom *className* obsahuje atribút s názvom *attributeName*.

Vo fáze *sémantického obohatenia* sú na základe Prolog faktov vygenerované nepriame vzťahy. Tieto nepriame vzťahy sú ukladané vo forme ďalších Prolog faktov na základe predgenerovaných pravidiel, ktoré opisujú UML metamodel diagramu tried. Následne sú všetky Prolog fakty porovnávané na ekvivalenciu vo fáze *detekcie konfliktov*, prípadne či sa jedna z verzií sémanticky obsahuje inú verziu. V tejto fáze sa taktiež vykonáva *diff* operácia, čiže operácia detekcie rozdielov. Táto operácia sa vykonáva medzi základnou a vývojovou verziou a medzi základnou a aktuálnou verziou modelu. Každá z týchto operácií zistí množinu pridaní a množinu odstránení. Ďalej nasleduje samotná detekcia konfliktov.

2.2 Zhrnutie a diskusia

V tejto kapitole sme uviedli metódy detekcie zmien v softvérových modeloch. Za najväčší nedostatok považujeme obmedzenie metódy Rainbow na diagramy

tried – odstránenie tohto nedostatku by vyžadovalo veľa úsilia, nakoľko by bolo nutné vytvoriť ďalšie množiny Prolog faktov a pravidiel v závislosti od typov diagramov, ktoré by sme chceli podporovať.

Za ďalší nedostatok považujeme obmedzenie metódy UMLDiff na zdrojový kód jazyku Java. Čiastočné riešenie tohto nedostatku by bolo možné pomocou konverzie zdrojového kódu do iných objektovo orientovaných jazykov. Toto však predstavuje zbytočnú réžiu – vhodnejšie by bola detekcia zmien priamo z modelov, resp. štandardizovaného súborového formátu modelov.

V prípade metódy, ktorá využíva genetický algoritmus NSGA-II, považujeme ako zbytočnú réžiu vytváranie vstupnej množiny možných refaktoriizačných operácií. Toto by však bolo možné vyriešiť automatizovaním procesu vytvárania takejto množiny, prípadne úpravou algoritmu na použitie všeobecných refaktoriizačných operácií, čo by ale malo za následok zvýšenie pamäťovej a časovej zložitosti.

V prípade metódy detekcie zmien na základe zhôd na sa domnievame, že použitie dekompozície modelu (grafu) na podmodely (grafy) by mohlo byť v prípade UML modelov vhodné použitím dekompozície modelu na jednotkové prvky podľa metamodelu UML. Týmto by sme dokázali porovnávať nie len jednotkové prvky (napr. typy atribútov) ale aj kompozitné prvky (napr. trieda) efektívnejšie oproti porovnávaní celého modelu.

3 Vizualizácia zmien v softvérových modeloch

3.1 Vizualizácia zmien

Prehľadná vizualizácia zmien vo vyvíjajúcich sa softvérových modeloch poskytuje rýchlu identifikáciu meniacich sa častí modelov. Práve preto je nesmierne dôležitá ako pre softvérových analytikov, tak aj pre vývojárov, ktorí zmeny v softvérových modeloch premietajú do kódu.

Softvéroví analytici a vývojári používajú rôzne nástroje určené na vizualizáciu zmien v softvérových modeloch, či už ide o samostatné nástroje alebo rozširujúce moduly samostatných nástrojov, ktoré sa medzi sebou líšia nie len spôsobom vizualizácie, ale aj podporou jednotlivých prvkov, ktorých zmeny dokážu vizualizovať.

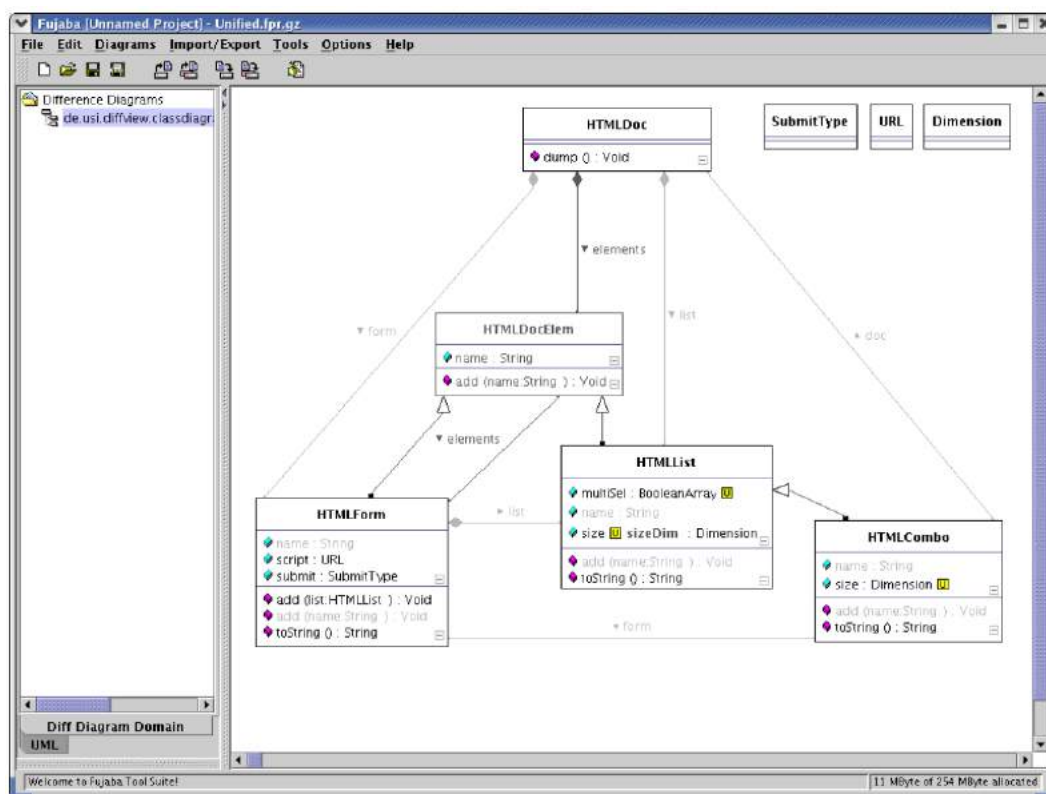
V tejto podkapitole analyzujeme niekoľko riešení zameraných na modelovanie softvéru podporujúcich vizualizáciu zmien, prípadne aj samotnú detekciu zmien medzi verziami modelov a opisujeme niekoľko metód vhodných na upriamanie pozornosti na určité miesta vizualizácie.

3.1.1 Fujaba Tool Suite

Fujaba Tool Suite (Burmester et al., 2004) predstavuje rozšíriteľný CASE nástroj podporujúci modelovanie v jazyku UML. Fujaba štandardne nepodporuje vizualizáciu zmien medzi modelmi – táto funkcionálna vrátane detekcie zmien (rozdielov) je dodaná prostredníctvom samostatných rozšírení (Niere, 2004). Tieto rozšírenia používajú rozšírený metamodel nástroja Fujaba Tool Suite, ktorý je založený na metamodeli jazyka UML a podporujú exportovanie a importovanie modelov vo formáte XMI špecifického pre nástroj Fujaba Tool Suite a Poseidon¹.

Vizualizácia zmien prebieha nad UML diagramom načítanom z XMI súboru, ktorý obsahuje iba potrebné informácie pre vizualizáciu zmien, t. j. samotné zmeny. Tento XMI súbor je vygenerovaný algoritmom detekcie zmien, ktorý porovnáva dve samostatné verzie softvérového modelu. Tieto zmeny (roz-

¹<http://www.gentleware.com/>



Obrázok 1: Vizualizácia zmien pomocou rozšírenia nástroja Fujaba (Niere, 2004).

diely) sa následne aplikujú na obe verzie modelu, pričom sa používa tzv. ľavý model, ktorý predstavuje jednu z porovnávaných verzií a pravý model, ktorý predstavuje druhú, resp. aktuálnu verziu modelu. Samotná vizualizácia prebieha priamo v nástroji, pričom obe verzie sú vizualizované na spoločnom modeli.

Na obrázku 1 môžeme vidieť ukážku vizualizácie zmien (rozdielov) detegovaných medzi dvoma verziami diagramu tried. Prvky, ktoré sa nachádzajú iba v ľavom modeli sú zvýraznené červenou (na obrázku svetlošedou) farbou. Naopak prvky, ktoré sa nachádzajú iba v pravom modeli, sú zvýraznené zelenou (na obrázku tmavošedou) farbou. Príkladom atribútu, ktorý sa nachádza iba v ľavom modeli je atribút *name* triedy *HTMList*. Príkladom triedy, ktorá sa nachádza iba v pravom modeli je trieda *HTMLDocElem*.

Zmeny v rámci prvkov (napr. atribúty), sú vizualizované prostredníctvom žltej značky *U*, ktorú môžeme vidieť v triede *HTMList*, pričom zvýraznená

je len konkrétna zmena a nie celý atribút. Táto značka má tri stavy, a to stav, kedy bola časť prvku upravená len v ľavom modeli (vizualizované červenou, resp. svetlošedou farbou), stav, kedy bola časť prvku upravená len v pravom modeli (vizualizované zelenou, resp. tmavošedou farbou) a stav, kedy bola časť prvku upravená v oboch modeloch (sú zobrazené hodnoty z oboch modelov, pričom sú zvýraznené farbou podľa modelu, ktorému patria). Tretí stav môžeme vidieť pri atribúte typu *Dimension* v triede *HTMLList*.

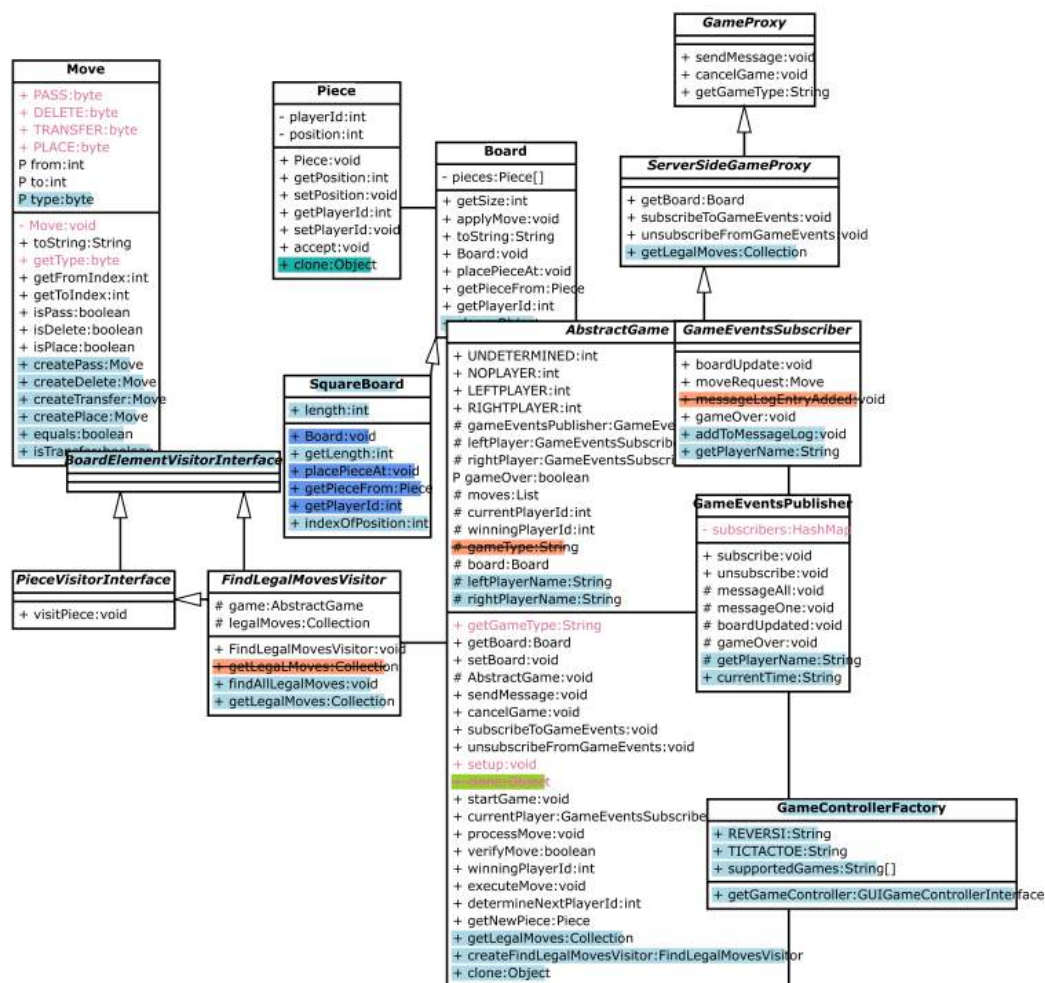
Toto rozšírenie však poskytuje vizualizáciu zmien iba pre jeden typ UML diagramu, a to pre diagram tried, pričom nepodporuje vizualizáciu zmien (aktualizácií) pre vzťahy medzi jednotlivými triedami. Podobne nepodporuje vizualizáciu presunov prvkov (napr. atribútov) medzi triedami.

3.1.2 UMLDiff_{cld}

UMLDiff_{cld} (Girschick a Darmstadt, 2006) predstavuje algoritmus zameraný na detekciu a vizualizáciu zmien v softvérových modeloch. Tento algoritmus neposkytuje grafické používateľské rozhranie, avšak poskytuje generovanie záznamov o detegovaných zmenách vrátane ich vizualizácie priamo v modeli (diagrame) a podporuje iba diagramy tried.

Vizualizácia zmien prebieha zvýraznením celého miesta zmeny, t. j. ak bol upravený len typ atribútu, zvýraznený je celý atribút. Ukážku vizualizácie môžeme vidieť na obrázku 2, pričom algoritmus podporuje osem farieb podľa typu zmeny, a to:

- bielu pre úspešné namapovanie prvkov medzi verziami modelov,
- šedú pre neúspešné namapovanie prvkov medzi verziami modelov (nie je použitá na obrázku),
- svetlomodrú pre pridanie prvku (napr. pridanie triedy *GameControllerFactory*, pričom sú zvýraznené aj atribúty),
- oranžovú, vrátane prečiarknutia prvku, pre odstránenie prvku (atribút *gameType* v triede *AbstractGame*),
- tmavozelenú pre novú pozíciu prvku pri presune (operácia *clone* triedy *Piece*),
- svetlozelenú pre pôvodnú pozíciu prvku pri presune (operácia *clone* triedy *AbstractGame*),



Obrázok 2: Vizualizácia zmien pomocou algoritmu *UMLDiff_{cd}* (Girschick a Darmstadt, 2006).

- tmavomodrú pre operáciu klonovania (duplikovania), čo predstavuje pridanie prvku, ktorý sa podobá na už existujúci prvok (napr. zvýraznenie prekonaných metód, ktoré môžeme vidieť v triede *SquareBoard*) a
- ružovú pre zmenu prvku (atribút *subscribers* triedy *GameEventsPublisher*).

Tento algoritmus nepodporuje exportovanie a importovanie modelov vo formáte XMI, avšak podporuje vlastný formát, ktorý je založený na XML.

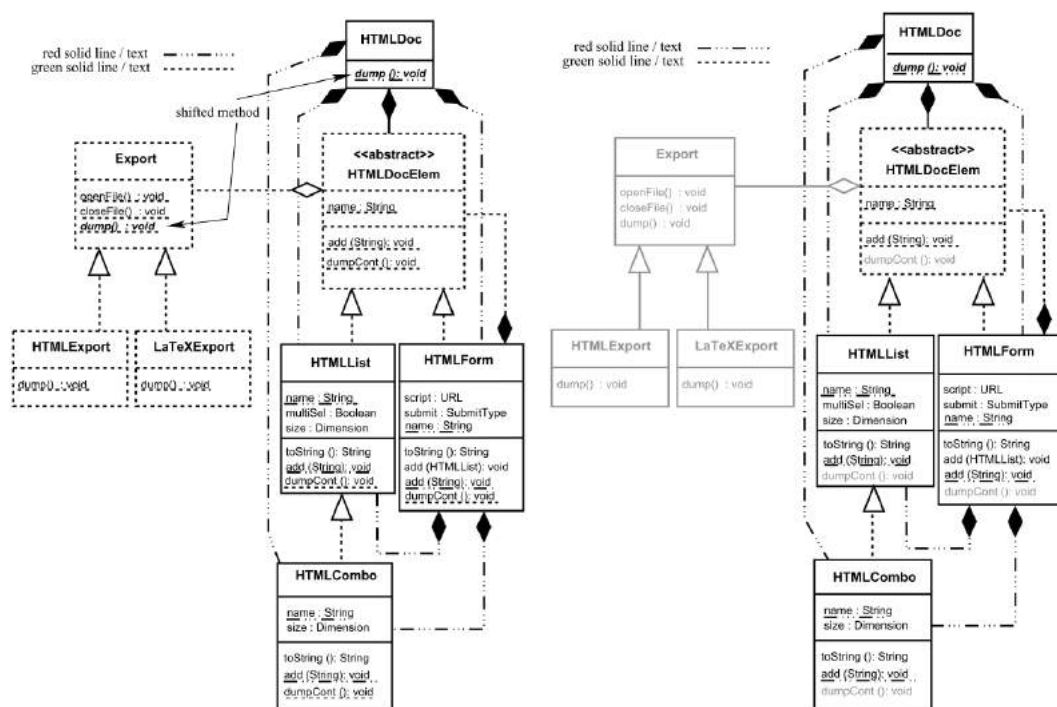
3.1.3 Rozšírenie pre nástroj PISET

Možnosť detegovať a vizualizovať zmeny v distribuovaných prostrediach pre vývoj softvéru (angl. *distributed software development environment*, *distributed SDE*) ponúka rozšírenie pre nástroj PISET (Kelter et al., 1999; Ohst et al., 2003). Toto rozšírenie poskytuje vizualizáciu zmien obmedzeným počtom farieb a možnosť vizualizácie presunu prvkov (napr. atribútov medzi triedami v diagrame tried). Ukážku vizualizácie zmien v diagrame tried môžeme vidieť na obrázku 3.

Vizualizácia spočíva v zvýraznení celých zmenených častí, a to zelenou farbou (bodkovaná čiara) v prípade pridania a červenou farbou (kombinácia bodiek a čiar) v prípade odobrania. Modifikácia existujúcich prvkov je vizualizovaná kombináciou pridania a odstránenia celého prvku, t. j. v prípade zmeny typu atribútu je vizualizované odstránenie a pridanie celého atribútu. Presun prvkov je vizualizovaný obojsmernou šípkou, pričom pôvodná pozícia je zvýraznená červenou farbou (čiara pozostávajúca z kombinácie bodiek a čiar) a nová pozícia je zvýraznená zelenou farbou (bodkovaná čiara). Presun môžeme vidieť na metóde *dump()*, ktorá bola presunutá z triedy *HTMLDoc* do triedy *Export* na obrázku 3a.

Rozšírenie taktiež poskytuje možnosť filtrovania zmien, ktoré môžeme vidieť na obrázku 3b. Tento filter poskytuje zvýraznenie prvkov podľa typu filtra, pričom prvky nevyhovujúce filtru sú vykreslené šedou farbou. Prvky je možné filtrovať podľa (i) typu prvku, v ktorom nastala zmena, napr. budú vizualizované zmeny iba v triedach, (ii) verzii modelu, t. j. budú vizualizované iba zmeny, ktoré nastali vo vybranej verzii modelu a (iii) zahrnutia alebo vylúčenia zmien vyplývajúcich z referencií. Tieto typy filtrov je možné kombinovať – je možné napr. zvýrazniť iba zmeny v triedach nachádzajúcich sa v prvej verzii modelu.

Rozšírenie, resp. nástroj pre ktorý je toto rozšírenie určené, používa na perzistenciu verzií modelov distribuovaný repozitár a neumožňuje akékoľvek exportovanie a importovanie modelu a jeho verzií.



(a) Štandardný pohľad na zmeny.

(b) Pohľad na zmeny s upriamením pozornosti.

Obrázok 3: Vizualizácia zmien v diagrame pomocou rozšírenia pre nástroj PI-SET (Ohst et al., 2003).

3.1.4 Sparx Enterprise Architect

Nástroj Sparx Enterprise Architect² je komplexný CASE nástroj založený na jazyku UML. Enterprise Architect poskytuje, okrem samotného návrhu softvérových systémov, možnosť vytvárania verzií softvérových modelov pomocou snímok⁴ (angl. *snapshots*) a ich porovnávania s aktuálnou verziou, resp. s inou verziou softvérového modelu. Vizualizáciu výsledku porovnávania môžeme vidieť na obrázku 4.

Tabuľkový prehľad zmien medzi verziami softvérových modelov (obrázok 4a) poskytuje štruktúrovaný prehľad zmien. V stromovej štruktúre v ľavej časti tabuľky sú uvedené všetky prvky, v ktorých nastala zmena. Po vybraní konkrétneho prvku sú zobrazene jeho všetky možné vlastnosti – v stĺpci *Model*

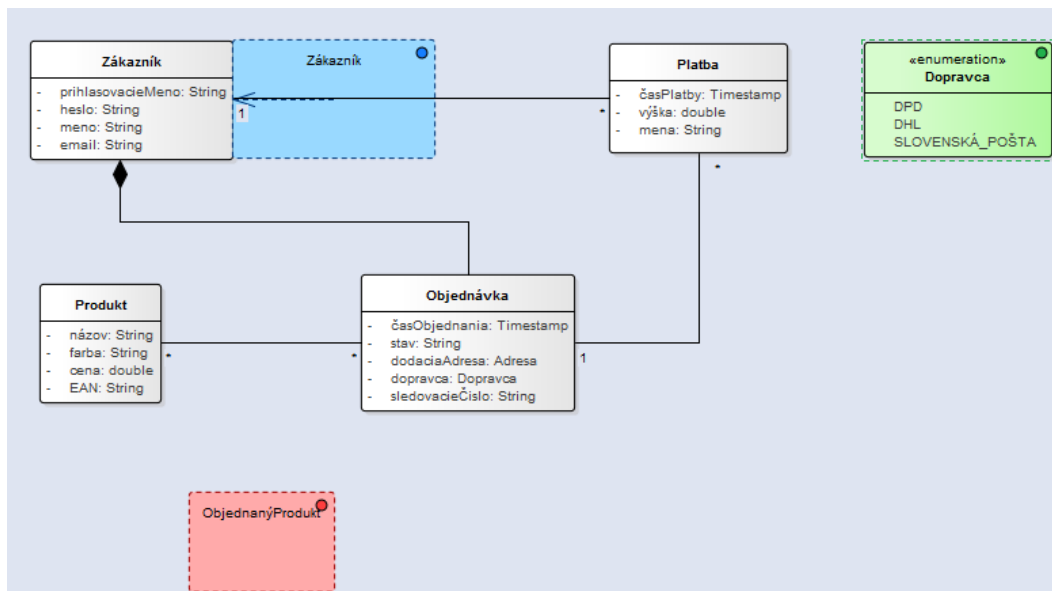
²<http://www.sparxsystems.com.au/>

³<http://www.sparxsystems.com.au/resources/diff/>

⁴<http://www.sparxsystems.com.au/resources/baseline/>

Baseline Comparison			
Comparing package Class Model against baseline version 1			
Model Elements	Property	Model	Baseline
Class Model	Alias		
Objednávka	Allow Duplicates	false	false
dodaciaAdresa	Collection	false	false
dopravca	Const	false	false
Dopravca	IsLiteral	false	false
«enum» DPD	Container		
«enum» DHL	Containment	Not Specified	Not Specified
«enum» SLOVENSKÁ POŠTA	Default		
Class Model	IsDerived	false	false
Visual Elements	IsOrdered	false	false
ObjednanýProdukt	Length	0	0
Zákazník	Lower Bound	1	1
Dopravca	Name	dopravca	dopravca
Visual Connectors	Notes		
Association	Position	3	3
Aggregation	Precision	0	0
	Scale	0	0
	Static	false	false
	Scope	private	Private
	Stereotype		
	Type	Dopravca	String
	Upper Bound	1	1
	Qualifier		

(a) Tabulkový prehľad zmien.



(b) Vizualizácia zmien priamo v diagrame.

Obrázok 4: Vizualizácia zmien pomocou Diff modulu nástroja Sparx Enterprise Architect³.

sú zobrazené hodnoty vlastností aktuálnej verzie modelu a v stĺpci *Baseline* sú zobrazené hodnoty porovnáwanej verzie modelu. Zelenou farbou je zvýraznená vlastnosť a hodnoty, ktoré boli medzi verziami zmenené.

Na obrázku 4b je zobrazená vizualizácia zmien priamo v diagrame porovnávaného modelu. Podporované sú len tri typy zmien, a to presun prvku v rámci diagramu (modrá farba), odstránenie prvku z diagramu (červené prerušované orámovanie) a pridanie prvku do diagramu z modelu (zelená farba). Vizualizácia typov iných zmien, akými sú napríklad pridanie alebo zmena vzťahu a pridanie alebo zmena atribútov a ich vlastností, nie je podporovaná priamo v diagrame (je podporovaná iba v tabuľkovom prehľade), čo môžeme vidieť na zmene typu atribútu *dopravca* v triede *Objednávka* z typu *String* na typ *Dopravca*. Taktiež nie sú zobrazené pôvodné atribúty a vzťahy spájacej entity *ObjednanýProdukt*, ktorá bola odstránená.

Enterprise Architect podporuje exportovanie a importovanie modelov vo formáte XMI, pričom používa vlastné rozšírenie, a to najmä na špecifikáciu pozície v rámci diagramu. Taktiež podporuje exportovanie a importovanie XMI súboru pre snímky, t. j. pre verzie, pričom takýto XMI súbor obsahuje iba detegované zmeny medzi verziami modelov.

3.1.5 IBM Rational Rhapsody DiffMerge

IBM Rational Rhapsody DiffMerge⁶ je samostatným porovnávacím a vizualizačným modulom nástroja IBM Rational Rhapsody⁵. Tento nástroj predstavuje riešenie pre modelovanie a návrh softvérových systémov, pričom podporuje jazyky UML a SysML a štandard AUTOSAR.

Modul DiffMerge podporuje vizualizáciu a detekciu zmien iba v modeloch vytvorených v nástroji IBM Rational Rhapsody, avšak do tohto nástroja je možné importovať modely vo formáte XMI. Ukážku vizualizácie zmien medzi dvoma verziami modelu (diagramu tried) môžeme vidieť na obrázku 5. Podobne ako aj nástroj Sparx Enterprise Architect, ktorý sme analyzovali v podkapitole 3.1.4, modul DiffMerge poskytuje tabuľkový prehľad (obrázok 5a) a prehľad priamo v diagrame (obrázky 5b a 5c).

⁵<https://www.ibm.com/software/products/en/ratirhapfami/>

⁶https://ibm.biz/knowctr#SSB2MU_8.2.0/com.ibm.rhp.diffmerge.doc/topics/rhp_c_col_parallel_dev_with_diffmerge.html

IBM Rational Rhapsody DiffMerge - [Project Version1 : Version2]

File Edit View Tools Window Help

Project Version1 : Version2

Version1

Default

Dopravca

DHL

int

DPD

int

Enum

Model1

SLOVENSKA_POSTA

int

ObjednanyProdukt

Objednavka

Model1

casObjednania

dodaciaAdresa

dopravca

Dopravca

String

itsPlatba

itsProdukt

itsZakaznik

sledovacieCislo

stav

Platba

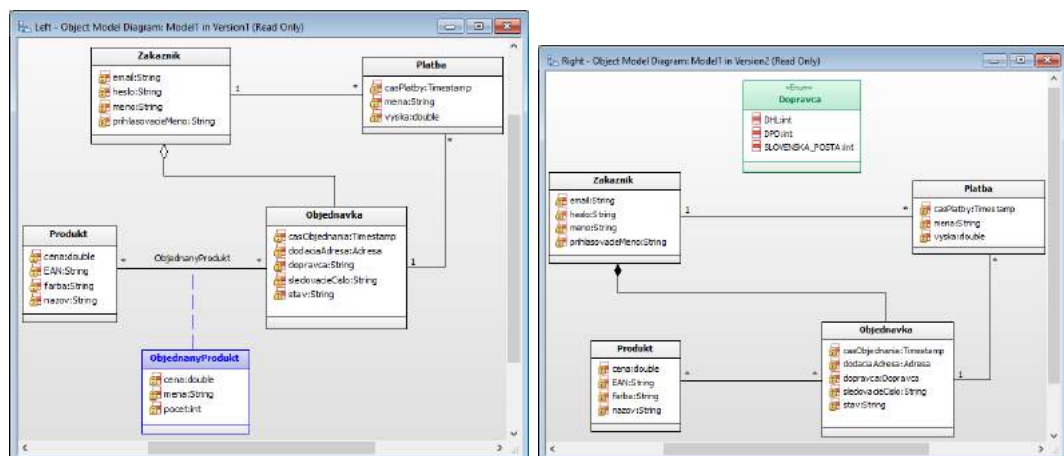
Produkt

Zakaznik

Attribute	Left Value	Right Value
partOfTheModelKind	localunit	localunit
unitSccProjName		
serverUrl		
Concurrency	sequential	sequential
isStub	0	0
persistAs		
displayName		
multiplicity		
description		
legalDisclaimer		
isReference	0	0
unitSccProjPath		
license		
name	Dopravca	Dopravca
classModifier	Unspecified	Unspecified
persistAsGenerated	0	0
language		
properties		
behaviorOverriden	0	0
Aggregate Order - Attributes		DPD DHL SLOVENSKA_POSTA

Project Version1 : Version2

(a) Tabulkový prehľad zmien.



(b) Vizualizácia zmien priamo v pôvodnom diagrame. (c) Vizualizácia zmien priamo v upravenom diagrame.

Obrázok 5: Vizualizácia zmien pomocou IBM Rational Rhapsody DiffMerge⁶.

V tabuľkovom prehľade na obrázku 5a sa v ľavej časti tabuľky nachádza stromová štruktúra všetkých prvkov modelu, pričom pri prvkoch v ktorých bola detegovaná zmena, je zobrazená ikonka vyjadrujúca typ zmeny (napr. pri atribúte *DHL* vymenovaného typu *Dopravca* sa nachádza zelený trojuholník, čo znázorňuje pridanie). Po kliknutí na jeden z prvkov modelu je zobrazený prehľad vlastností, pričom zmeny vo vlastnostiach sú zvýraznené tmavočervenou farbou, ako je to v prípade zmeny atribútov vymenovaného typu *Dopravca*, ktorá je zobrazená v tabuľke.

Vizualizácia zmien v diagramoch prebieha samostatne pre prvú verziu diagramu (tzv. ľavý model), ktorú môžeme vidieť na obrázku 5b a druhú verziu diagramu (tzv. pravý model), ktorý môžeme vidieť na obrázku 5c. Modul Diff-Merge podporuje niekoľko typov zmien, ktoré dokáže zvýrazniť v diagrame a sú nimi (i) pridanie nového prvku do diagramu, ktoré je vizualizované zelenou farbou (vymenovaný typ *Dopravca* na obrázku 5c), (ii) odstránenie prvku z diagramu, ktoré je vizualizované modrou farbou (spájacia entita *Objednaný-Produkt* na obrázku 5b) a (iii) odstránenie, resp. pridanie vzťahov, ktoré sú vizualizované rovnakými farbami ako rovnaké operácie pre prvky.

Vizualizácia zmien v atribútoch priamo v diagrame podporovaná nie je, čo môžeme vidieť na zmene typu atribútu *dopravca* v triede *Objednavka* z typu *String* na typ *Dopravca*. Taktiež nie je podporovaná vizualizácia zmeny typu vzťahov (medzi verziami modelov bola zmenená kompozícia triedy *Objednavka* do triedy *Zakaznik* na agregáciu) a ani zmena pozície prvku v rámci diagramu (medzi verziami modelov bola presunutá trieda *Zakaznik*).

3.2 Upriamenie pozornosti

Neoddeliteľnou súčasťou vizualizácie sú techniky upriamania, resp. vyžiadania si pozornosti pozorovateľa na určitú časť plochy, na ktorej prebieha vizualizácia. Taktiež je nutné nezahliť pozorovateľa veľkým množstvom informácií, ale umožniť mu ich filtrovať, prípadne vytvoriť odvodený pohľad, ktorý bude poskytovať zjednodušený prehľad vizualizovaných informácií (Godfrey et al., 2015). Avšak pri použití metód na upriamenie pozornosti je dôležité poskytnúť a zachovať kontext informácie na ktorú je upriamená pozornosť (Hauser, 2005), či už ide o softvérový model, resp. zmeny v softvérových modeloch alebo inú,

napr. grafickú informáciu. Práve zachovanie kontextu pri upriamení pozornosti zabezpečujú metódy pozornosť+kontext (angl. *focus+context*) (Cockburn et al., 2008).

3.2.1 Rybie oko

Jednou z metód zo skupiny pozornosť+kontext je metóda s názvom *rybie oko* (angl. *fisheye*), ktorá nachádza využitie aj pri grafových štruktúrach (Tominski et al., 2006). Táto metóda zabezpečuje vizuálne priblíženie (angl. *zoom*) dôležitej informácie s relatívnym zachovaním kontextu. Problémom však je prípadné skreslenie kontextu v blízkosti plochy, na ktorú je táto metóda aplikovaná, čo môže spôsobiť nesprávnu interpretáciu dát a sťaženie výberu ďalšej časti plochy, na ktorú ma byť táto metóda aplikovaná (Cockburn et al., 2008).

V prípade použitia tejto metódy na upriamenie pozornosti v kontexte vizualizácie zmien v softvérových modeloch je možné, že pozornosť bude upriamená aj na nerelevantné časti modelu, na ktoré pozornosť nemá byť upriamená, napr. vzťahy, ktoré sú v blízkosti relevantného elementu, ale nijak s ním nesúvisia. Čiastočné riešenie tohto problému je možné nájsť v použití dodatočných vrstiev (šošoviek), resp. filtrov, ktoré zvýrazňujú iba lokálne hrany (Wong et al., 2003), prípadne ich kombinácia s metódou *fisheye* pomocou metódy *MultiLens* (Kister et al., 2016).

Metóda *MultiLens* (Kister et al., 2016) poskytuje kombinovanie niekoľkých metód (šošoviek) typu pozornosť+kontext, a to *fisheye*, metódu pripoj suseda (angl. *Bring Neighbor*) (Tominski et al., 2006), metódu *K-susedstvo* (angl. *K-Neighborhood*) (Kister et al., 2016) a metódu lokálna hrana (angl. *Local Edge*) (Wong et al., 2003). Ukážku metódy *MultiLens* môžeme vidieť na obrázku 6, pričom na obrázku 6a môžeme vidieť individuálne použitie šošoviek typu lokálna hrana (rúžový okraj) a *fisheye* (modrý okraj). Môžeme vidieť, že pod šošovkou typu *fisheye* sú okrem zväčšeného bodu pozornosti (záujmu) – fotografiou, zobrazené aj hrany, ktoré nesúvisia s bodom pozornosti. Naopak, pod šošovkou typu lokálna hrana sú zobrazené iba súvisiace hrany.

Kombinácia týchto typov šošoviek je zobrazená na obrázku 6b. Môžeme vidieť, že šošovka typu *fisheye* spôsobuje zväčšenie bodu pozornosti – fotografie a šošovka typu lokálna hrana zabezpečuje zobrazenie iba relevantných hrán vo



(a) Šošovky lokálna hrana (rúžová) a fisheye (modrá).



(b) Kombinácia šošoviek typu fisheye a lokálna hrana.

Obrázok 6: Ukážka metódy MultiLens (Kister et al., 2016).

vybranej oblasti.

3.2.2 Hĺbka poľa

Využitie hĺbky poľa predstavuje ďalší z prístupov zo skupiny pozornosť+kontext. Tento prístup pochádza z oblasti fotografie a predstavuje najčastejšiu metódu na upriamovanie pozornosti na určitú oblasť v 2D obrázkoch (Bailey et al., 2009) na základe skreslenia (rozmazania) menej relevantných objektov na základe vzdialenosti od šošovky kamery – najvzdialenejšie objekty od šošovky sú rozmazané.

Tento prístup bol aplikovaný aj na 3D objekty pod názvom sémantická hĺbka poľa (Kosara et al., 2001), avšak bez využitia vzdialenosti od šošovky kamery. Rozhodnutie, či objekt má alebo nemá byť rozmazaný nezáleží teda na vzdialenosti šošovky od daného objektu ale na jeho relevantnosti – sémanticky nerelevantné objekty sú úplne rozmazané. Taktiež, oproti tradičnému využitiu hĺbky poľa, tento prístup neposkytuje žiaden gradient rozmazania na základe relevantnosti, teda relevantnosť je binárna – objekty na obrázku alebo v scéne sú buď relevantné alebo nie. Rozmazanie nerelevantných objektov je prevádzané na základe vzdialenosti virtuálnej šošovky od objektov. V prípade 2D obrázku je vzdialenosť braná od šošovky k rovine, v ktorej sa objekty nachádzajú – ak je kamera zaostrená na súradnicu z_{focus} , objekty, ktoré majú byť rozmazané musia byť premiestnené do vzdialenosti z od šošovky. V prípade



(a) *Sémantická hĺbka poľa* (Kosara et al., 2001).



(b) *Použitie tieňov (zošednutia)* (Cole et al., 2006).

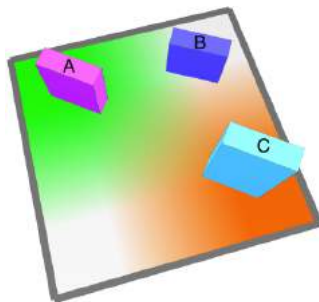
Obrázok 7: Ukážka prístupov využívajúcich hĺbku poľa.

3D scény je použitý adaptívny model kamery, ktorý je v princípe rovnaký ako model kamery pre 2D obrázky, avšak rozdiel spočíva v dynamickom určovaní vzdialenosti pre každý bod nachádzajúci sa v scéne. To znamená, že body sa nenachádzajú v rovnakej vzdialenosti od šošovky, ale ich vzdialenosť je vypočítavaná dynamicky. Ukážku tohto prístupu môžeme vidieť na obrázku 7a.

Tieto techniky vo veľkej miere ovplyvňujú celkový vzhľad obrázku a v prípade 3D prostredia celkový vzhľad scény – jednotlivé objekty sú skreslené a zdeformované. Tento problém bol čiastočne vyriešený použitím tieňov (zošednutia), resp. zmien osvetlenia a modulácie teplých a studených farieb v nerelevantných oblastiach scény (Cole et al., 2006). Tento prístup podporuje štyri modely zaostrenia, a to (i) segmentáciu (úroveň zaostrenia je určená pre každý relevantný objekt), (ii) 2D ohniskový bod (úroveň zaostrenia je určená ako vzdialenosť bodu v scéne od 2D ohniskového bodu), (iii) 3D ohniskový bod (úroveň zaostrenia sa radiálne zmenšuje podľa vzdialenosti bodu v scéne od 3D ohniskového bodu) a (iv) ohniskovú rovinu (úroveň zaostrenia je určená ako vzdialenosť bodu od ohniskovej roviny). Ukážku tohto prístupu môžeme vidieť na obrázku 7b.

3.2.3 Teplotná mapa

Ďalšou z vizualizačných techník, ktorá nepriamo podporuje vyžiadanie si pozornosti používateľa na určité časti zobrazených dát, je teplotná mapa (angl. *heat map*). Teplotná mapa je reprezentovaná ako 2D vrstva, ktorá používa



Obrázok 8: Vizualizácia frekvencie zmien v balíkoch softvéru (Benomar et al., 2013).

teplotu (farbu) na zvýraznenie častí vizualizovaných dát podľa ich dôležitosti, prípadne intenzity a iných podobných kritérií (Röthlisberger et al., 2009). Menšie dôležité časti sú teda zvýraznené modrou (chladnou) farbou a naopak, najdôležitejšie časti sú zvýraznené červenou (teplou) farbou.

Táto technika sa v kontexte pozornosti a sledovania pohľadu používa prevažne na vizualizáciu dát zozbieraných pomocou sledovača očí (angl. *eye-tracker*) v 2D priestore (Bojko, 2009) ale aj v 3D priestore (Maurus et al., 2014). Je ju však možné použiť aj na vizualizáciu dynamických vlastností softvéru. Príkladom dynamickej vlastnosti softvéru je frekvencia zmien vykonaných v určitých častiach softvéru, napr. triedach alebo balíkoch (Benomar et al., 2013). Príklad takéhoto typu použitia teplotnej mapy môžeme vidieť na obrázku 8. Môžeme vidieť, že stupeň teploty pri balíku A je studený (zelený) a pri balíku C teplý (červený), čo znamená, že frekvencia zmien v balíku A je nižšia ako v balíku C, t. j. balík A sa mení pomalšie ako balík C. Balík B, ktorý nebol menený, nemá žiaden stupeň teploty. Podobným spôsobom je možné vizualizovať aj iné vlastnosti softvéru ako zmeny, napr. počet testov alebo volaných metód z danej triedy a pod.

3.2.4 Mapa nápadnosti

Mapa nápadnosti (angl. *saliency map*, obrázok 9) predstavuje grafickú reprezentáciu modelu nápadnosti (angl. *saliency model*), ktorý predpovedá pohyb zraku počas prezerania vykreslených obrázkov a scén, resp. akýchkoľvek vizualizovaných údajov (Koehler et al., 2014) a vychádza z vlastností akými sú kontrast, okrajový obsah, intenzita, orientácia a farba (Xu et al., 2014). Túto predpoveď je následne možné použiť na identifikáciu častí, ktoré priťa-



Obrázok 9: Mapa nápadnosti pre scénu ulice (Torralba et al., 2006).

hujú pozornosť (Judd et al., 2009). Existujú dva prístupy vytvárania modelu nápadnosti a to prístup zdola-nahor a prístup zhora-nadol.

V prípade prístupu zdola-nahor sa model nápadnosti vytvára na základe porovnania objektov v danej scéne, resp. obrázku (Itti a Koch, 2000). Problém však nastáva v prípade, že používateľ je ovplyvnený kontextom, t. j. hľadá iba časti, ktoré ho zaujímajú a na obrázok sa nepozera ako na celok (Melloni et al., 2012). Práve toto rieši prístup vytvárania modelu nápadnosti zhora-nadol.

Prístup zhora-nadol berie do úvahy objekt záujmu (používateľ teda hľadá konkrétny objekt) a modifikuje spôsob akým sú použité globálne vlastnosti na výber relevantných častí obrázku. Týmto sa dosahuje vyššia miera správnosti predikcie častí obrázkov, na ktoré sa používateľ pozerá v danom kontexte (Torralba et al., 2006), ktorým môže byť napr. úloha hľadania konkrétneho predmetu.

3.3 Zhrnutie a diskusia

V tejto kapitole sme uviedli spôsoby vizualizácie zmien v softvérových modeloch spolu s metódami na upriamenie pozornosti. Na základe analýzy existujúcich riešení sme identifikovali viaceré nedostatky:

- obmedzená podpora vizualizácie zmien len pre vybrané typy diagramov,
- chýbajúca vizualizácia presunov vlastností prvkov (napr. atribútov a ich typov v prípade tried),
- chýbajúca podpora štandardného formátu pre UML diagramy (XMI),

- prípadne nemožnosť exportovania a importovania existujúcich modelov,
- obmedzená podpora typov zmien,
- chýbajúca podpora vizualizácie zmien určitých prvkov (napr. vzťahov a stereotypov),
- veľké množstvo použitých farieb na vizualizáciu zmien a
- vizualizácia zmien spôsobom zobrazenia dvoch verzií modelu.

Všetky tieto nedostatky predstavujú veľký problém súvisiaci s flexibilitou vizualizácie zmien. Ide najmä o veľké obmedzenie podpory vizualizácie zmien len na vybrané typy diagramov a ich elementov a o chýbajúcu podporu vizualizácie pre už existujúce verzie modelov v prípade niektorých nástrojov.

V prípade vizualizácie zmien spôsobom zobrazenia pôvodnej a upravenej verzie modelu vedľa seba, domnievame sa, že takýto typ vizualizácie, ktorý sa zvykne používať pri vizualizácii zmien v texte, nie je pre softvérové modely vhodný. Aj napriek tomu, že softvérový model, podobne ako text, je štruktúrovaný, pri zmenách textu sa nevyskytujú viaceré typy prvkov (vyskytujú sa len znaky) ako je to v prípade softvérových modelov. Je preto vhodnejšie použiť vizualizáciu v rámci jedného modelu, v ktorom sú zobrazené zmeny medzi verziami modelov s možnosťou ich filtrovania podľa typu.

V prípade niektorých typov zmien, akým je napríklad presun atribútu medzi dvoma triedami, je vhodné použiť techniku na upriamanie pohľadu z dôvodu prehľadnosti. V prípade, že by takáto technika nebola použitá, samotná vizualizácia presunu by sa mohla stať neprehľadnou – v prípade vizualizácie pomocou šípky by táto šípka mohla pretínať vzťahy medzi prvkami v modeli. Z techník, ktoré sme analyzovali, je vhodnejšie použiť techniku využívajúcu hĺbku poľa. Techniku Fisheye nie je vhodné použiť najmä z dôvodu zobrazenia aj nepotrebných častí modelu v prípade použitia jednej šošovky. Čiastočné riešenie sa nachádza v použití viacerých šošoviek naraz v blízkosti relevantných častí modelu. Toto však nie je vhodné najmä z dôvodu potenciálneho zobrazenia nerelevantných častí modelu v ich blízkosti (napr. počiatočné časti nerelevantných vzťahov vychádzajúcich z triedy). V prípade hĺbky poľa budú ostro zobrazené len časti relevantné zmeny a nerelevantné časti budú rozmazané, prípadne zošednuté.

Taktiež sme analyzovali metódy teplotnej mapy a mapy nápadnosti z dô-

vodu poskytnutia globálneho prehľadu zmien v modeli, pričom tieto techniky by sme použili reverzne, t. j. nepoužili by sme ich na detekciu miest, na ktoré sa používateľ pozerá ale na zvýraznenie miest na ktoré sa má pozerieť. Z týchto dvoch techník preferujeme techniku teplotnej mapy. Mapa nápadnosti je síce taktiež vhodná na poskytnutie takéhoto prehľadu v prípade, že sa použije ako maska, ale problém sa nachádza v odlišovaní typov vizualizovaných zmien. V prípade použitia mapy nápadnosti by nebolo možné poskytnúť prehľad všetkých typov zmien, keďže táto technika pracuje v princípe iba s jednou farbou. Zmeny by teda bolo možné vizualizovať iba podľa typov použitím filtrov. V prípade teplotnej mapy je ju možné upraviť a farby teploty zameniť za farby zodpovedajúce typom zmien a počet zmien vizualizovať veľkosťou časti teplotnej mapy v prípade prehľadu všetkých zmien. Pri použití filtrov na zobrazenie konkrétneho typu zmien by bola použitá klasická teplotná mapa – teplota by zodpovedala počtu konkrétneho typu zmeny.

4 Metóda detekcie zmien v softvérových modeloch

Na základe analýzy existujúcich algoritmov zameraných na detekciu zmien v modeloch sme vyslovili pochybnosti o efektivite a dostatočnej abstrakcie týchto algoritmov. Cieľom nami navrhovanej metódy na detekciu zmien v softvérových modeloch správnosť a dostatočná všeobecnosť na to, aby ju bolo možné použiť na rôzne reprezentácie softvérových modelov.

Vzhľadom na dané skutočnosti sme sa rozhodli aplikovať všeobecný algoritmus určený na verifikáciu veľkých dátových štruktúr – Merkleho strom (Merkle, 1988). Tento algoritmus sme pre naše potreby upravili z dôvodu zjednodušenia reprezentácie a zlepšenia časovej zložitosti. Okrem tohto algoritmu používame aj algoritmus najdlhšej podobnej podpostupnosti (angl. *Longest Common Subsequence*) pre textové časti, ktoré sa môžu nachádzať v modeloch najčastejšie vo forme poznámok a opisov jednotlivých elementov. Naša metóda pozostáva z troch krokov:

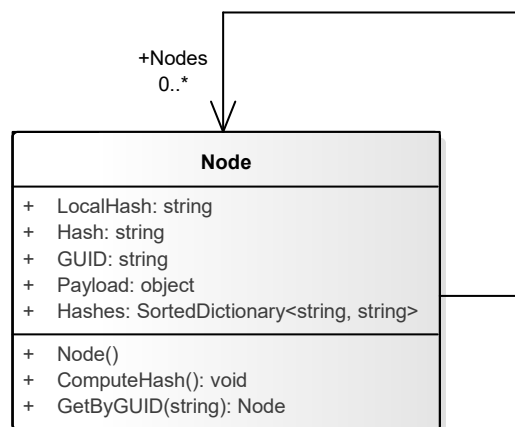
1. vytvorenie upraveného Merkleho stromu,
2. prechádzanie vytvorenej stromovej štruktúry z dôvodu detekcie zmien a
3. spracovanie detegovaných zmien a vytvorenie prenosových objektov pre ďalšie spracovanie.

Zmeny je možné detegovať aj na základe aktivít vývojárov (Ondik et al., 2017). Takýto spôsob priamo poskytuje dáta vhodné ako základ pre vizualizáciu zmien, avšak je nutné, aby dané aktivity boli zozbierané od všetkých vývojárov spolupracujúcich na danej verzii modelu. Naša metóda rieši práve tento nedostatok.

4.1 Vytvorenie upraveného Merkleho stromu

Vytvorenie upraveného Merkleho stromu je prvým a najdôležitejším krokom nami navrhovanej metódy. Pre potreby objasnenia tohto kroku budeme používať stromovú štruktúru UML modelov.

V prvej fáze tohto kroku sa načítajú už existujúce UML modely určené



Obrázok 10: Nami navrhovaná reprezentácia uzla pre dátovú štruktúru Merkleho stromu.

na porovnanie, ktoré sú reprezentované pomocou XML. Na načítanie takýchto modelov používame štandardizovanú definíciu XMI v kombinácii s definíciou rozšírenia, ktoré povoľuje UML, pre Sparx Enterprise Architect. Toto rozšírenie používame z dôvodu zachovania vnútorných vzťahov modelovaných pomocou menovaného modelovacieho nástroja, akými sú napríklad pozície v rámci diagramov, ktoré je vhodné ponechať napríklad pre vizualizáciu jednotlivých elementov a ich zmien.

Po načítaní modelov prichádza fáza konverzie na upravený Merkleho strom zložený z nami navrhnutej dátovej štruktúry pre uzly stromu. Reprezentáciu uzla nami upraveného Merkleho stromu je možné vidieť na obrázku 10. Uzol v tomto prípade predstavuje inštanciu typu *PackagedElement* jazyka UML. Táto štruktúra obsahuje nasledujúce polia:

- *LocalHash*, ktoré v sebe nesie hodnotu rozptylovej funkcie aplikovanej na postupnosť hodnôt rozptylovej funkcie aplikovanej vlastností reprezentovaného elementu modelu,
- *Hash* predstavujúce hodnotu rozptylovej funkcie aplikovanej na zlúčené hodnoty poľa *LocalHash* a hodnôt polí *Hash* listov daného uzla,
- *GUID* reprezentujúce unikátny alfanumerický identifikátor v rámci verzie modelu,
- *Payload* nesúce referenciu na pôvodný element modelu,
- *Hashes* obsahujúce hodnoty rozptylovej funkcie aplikovanej na jednotlivé

vlastnosti elementu modelu a

- *Nodes* predstavujúce zoznam listov vzhľadom na daný uzol v strome.

Algoritmus 1 Vytvorenie Merkleho stromu

```
1: function VYTVORSTROM(model)
2:   strom  $\leftarrow$  new Node()
3:   for každý prvok modelu do
4:     strom.Nodes.Add(GenerateNode(prvokModelu))
5:   end for
6:   strom.ComputeHash()
7:   return strom
8: end function
9: procedure COMPUTEHASH
10:  for každý uzol v Nodes do
11:    uzol.ComputeHash()
12:  end for
13:  generatedHash  $\leftarrow$  ComputeHashForPayload(this)
14:  LocalHash  $\leftarrow$  generatedHash.LocalHash
15:  Hashes  $\leftarrow$  generatedHash.Hashes
16:  for každý uzol v Nodes do
17:    Hash  $\leftarrow$  Hash xor uzol.Hash
18:  end for
19:  Hash  $\leftarrow$  rozptylovaFunkcia(Hash)
20: end procedure
```

Daná štruktúra taktiež obsahuje operáciu *ComputeHash()*, ktorá rekurzívne volá rovnakú metódu na listy uzla a vypočítava hodnotu rozptylovej funkcie aplikovanej na vlastnosti daného uzla a pomocnú operáciu *GetByGuid(GUID: string)*, ktorá sa používa na získanie uzla reprezentujúceho element modelu s daným unikátnym identifikátorom GUID.

Do množiny inštancií takto navrhutej dátovej štruktúry sa postupným prechádzaním konvertujú načítané modely. Po konverzii je nad koreňom vytvorených stromov zavolaná už zmienená operácia *ComputeHash()*, ktorá rekurzívne vypočíta hodnoty rozptylovej funkcie pre jednotlivé uzly a uloží ich do polí na to určených, t. j. *LocalHash*, *Hash* a *hashes*. Takto naplnená stromová štruktúra upraveného Merkleho stromu je pripravená na prechádzanie určené na samotnú detekciu rozdielov, t. j. porovnávanie hodnôt vypočítaných

rozptylovou funkciou.

4.2 Prechádzanie vytvorenej stromovej štruktúry

Po vytvorení a naplnení stromovej štruktúry prebieha krok jej prechádzania za účelom detekcie rozdielov. Prechádzanie prebieha súbežne pre všetky inštancie (modely) pričom do podstromov jednotlivých uzlov sa vchádza iba v prípade, že ich hodnoty podľa *Hash* vypočítané rozptylovou funkciou sa nezhodujú. V takomto prípade sa v daných nezhodujúcich sa podstromoch hľadajú konkrétne uzly, ktorých hodnoty rozptylovej funkcie sú zhodné. V prípade, že sa takýto uzol nájde, rozdiel je detegovaný v rodičovských uzloch. Takýto uzol sa následne vyhľadá v inštancii stromu porovnávaného modelu podľa hodnoty podľa *LocalHash*. Následne sa porovnávajú hodnoty vlastností takto napárovaných uzlov, pričom výsledok porovnania sa berie za zmenu. Rodičovské uzly sú následne porovnávané pre prípadne rozdiely pomocou hodnoty podľa *LocalHash*. Zmeny v textových vlastnostiach, akými sú napríklad poznámky a popisy, sa dodatočne detegujú podľa algoritmu najdlhšej spoločnej podpostupnosti.

V prípade, že sa nepodari vyhľadať uzol s danou hodnotou podľa *LocalHash*, takáto skutočnosť je vyhodnotená ako zmena typu *odstránenie*. V prípade takejto zmeny sa taktiež porovnávajú hodnoty podľa *LocalHash* v rodičovských uzloch pre prípadné zmeny. Naopak, pokiaľ nejaký uzol zostane v porovnávanom modeli nenapárovaný, považuje sa takáto skutočnosť za zmenu typu *pridanie*. Toto zabezpečuje napárovanie prvkov v diagramoch aj v prípade, že bol daný prvok zmazaný a následne vytvorený s totožnými vlastnosťami, prípadne bol presunutý v rámci modelu.

Detegované zmeny sú podľa typu rozdelené do troch množín, t. j. do množiny pre *zmeny*, do množiny pre *odstránenia* a do množiny pre *pridania*. Tieto množiny sú pripravené na transformáciu do jednotnej formy vhodnej pre ďalšie použitie, prípadne spracovanie.

4.3 Spracovanie detegovaných zmien

V tomto kroku sú jednotlivé množiny obsahujúce detegované zmeny transformované na jednotné prenosové objekty (z angl. *Data Transfer Objects*). Jed-

Difference	
+	Type: string
+	OriginalValue: dynamic
+	CurrentValue: dynamic
+	OriginalObject: dynamic
+	CurrentObject: dynamic
+	GetAllDifferences(): List<Difference>

Obrázok 11: Štruktúra jednotného prenosového objektu pre zmenu.

notná forma zabezpečuje jednotný spôsob interakcie s rôznymi typmi zmien počas ďalšieho spracovania, v našom prípade vizualizácie zmien. Avšak taktiež ju je možné použiť ako základ pre iné operácie, napríklad pre operáciu synchronizácie a spájania viacerých verzií modelov. Štruktúru jednotného prenosového objektu zmien je možné vidieť na obrázku 11.

Pole *Type* predstavuje typ zmeny. Príkladmi sú *NameChange* v prípade zmeny názvu prvku a *Addition* v prípade pridania prvku. Pole *OriginalValue* predstavuje pôvodnú hodnotu prvku alebo atribútu prvku v modeli. Toto pole, rovnako ako aj *CurrentValue*, je využité len pri vyžiadaní si jednotlivých zmien pre daný prvok zavolaním metódy *GetAllDifferences()* tejto triedy. Táto metóda vráti zmeny prvku v modeli na nižšej granularite. V prípade, že v prvku bola zmenená len jedna jeho vlastnosť, metóda vráti objekt, nad ktorým bola volaná (pôvodnú inštanciu triedy *Difference*). Polia *OriginalObject* a *CurrentObject* obsahujú celý pôvodný objekt pred zmenami, respektíve aktuálny objekt modelu po zmenách.

5 Metóda vizualizácie zmien v softvérových modeloch

Nami navrhovaná metóda sa zameriava na vizualizáciu zmien v softvérových modeloch vytvorených pomocou jazyka UML⁷. V podkapitole 3.1 sme uvideli niekoľko metód a nástrojov podporujúcich vizualizáciu zmien v UML modeloch. V podkapitole 3.3 sme uvideli niekoľko vážnych nedostatkov, ktoré sme pri analýze a testovaní týchto metód a nástrojov odhalili. Išlo najmä o nepostačujúcu podporu typov UML diagramov, chýbajúcu podporu spoločného formátu na výmenu dát a vizualizácia zmien zobrazením oboch verzií diagramov, o ktorej sa domnievame, že pre vizualizáciu UML modelov nie je vhodná. Na základe týchto poznatkov si volíme nasledujúcu hypotézu, ktorej dôkaz bude naším cieľom:

Hypotéza 1. *Zobrazenie zmien medzi dvomi verziami modelov v jednom pohľade, resp. prehľadovom okne, zmenší pravdepodobnosť prehliadnutia zmien.*

V tejto práci zameriavame na tímy analytikov a podporu ich diskusie o zmenách. Na základe tohto faktu si volíme ďalšie dve hypotézy, ktoré chceme dokázať:

Hypotéza 2. *Sprístupnenie metódy (nástroja) umožňujúceho vizualizáciu zmien medzi verziami UML modelov zefektívni prácu softvérových analytikov.*

Hypotéza 3. *Poskytnutie metódy (nástroja) umožňujúceho vizualizáciu zmien medzi verziami UML modelov podporí diskusiu medzi analytikmi v tíme.*

V nasledujúcich podkapitolách opisujeme nami navrhovanú metódu pre statické a dynamické typy UML diagramov, opisujeme riešenie vizualizácie zmien v rozsahovo veľkých modeloch a uvádzame možnosť personalizovaného prispôsobenie farebnej schémy vizualizácie zmien.

⁷<http://www.omg.org/spec/UML/2.4/>

5.1 Statické časti modelov

Modelovací jazyk UML sa delí na dve hlavné kategórie. Sú nimi štrukturálne (statické) a behaviorálne (dynamické) diagramy, pričom behaviorálne diagramy obsahujú dodatočnú kategóriu, ktorou sú diagramy interakcie. V tejto kapitole sa budeme zaoberať štrukturálnymi diagramami, nakoľko je nutné vizualizáciu prispôbovať samostatne pre každú kategóriu UML diagramov. Je to z toho dôvodu, že jednotlivé kategórie UML diagramov modelujú odlišné pohľady a teda vplyv a spôsob vizualizácie zmien, na ktoré je potrebné upozorniť, sa líšia.

Na obrázku 12 môžeme vidieť základný pohľad vizualizácie zmien v diagrame tried. Na samotnú vizualizáciu sú použité iba 4 farby, pričom sa vizualizujú iba reálne zmeny a nie miesto zmien, ako je to v podobných vizualizačných nástrojoch, ktoré sme analyzovali – zvýraznenie celej operácie aj v prípade, že bol modifikovaný iba typ parametra, pričom na takýto typ modifikácie je použitá samostatná farba. Pridanie a odstránenie triedy je vizualizované zvýraznením okraja danej triedy – napr. zvýraznenie celého obsahu triedy by mohlo byť neprehľadné. Zmena, pridanie a odstránenie vzťahu je vizualizované zvýraznením modifikovanej časti. Podobne sú vizualizované aj zmeny ďalších prvkov diagramu (početnosti, názvy vzťahov a pod.), ktoré nie sú zvýraznené na tomto obrázku. Naša metóda taktiež podporuje vizualizáciu zmien v popisoch jednotlivých prvkov, čo môžeme vidieť na triede *EntitlementType*, ktorá obsahuje zvýraznenú ikonu poznámky oranžovou farbou, čo predstavuje zmenu. Prípady, kedy bol prvok v diagrame iba skrytý a teda nebol definitívne odstránený z daného modelu, naša metóda rieši aplikáciou vzoru (napr. aplikovaním tieňovaných diagonálnych pásov).

Na obrázku 13 je znázornená interaktívna vizualizácia zmien. Po presunutí kurzora (v prípade virtuálnej reality ukázanie prstom alebo namierenie pohľadu) na miesto konkrétnej zmeny, sa zobrazia pôvodné vzťahy, hodnoty a pod., ktoré boli modifikované. V prípade presunutia kurzora na zmenu typu presunutie sa zobrazí šípka vychádzajúca z miesta, kde boli jednotlivé časti triedy pôvodne umiestnené. Nerelevantné prvky diagramu, ktorých sa presun netýka sú rozmazané, čím sa zvýši prehľad v diagrame a upriami pozornosť na relevantné časti diagramu (obrázok 14). Na toto používame techniku typu pozornosť+kontext (angl. *focus+context*), konkrétne techniku sémantickej hĺbky



Obrázok 14: Pohľad na konkrétnu zmenu typu presun v diagrame tried našou metódou.

poľa, ktorú sme opísali v podkapitole 3.2.2. Presunutie celej triedy v rámci diagramu nie je vizualizované.

5.2 Dynamické časti modelov

Ako sme naznačili v predchádzajúcej kapitole, jednotlivé kategórie UML diagramov vyžadujú rôzne spôsoby vizualizácie zmien, nakoľko tieto zmeny majú odlišný vplyv. Príkladom je presun správy v diagrame sekvencií. Takáto zmena ovplyvní význam všetkých správ, ktoré sa nachádzajú v postupnosti za ňou. Naopak, presun triedy v diagrame tried nemá okrem vizuálnej zmeny žiaden vplyv na význam diagramu, ak teda nerátame zhlukovanie významovo podobných tried, resp. prvkov a samotnú čitateľnosť diagramu.

Ďalším dôležitým typom zmien je zmena týkajúca sa presunutia aktivity alebo stavu. Jednou z možných presunutí je vnorenie aktivity alebo stavu do vnútra inej aktivity alebo stavu – takýto typ zmeny je nutné vizualizovať. Avšak ak ide iba o presun aktivity alebo stavu v rámci diagramu, t.j. nejde o zmenu rodiča presúvaného elementu, takúto zmenu nie je nutné vizualizovať, nakoľko je vo väčšine prípadov iba dôsledkom inej zmeny, napr. pridaním alebo úpravou prepojení (tokov). Podobne ak je presunutá aktivita alebo stav s vnorenými elementami, nie je nutné vizualizovať presun všetkých prvkov a prepojení ale postačuje vizualizovať presun najvyššieho rodičovského elementu, ktorý obsahuje ostatné elementy a prepojenia.

5.3 Vizualizácia veľkých softvérových diagramov

Rozsahovo veľké diagramy (a modely) si vyžadujú dodatočný spôsob vizualizácie zmien. Je to najmä z toho dôvodu, že v niektorých prípadoch jednotlivé zmeny nie je dobre vidno, nakoľko sa v takýchto diagramoch “stratia”. Tieto problémy riešime poskytnutím prehľadu využitím metódy teplotnej mapy 15, ktorú sme opísali v podkapitole 3.2.3.

Túto metódu, oproti metóde mapy nápadnosti (opísaná v podkapitole 3.2.4), sme zvolili najmä z dôvodu lepšej vhodnosti pre použitie filtrov pre zobrazenie frekvencie zmien konkrétneho typu v modeli. Naša metóda bude teda poskytovať dva varianty teplotnej mapy, a to (i) variant, v ktorom budú

zobrazené všetky typy zmien, pričom teplo (angl. *heat*) bude reprezentované farbou korešpondujúcej typu zmeny a (ii) variant, v ktorom bude zobrazený jeden konkrétny typ zmeny, vybraný pomocou filtra, a teplo sa bude správať ako je to štandardom pri teplotnej mape, t. j. bude zobrazovať počet (frekvenciu) konkrétneho typu zmeny v jednotlivých častiach diagramu.

6 Overenie metódy detekcie zmien v softvérových modeloch

Overenie nami navrhutej metódy na detekciu zmien v softvérových modeloch sme vykonali na viacerých verziách rôznych modelov pomocou nami implementovaného prototypu tejto metódy. Medzi tieto modely patria nami vytvorené modely, ktoré sme vytvorili počas kurzov a taktiež aj projektov, na ktorých sme v určitej miere pôsobili. Jedná sa najmä o modely obsahujúce diagramy tried, diagramy aktivít a sekvenčné diagramy ku ktorým máme k dispozícii viacero verzií v rozdielnych fázach návrhu.

Ďalšou skupinou modelov, na ktorých sme metódu overovali, sú modely obsahujúce diagramy tried automaticky generované zo zdrojového kódu. Keďže máme k dispozícii historické verzie projektov, je možné vygenerovať diagramy v rozdielnych fázach vývoja a následne porovnávať výstup našej metódy s reálnymi zmenami v zdrojovom kóde. Taktiež sme overenie vykonali na priemyselných diagramoch poskytnutých spoločnosťou Continental Automotive Systems Slovakia s.r.o., pričom sa jedná najmä o diagramy aktivít a stavové diagramy.

6.1 Prototyp

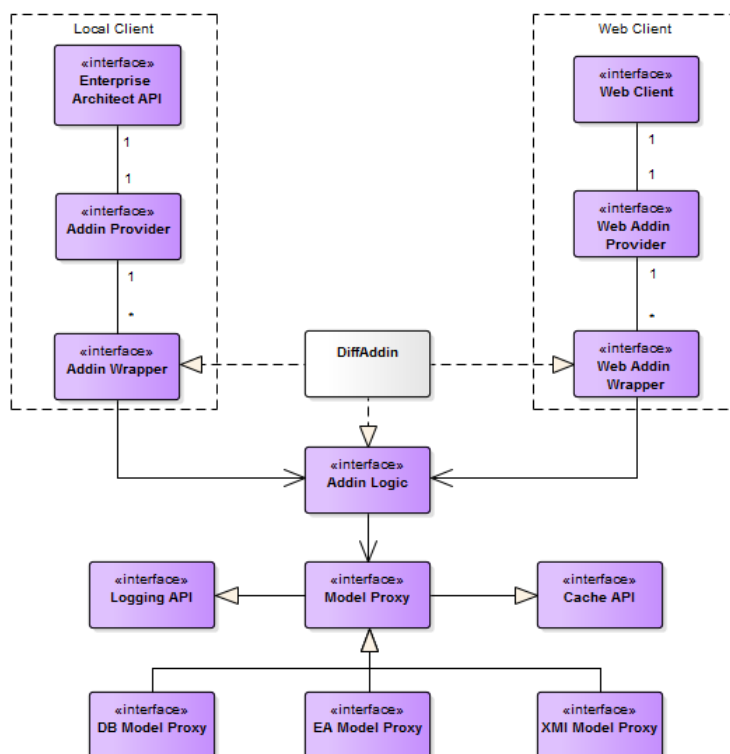
Prototyp metódy na detekciu zmien v softvérových modeloch sme implementovali ako rozšírenie nami navrhutej a implementovanej infraštruktúry zameranej na modelovanie softvéru⁸, ktorá je nadstavbou modulu rozšírení nástroja Sparx Systems Enterprise Architect⁹. Implementovaný prototyp podporuje mód porovnávania dvoch verzií modelu vo formáte XMI a mód porovnávania verzie modelu načítaného v nástroji Sparx Enterprise Architect s modelom vo formáte XMI.

Ako rozptylovú funkciu sme zvolili funkciu xxHash¹⁰ najmä z dôvodu nízkeho počtu kolízií a rýchlosti porovnateľnej s rýchlosťou pamäte. Túto funkciu používame na vypočítanie rozptylovej hodnoty pre jednotlivé uzly nami upra-

⁸<http://umlviz.fiit.stuba.sk/>

⁹<http://www.sparxsystems.com.au/>

¹⁰<http://cyan4973.github.io/xxHash/>



Obrázok 16: Architektúra rámca, ktorý je použitý naším prototypom metódy detekcie zmien v softvérových modeloch (*DiffAddin*).

veného Merkleho stromu.

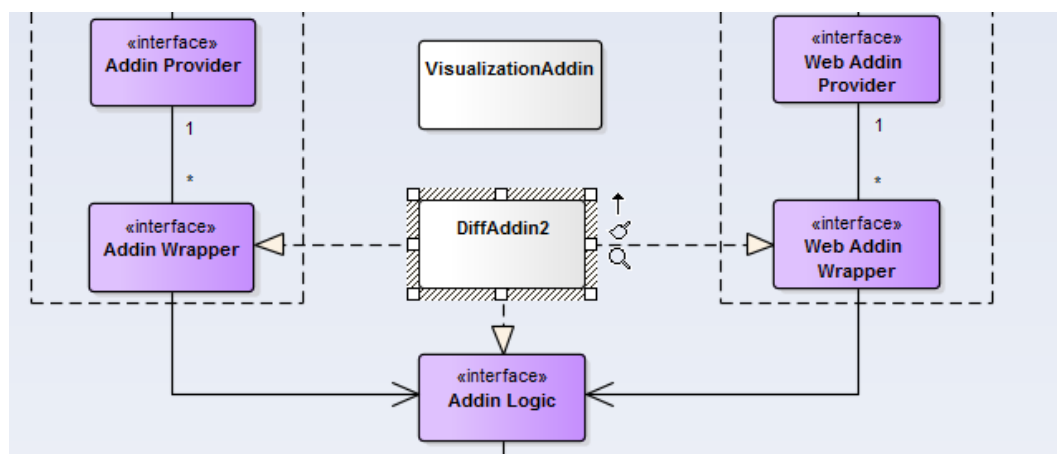
Na obrázku 16 môžeme vidieť architektúru infraštruktúry zameranej na modelovanie softvéru, do ktorej je vsadený nami implementovaný prototyp metódy detekcie zmien v softvérových modeloch *DiffAddin*. Z daného diagramu je možné vidieť, že náš prototyp implementuje rozhranie *Addin Logic*, ktoré slúži ako bod prístupu k ostatným častiam rámca. Taktiež je možné vidieť, že náš prototyp dokáže pracovať aj vo webovom prostredí a teda nie len v prostredí nástroja Sparx Systems Enterprise Architect.

Základnou vizualizáciou, ktorú podporuje prototyp vedľajšej metódy, je vizualizácia zmien vo forme tabuľky (obrázok 17). Táto tabuľka je po vykonaní porovnania verzií modelov zobrazená v okne nástroja Sparx Enterprise Architect určenom pre rozšírenia, pričom farebné rozlíšenie typov zmien korešponduje s farbami použitými metódou vizualizácie zmien v softvérových modeloch. Na príklade je vizualizovaná zmena v názve prvku diagramu, pričom je zobrazená pôvodná hodnota atribútu *name* daného prvku v diagrame –

XMI Path: D:\test dp.xmi			Open XMI	Diff
Type	Previous	Current		
Difference - name	DiffAddin	DiffAddin2		
Addition	-	VisualizationAddin		

SW Modeling Infrastructure	Diff Addin
----------------------------	------------

Obrázok 17: Vizualizácia zmien vo forme tabuľky v nástroji Sparx Enterprise Architect. Prvku s názvom DiffAddin bol zmenený názov na DiffAddin2 a bol pridaný nový prvok s názvom VisualizationAddin.



Obrázok 18: Zvýraznenie prvku, ktorého sa týka zmena, v diagrame.

DiffAddin a aktuálna hodnota DiffAddin2. Samozrejmosťou je použitie štruktúr definovaných pri návrhu metódy detekcie zmien v softvérových modeloch v časti 4.

Po kliknutí na konkrétnu zmenu v okne na obrázku 17 sa zobrazí diagram, v ktorom sa nachádza daný prvok (v prípade existencie viacerých diagramov sa otvorí iba jeden, prípadne ten, ktorý je v nástroji otvorený) a tento prvok sa zvýrazní. Ukážku môžeme vidieť na obrázku 18.

6.2 Vyhodnotenie

Počas overovania nami navrhovanej metódy na detekciu zmien v softvérových modeloch sme sa zamerali najmä na *granularitu* a *správnosť*. To znamená, že sme sledovali či nami navrhovaná metóda dokázala detegovať zmenu na čo

najnižšej granularite, t.j. či dokázala detegovať zmenu na čo najnižšej úrovni prvku modelu (napr. či nevyhlásila zmenu v názve prvku za zmenu celého diagramu) a či zmena, ktorá bola detegovaná, je zmena, ktorá bola reálne vykonaná.

Efektívnosť nami navrhovanej metódy sa nám voči existujúcim riešeniam nepodarilo overiť, nakoľko existujúce analyzované riešenia neposkytujú zdrojový kód. Z tohto dôvodu nebolo možné zmerať a porovnať rýchlosť vykonania ich algoritmov na detekciu zmien s nami navrhovaným algoritmom.

Granularitu a správnosť našej metódy sme overovali na diagramoch aktivít, stavových diagramoch, diagramoch tried a sekvenčných diagramoch. Tieto typy diagramov sme zvolili práve kvôli ich častému použitiu pri modelovaní softvéru (Reggio et al., 2015) a pokrývajú značnú časť všeobecných typov diagramov UML. Na overenie sme použili diagramy poskytnuté spoločnosťou Continental Automotive Systems Slovakia s.r.o. a naše vlastné diagramy.

Na uvedených typoch diagramov sme overovali detekciu pridania, odstránenia, zmeny a presunu jednotlivých prvkov spolu s ich vnorenými prvkami podľa UML metamodelu. To znamená, že sme overovali aj zmeny týkajúce sa, napr. v prípade diagramov tried, aj ich atribútov a operácií. V prípade diagramov aktivít, stavových diagramov a diagramov tried sme nezaznamenali žiadnu nesprávnu alebo chýbajúcu zmenu. Zmeny boli detegované presne v mieste ich vzniku.

V prípade sekvenčných diagramov sme počas overenia zaznamenali nedostatok. Pri operácii presunu správy v rámci sekvenčného diagramu náš algoritmus takúto operáciu vyhodnotil ako kombináciu pridania novej správy a odstránenia pôvodnej správy. Takéto správanie je spôsobené ignorovaním poradia zapísaných prvkov v XMI formáte zodpovedajúcom verzii 2.4 UML – v XMI sú správy zoradené podľa ich časovej nadväznosti, čo náš algoritmus neberie do úvahy. Riešenie tohto problému zostáva predmetom ďalšej práce, pričom riešenie bude pravdepodobne spočívať v predspracovaní obsahu diagramu takéhoto typu a postupnému prideleniu poradového čísla pre vyskytujúce sa správy.

7 Overenie metódy vizualizácie zmien v softvérových modeloch

Overenie metódy vizualizácie zmien v softvérových modeloch sme vykonali prostredníctvom nami implementovaného prototypu, nad ktorým sme previedli používateľské testovanie. Používateľské testovanie prebiehalo formou poskytnutia dvoch verzií softvérových modelov, t.j. pôvodnej verzie a upravenej verzie v prvý deň a pomocou nami implementovanej metódy nasledujúci deň. Pomocou merania času, za ktorý používatelia prehlásili, že identifikovali všetky zmeny sme overili nami zvolenú hypotézu 2. Taktiež sme sledovali úplnosť identifikovaných zmien, t.j. či používatelia identifikovali všetky vykonané zmeny vykonané vo verziách modelu. Týmto sme overili hypotézu 1.

Príklad jednej z dvojíc verzií diagramov použitých pri experimente môžeme vidieť na obrázku 19.

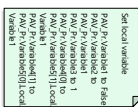
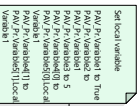
7.1 Prototyp

Prototyp metódy na detekciu zmien v softvérových modeloch sme implementovali ako samostatný nástroj v prostredí Unity¹¹. Tento prototyp však zabezpečuje iba samotnú vizualizáciu zmien podľa nami navrhutej metódy opísanej v kapitole 5. Ako základ nám slúži prototyp opísaný v kapitole 6.1, ktorý poskytuje dáta o zmenách medzi dvomi verziami modelov. Ako základ detekcie zmien v texte sme použili existujúci algoritmus Google Diff-Match-Patch¹².

Nakoľko prototyp na vizualizáciu zmien používa nami implementovaný prototyp na detekciu zmien, taktiež pracuje nad metamodelom UML. V aktuálnom stave náš prototyp podporuje vizualizáciu zmien v diagramoch aktivít a stavových diagramov. Avšak v prípade rozšírenia o nové typy diagramov je nutné iba vytvoriť textúry reprezentujúce nové typy prvkov. V prípade, že ide o prvky so špecifickými vlastnosťami, napr. triedy v diagrame tried, ktoré obsahujú atribúty a metódy, je nutné rozšíriť všeobecný prvok *UmlElement*.

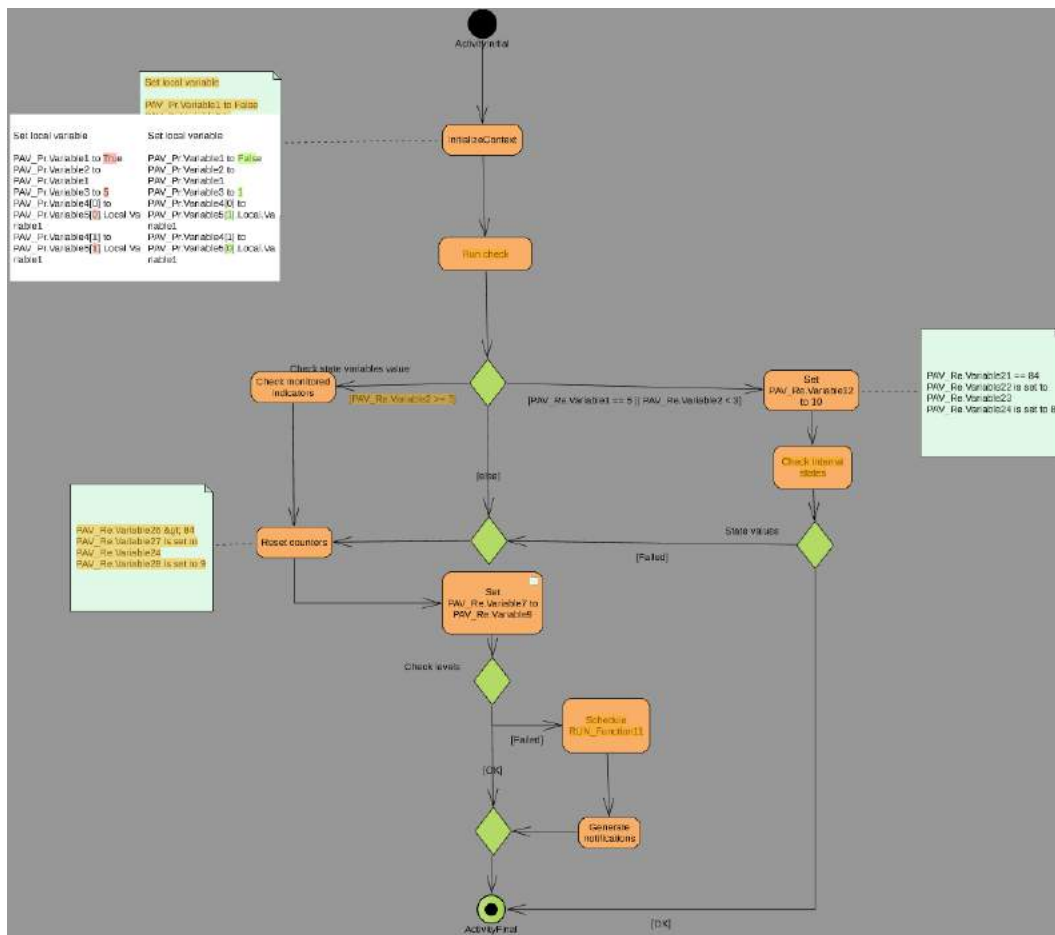
¹¹<https://unity3d.com/>

¹²<https://github.com/google/diff-match-patch>



(b) *Upravená verzia diagramu použitá pri experimente.*

48

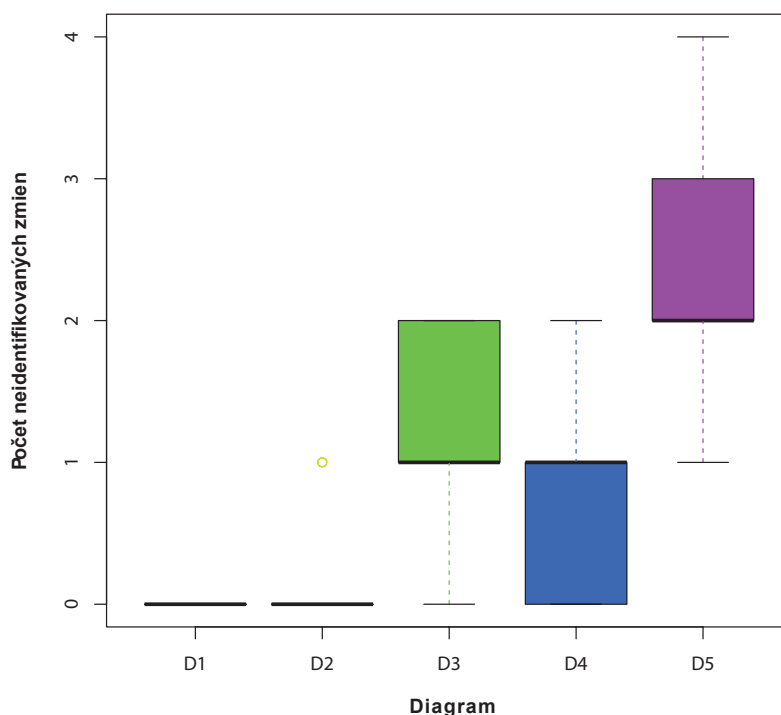


Obrázok 20: Ukážka vizualizácie zmien medzi dvomi verziami diagramu pomocou našej metódy.

7.2 Vyhodnotenie

Opísaný experiment sme vykonali v spolupráci s deviatimi používateľmi, ktorým sme poskytli päť diagramov aktivít, pričom každý diagram mal dve verzie, v ktorých mali identifikovať zmeny. Experiment prebiehal počas dvoch po sebe nasledujúcich dní. Prvý deň prebiehal formou identifikácie zmien na dvoch verziách diagramov zobrazených vedľa seba. Druhý deň bola identifikácia zmien vykonávaná pomocou nami implementovanej metódy, ktorej ukážku je možné vidieť na obrázku 20.

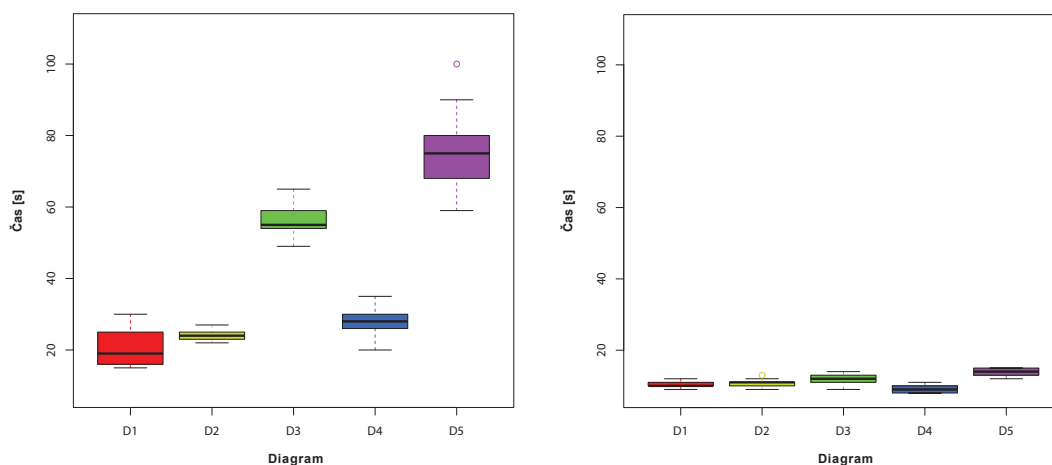
Na overenie hypotézy 1 sme sledovali počet zmien, ktoré používatelia počas experimentu neidentifikovali. Výsledky pre metódu "vedľa seba" môžeme



Obrázok 21: Neidentifikované zmeny v diagramoch pri metóde "vedľa seba".

vidieť na obrázku 21. Výsledky pri použití našej metódy neuvádzame v podobe grafu, nakoľko nedošlo k situácii, kedy by neboli identifikované všetky vykonané zmeny v modeloch. Môžeme teda prehlásiť hypotézu 1 za dokázanú, nakoľko nedošlo k prehliadnutiu akejkoľvek zmeny.

Na overenie hypotézy 2 sme zvolili sledovanie času, za ktorý používatelia prehlásili, že identifikovali všetky zmeny v zobrazenom diagrame aktivít. Výsledky pre metódu identifikácie zmien pomocou metódy dvoch vedľa seba zobrazených verzií modelov môžeme vidieť na obrázku 22a. Výsledky pre našu metódu je možné vidieť na obrázku 22b. Porovnanie mediánov časov je uvedené v tabuľke 1.



(a) Metóde "vedľa seba".

(b) Naša metóda.

Obrázok 22: Časy potrebný na identifikáciu zmien medzi dvoma verziami modelov.

Tabuľka 1: Porovnanie mediánov časov pri použití metódy "vedľa seba" a našej metódy.

Diagram	Metóda "vedľa seba"[s]	Naša metóda [s]	Zlepšenie
D1	19	10	43,36%
D2	24	11	54,16%
D3	55	12	78,18%
D4	28	9	67,86%
D5	75	14	81,33%

Z daných výsledkov môžeme usúdiť, že naša metóda má významný vplyv na efektívnosť práce súvisiacej s identifikáciou zmien v softvérových modeloch. Taktiež môžeme usúdiť, že najväčšie zlepšenie naša metóda priniesla pri diagramoch, v ktorých sa nachádzali poznámky, t. j. text, pričom išlo najmä o diagramy *D4* a *D5*. Hypotézu 2 teda môžeme prehlásiť za dokázanú.

Hypotézu 3 sa nám z časových dôvodov nepodarilo dokázať. Jej dôkaz preto berieme ako predmet ďalšej práce.

8 Zhodnotenie a záver

V práci sme poskytli základný prehľad metód na detekciu a vizualizáciu zmien v softvérových modeloch. Uviedli sme prehľad metód na detekciu zmien nie len pre UML modely, ktorým sa v tejto práci venujeme. Analyzované metódy sme zhodnotili a uviedli ich nedostatky a postrehy, ktoré by bolo vhodné zakomponovať do nami navrhovanej metódy na detekciu zmien. Taktiež sme analyzovali nástroje, metódy a techniky určené na vizualizáciu zmien v UML modeloch. Okrem toho sme analyzovali aj metódy a techniky určené na upriamanie pozornosti, ktoré využívame pri upozornení na konkrétne zmeny a na poskytnutie prehľadu zmien v rozsiahlych softvérových modeloch. K týmto metódam sa v krátkosti vyjadrujeme, uvádzame ich nedostatky, výhody a možnosti ich úpravy a zakomponovania do nami navrhovanej metódy na vizualizáciu zmien v softvérových modeloch.

Následne opisujeme náš návrh metódy na vizualizáciu zmien v softvérových modeloch, pričom sa zameriavame na modely vytvorené pomocou jazyka UML. Uvádzame tri základné pohľady na zmeny, ktoré sme demonštrovali na diagrame tried. Taktiež uvádzame potreby vizualizácie pre jednotlivé kategórie UML diagramov. Ako riešenie prehľadnosti vizualizácie zmien pre modely veľkého rozsahu navrhujeme poskytnutie prehľadu zmien s využitím metódy teplotnej mapy.

Dodatočné vylepšenie vizualizácie, ako predmetu ďalšej práce, sa ukazuje v poskytnutí personalizovanej vizualizácie, ktorej hlavným dôvodom je podpora farboslepých používateľov. Táto metóda by poskytovala preddefinované farebné schémy vhodné pre používateľov trpiacich anomálnou trichromáziou, dichromáziou a monochromáziou. V prípade monochromázie, tzv. úplnej farbosleposti, ktorá zapríčiňuje videnie farieb iba v odtieňoch šedej, by metóda poskytovala rozlišovanie jednotlivých typov zmien pomocou vzorov aplikovaných na odtiene šedej. V ostatných prípadoch farbosleposti by boli poskytnuté predpripravené farebné schémy.

Ďalej sme predstavili návrh vedľajšej všeobecnej metódy určenej na detekciu zmien v softvérových modeloch. Opisujeme nami upravený algoritmus Merkleho stromov, ktorý je pôvodne určený na detekciu integrity veľkých dát.

Naša úprava spočíva v možnosti jednoznačnej identifikácie uzlov v strome, v ktorých sa nachádza zmena alebo predstavujú zmenu. Domnievame sa, že túto metódu je možné použiť na rôzne typy softvérových modelov, ktoré je možné transformovať do stromovej štruktúry. Príkladmi takýchto modelov sú modely UML, na ktoré sa zameriavame, prípadne reprezentácia zdrojového kódu v podobe stromu abstraktnej syntaxe.

Overenie metódy na detekciu zmien sme vykonali pomocou prototypu nad modelmi poskytnutými spoločnosťou Continental Automotive Systems Slovakia s.r.o. a nad vlastnými modelmi. Sledovali sme granularitu a správnosť detekcie zmien, t.j. či boli zmeny detegované priamo v mieste ich vzniku a nie v rodičovských prvkoch. Na overenie sme použili najčastejšie používané typy diagramov, a to diagramy aktivít, stavové diagramy, diagramy tried a sekvenčné diagramy. V prípade sekvenčných diagramov sme zaznamenali nedostatok v detekcii presunov jednotlivých správ, ktorý vychádza z povahy reprezentácie správ v XMI formáte. Ako predmet ďalšej práce môže byť práve odstránenie tohto nedostatku úpravou nami navrhovanej metódy o predspracovanie správ v sekvenčných diagramoch a pridelenia im jednoznačného poradia.

Metódu na vizualizáciu zmien sme vykonali prostredníctvom používateľského testovania, pričom sme používateľom poskytli pohľad na zmeny metódou "vedľa seba", t.j. poskytli sme pôvodnú a upravenú verziu modelov a taktiež pohľad na zmeny pomocou našej metódy. Počas experimentu sme merali čas, za ktorý používatelia prehlásili, že identifikovali všetky zmeny a taktiež sme sledovali počet neidentifikovaných zmien. Po porovnaní výsledkov použitím metódy "vedľa seba" a našej metódy môžeme vyhlásiť, že naša metóda dosiahla významne lepšie výsledky. Taktiež sa nám podarilo dokázať hypotézy 1 a 2. Hypotéza 3 zostala z časových dôvodov nedokázaná a teda bude predmetom ďalšej práce.

Overenie hypotézy 3 plánujeme vykonať prostredníctvom prípadovej štúdie vykonanej v spolupráci so spoločnosťou Continental Automotive Systems Slovakia s.r.o. a jej tímom testerov. V súčasnom stave testerí identifikujú zmenené časti modelov okometricky na základe čoho upravujú vytvorené testy tak, aby vyhovovali upraveným modelom. Tento fakt sa týka aj analytikov, ktorí diskutujú o prevedených zmenách kvôli dopadu zmien na bezpečnosť vyvíjaného softvéru. Prípadová štúdia bude zahŕňať kvalitatívny dotazník, ktorý

by spomínané tímy vyplnili. Na základe zozbieraných odpovedí chceme sledovať vplyv našej metódy na postupnú diskusiu v rámci tímov a jej vplyv na rozhodnutia. Taktiež budeme vyhodnocovať časy, ktoré strávili tester pri implementácii testov k zmene v návrhu pred a po začlenení navrhovanej metódy do vývojového procesu. Tieto časy získame z nástroja pre manažment úloh, pričom ich budeme normalizovať vzhľadom na množstvo kódu zmeneného pri týchto úlohách.

Literatúra

- [Bailey et al., 2009] Bailey, R.; McNamara, A.; Sudarsanam, N.; et al.: Subtle gaze direction. *ACM Transactions on Graphics*, ročník 28, č. 4, 2009: s. 1–14, ISSN 07300301.
- [Benomar et al., 2013] Benomar, O.; Sahraoui, H.; Poulin, P.: Visualizing software dynamicities with heat maps. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [Bojko, 2009] Bojko, A.: Informative or misleading? Heatmaps deconstructed. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, ročník 5610 LNCS, č. PART 1, 2009: s. 30–39, ISSN 03029743.
- [Burmester et al., 2004] Burmester, S.; Giese, H.; Niere, J.; et al.: Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer*, ročník 6, č. 3, 2004: s. 203–218, ISSN 1433-2779.
- [Cockburn et al., 2008] Cockburn, A.; Karlson, A.; Bederson, B. B.: A review of overview+detail, zooming, and focus+context interfaces. ročník 41, č. 1, 2008: s. 2:1–2:31, ISSN 0360-0300.
- [Cole et al., 2006] Cole, F.; DeCarlo, D.; Finkelstein, A.; et al.: Directing Gaze in 3D Models with Stylized Focus. *Eurographics Symposium on Rendering*, 2006.
- [Costa et al., 2014] Costa, V.; Monteiro, R.; Murta, L.: Detecting Semantic Equivalence in UML Class Diagrams. *The 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2014: s. 318–323, ISSN 23259086.
- [Deb et al., 2002] Deb, K.; Pratap, A.; Agarwal, S.; et al.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, ročník 6, č. 2, 2002: s. 182–197, ISSN 1089778X.
- [Gerth a Engels, 2008] Gerth, C.; Engels, G.: Detecting and Resolving Process Model Differences in. *Differences*, 2008.
- [Girschick a Darmstadt, 2006] Girschick, M.; Darmstadt, T.: Difference detection and visualization in UML class diagrams. *Technical University of*

- Darmstadt Technical Report TUD-CS-2006-5*, 2006.
- [Godfrey et al., 2015] Godfrey, P.; Gryz, J.; Lasek, P.: Interactive Visualization of Large Data Sets Technical Report EECS-2015-03. *IEEE Transactions on Knowledge and Data Engineering*, ročník 28, č. 8, 2015: s. 2142–2157, ISSN 10414347.
- [Hauser, 2005] Hauser, H.: Generalizing focus+ context visualization. *Scientific Visualization: The Visual Extraction of Knowledge from Data*, 2005.
- [Itti a Koch, 2000] Itti, L.; Koch, C.: A saliency-based search mechanism for overt and covert shifts of visual attention. *Vision Research*, ročník 40, č. 10-12, 2000: s. 1489–1506, ISSN 00426989.
- [Judd et al., 2009] Judd, T.; Ehinger, K.; Durand, F.; et al.: Learning to predict where humans look. In *2009 IEEE 12th International Conference on Computer Vision, Iccv*, IEEE, 9 2009, ISBN 978-1-4244-4420-5, s. 2106–2113.
- [Kelter et al., 1999] Kelter, U.; Monecke, M.; Platz, D.: Constructing Distributed SDEs using an Active Repository. *Proc. 1st Intl. Symposium on Constructing Software Engineering Tools*, 1999.
- [Kessentini et al., 2016] Kessentini, M.; Mansoor, U.; Wimmer, M.; et al.: Search-based detection of model level changes. *Empirical Software Engineering*, 2016: s. 1–46, ISSN 15737616.
- [Kister et al., 2016] Kister, U.; Reipschläger, P.; Dachsel, R.: MultiLens: fluent interaction with multi-functional multi-touch lenses for information visualization. *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces - ISS '16*, 2016.
- [Koehler et al., 2014] Koehler, K.; Guo, F.; Zhang, S.; et al.: What do saliency models predict? *Journal of Vision*, ročník 14, č. 3, 2014: s. 1–27, ISSN 1534-7362.
- [Kosara et al., 2001] Kosara, R.; Miksch, S.; Hauser, H.: Semantic depth of field. *Proc. InfoVis*, ročník 2001, 2001: str. 97, ISSN 1522-404X.
- [Maurus et al., 2014] Maurus, M.; Hammer, J. H.; Beyerer, J.: Realistic Heatmap Visualization for Interactive Analysis of 3D Gaze Data. *Proceedings of the Symposium on Eye Tracking Research and Applications. Symposium on Eye Tracking Research & Applications (ETRA-14), March 26-28, Safety Harbor,, FL, USA*, 2014.

- [Melloni et al., 2012] Melloni, L.; Van Leeuwen, S.; Alink, A.; et al.: Interaction between bottom-up saliency and top-down control: How saliency maps are created in the human brain. *Cerebral Cortex*, ročník 22, č. 12, 2012: s. 2943–2952, ISSN 10473211.
- [Merkle, 1988] Merkle, R. C.: A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87: Proceedings*, editácia C. Pomerance, Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, ISBN 978-3-540-48184-3, s. 369–378.
- Dostupné na internete: https://doi.org/10.1007/3-540-48184-2_32
- [Niere, 2004] Niere, J.: Visualizing differences of UML diagrams with Fujaba. *Proceedings of the 2nd International Fujaba Days 2004*, 2004.
- [Ohst et al., 2003] Ohst, D.; Welle, M.; Kelter, U.: Differences between versions of UML diagrams. *ACM SIGSOFT Software Engineering Notes*, ročník 28, č. 5, 2003: str. 227, ISSN 01635948.
- [Ondik et al., 2017] Ondik, J.; Olejár, M.; Rástočný, K.; et al.: Activity-Based Model Synchronization and Defects Detection for Small Teams. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017, Prague, Czech Republic, July 25-29, 2017*, IEEE, 2017, ISBN 978-1-5386-2072-4, s. 8–15.
- [Reggio et al., 2015] Reggio, G.; Leotta, M.; Ricca, F.; et al.: What Are the Used UML Diagram Constructs? A Document and Tool Analysis Study Covering Activity and Use Case Diagrams. In *Model-Driven Engineering and Software Development*, editácia S. Hammoudi; L. F. Pires; J. Filipe; R. C. das Neves, Cham: Springer International Publishing, 2015, ISBN 978-3-319-25156-1, s. 66–83.
- [Röthlisberger et al., 2009] Röthlisberger, D.; Nierstrasz, O.; Ducasse, S.; et al.: Supporting task-oriented navigation in IDEs with configurable Heat-Maps. *IEEE International Conference on Program Comprehension*, 2009: s. 253–257, ISSN 1092-8138.
- [Tominski et al., 2006] Tominski, C.; Abello, J.; Van Ham, F.; et al.: Fisheye tree views and lenses for graph visualization. *Proceedings of the International Conference on Information Visualisation*, 2006: s. 17–24, ISSN 10939547.
- [Torralba et al., 2006] Torralba, A.; Oliva, A.; Castelhana, M. S.; et al.: Con-

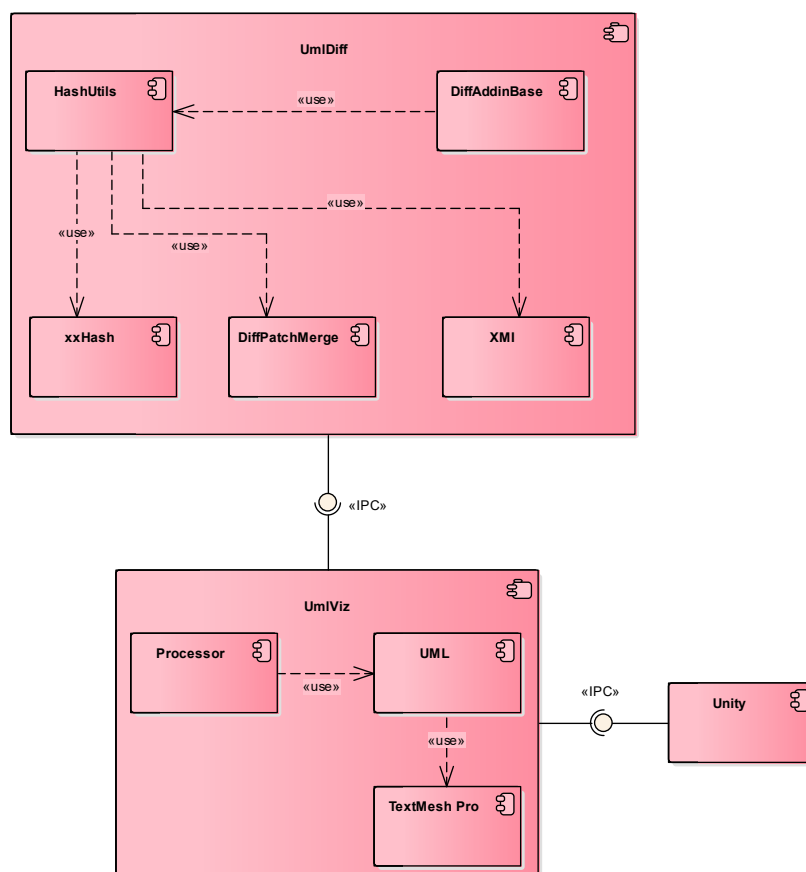
- textual guidance of eye movements and attention in real-world scenes: the role of global features in object search. *Psychological Review*, ročník 113, č. 4, 2006: s. 766–786, ISSN 0033-295X.
- [Vanhatalo et al., 2007] Vanhatalo, J.; Völzer, H.; Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *Service-Oriented Computing – ICSOC 2007*, ročník 4749, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN 978-3-540-74974-5, s. 43–55, 0956814.
- [Wong et al., 2003] Wong, N.; Carpendale, S.; Greenberg, S.: EdgeLens: An interactive method for managing edge congestion in graphs. *Proceedings - IEEE Symposium on Information Visualization, INFO VIS*, ročník 2003, 2003: s. 51–58, ISSN 1522404X.
- [Xing a Stroulia, 2005] Xing, Z.; Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, 2005.
- [Xu et al., 2014] Xu, J.; Jiang, M.; Wang, S.; et al.: Predicting human gaze beyond pixels. *Journal of vision*, ročník 14, č. 1, 2014: str. 28, ISSN 1534-7362.

Príloha A: Technická dokumentácia

Architektúra nášho prototypu je zobrazená na obrázku 23 a pozostáva z dvoch hlavných komponentov:

- *UmlDiff*, ktorý zabezpečuje detekciu zmien medzi verziami modelov a
- *UmlViz*, ktorý zabezpečuje vizualizáciu zmien.

Komponent *Unity*¹³ predstavuje infraštruktúru na poskytnutie vizualizačných funkcií, ktoré používa komponent *UmlViz* a ďalej ho neopisujeme.



Obrázok 23: Architektúra implementácie.

¹³<https://unity3d.com/>

Komponent *UmlDiff* pozostáva z 5 komponentov:

- *DiffAddinBase*, ktorý predstavuje komponent poskytujúci rozhrania metód určených na detekciu zmien a načítanie modelov,
- *HashUtil*, ktorý predstavuje komponent, ktorý zabezpečuje konverziu modelov z XMI formátu na upravené Merkleho stromy a stratégiu výpočtu hodnôt rozptylovej funkcie pre jednotlivé UML prvky,
- *xxHash*¹⁴, ktorý predstavuje externý komponent, ktorý poskytuje rozhranie pre rozptylovú funkciu xxHash,
- *DiffPatchMerge*, ktorý predstavuje komponent zodpovedný za výpočet rozdielu textov,
- *XMI*, ktorý predstavuje komponent obsahujúci schému pre všetky UML prvky.

Komponent *UmlViz* pozostáva z 3 komponentov:

- *Processor*, ktorý poskytuje logiku na poskytnutie modelov komponentu *UmlDiff* a následné prebratie zmien, vytvorenie jednotlivých UML prvkov a spustenie vykresľovania,
- *UML*, ktorý predstavuje komponent, ktorý poskytuje stratégiu vykresľovania jednotlivých UML komponentov,
- *TextMesh Pro*¹⁵, ktorý predstavuje komponent na vykresľovanie textu a manipuláciu s ním.

Komponenty *UmlDiff* a *UmlViz* medzi sebou komunikujú prostredníctvom medziprocesnej komunikácie (angl. *Inter-process communication*). Rovnako sú na tom aj komponent *UmlViz* a externý komponent *Unity*.

¹⁴<http://cyan4973.github.io/xxHash/>

¹⁵<https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126>

Príloha B: Inštalačná príručka

B1: Inštalácia rozšírenia

Pre inštaláciu a používanie nášho rozšírenia je potrebné mať nainštalované:

- operačný systém Windows vo verzii 7 alebo vyššej,
- .NET Framework vo verzii 4.5.2 alebo vyššej,
- Sparx Systems Enterprise Architect vo verzii 12,
- miesto na disku s veľkosťou aspoň 5 MB.

Inštalácia infraštruktúry rozšírenia prebieha pravým kliknutím na inštalačný súbor (obrázok 24) *Installer.msi* zo zložky *installer/* na priloženom elektronickom médiu a kliknutí na možnosť *Inštalovať* (1), čím sa rozšírenie automaticky nainštaluje do zložky *%ProgramFiles%\SWModelingInfrastructure*.



Obrázok 24: Kontextové menu pre inštalátor

V prípade problémov je možné rozšírenie preinštalovať kliknutím na možnosť *Opraviť* (2) a odstrániť kliknutím na možnosť *Odinštalovať* (3).

Po úspešnej inštalácii infraštruktúry rozšírenia je možné nainštalovať samotné rozšírenie. Inštalácia rozšírenia prebieha rozbalením archívu *UMLDiff.zip* zo zložky *installer/* na priloženom elektronickom médiu do zložky *%AppData%\SWModelingInfrastructure\addins*. Po úspešnom rozbalení archívu je možné k rozšíreniu prísť prostredníctvom nástroja Sparx Systems Enterprise Architect.

B2: Inštalácia vizualizačného nástroja

Inštalácia rozšírenia prebieha rozbalením archívu *UmlViz.zip* zo zložky *installer/* na priloženom elektronickom médiu do ľubovoľnej zložky na disku. Po

úspešnom rozbalení archívu je možné vizualizačný nástroj spustiť prostredníctvom súboru UmlViz.exe.

Príloha C: Používateľská príručka

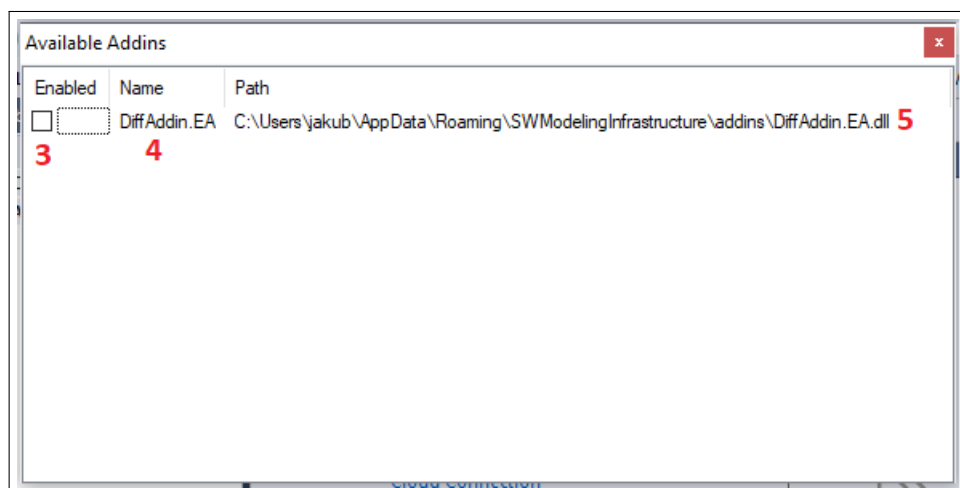
C1: Rozšírenie UmlDiff

Po spustení nástroja Sparx Systems Enterprise Architect sa rozšírenie načíta automaticky. Základná ponuka, ku ktorej sa pristupuje kliknutím na *EXTENSIONS* v hornom menu Enterprise Architect, je zobrazená na obrázku 25.



Obrázok 25: Ponuka rozšírenia.

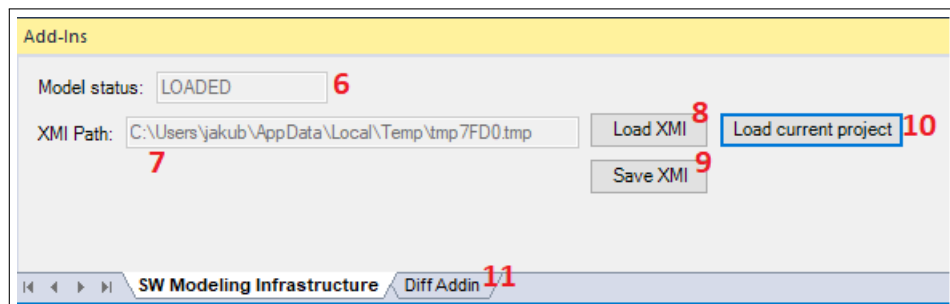
Možnosť *Load* (1) predstavuje funkciu pre znovu-načítanie celého rozšírenia. Po zvolení možnosti *Addins* (2) sa zobrazí okno na obrázku 26 s dostupnými rozšíreniami.



Obrázok 26: Ponuka dostupných rozšírení.

Toto okno ponúka možnosti na povolenie rozšírenia (3), informáciu o názve rozšírenia (4) a cestu k rozšíreniu (5).

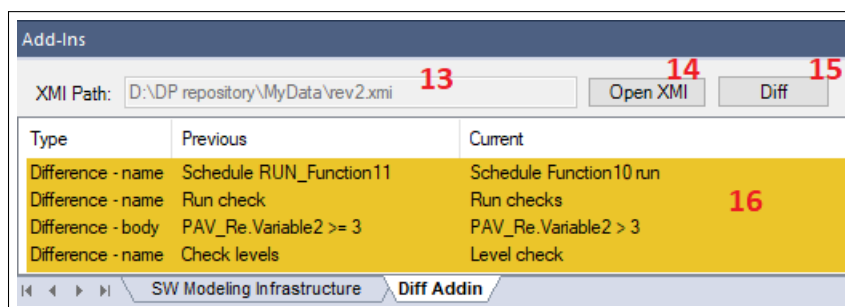
Aktuálne povolené rozšírenia vrátane infraštruktúrneho rozšírenia je možné nájsť v okne nástroja Sparx Systems Enterprise Architect určeného pre rozšírenia, ktoré je možné vidieť na obrázku 27.



Obrázok 27: Okno pre rozšírenia.

Na obrázku 27 je zobrazená časť okna pre infraštruktúru rozšírení. V tomto okne sa nachádzajú informácie o aktuálnom stave modelu (6), ceste k modelu (7) a možnosti na načítanie modelu z XMI – *Load XMI* (8), na uloženie aktuálne načítaného modelu do formátu XMI – *Save XMI* (9) a na načítanie aktuálne otvoreného modelu – *Load current project* (10).

V dolnej časti okna sa nachádzajú záložky pre jednotlivé rozšírenia, v tomto prípade iba pre rozšírenie *UmlDiff* – *Diff Addin* (11), pričom po kliknutí na túto záložku sa zobrazí okno pre toto rozšírenie (obrázok 28).



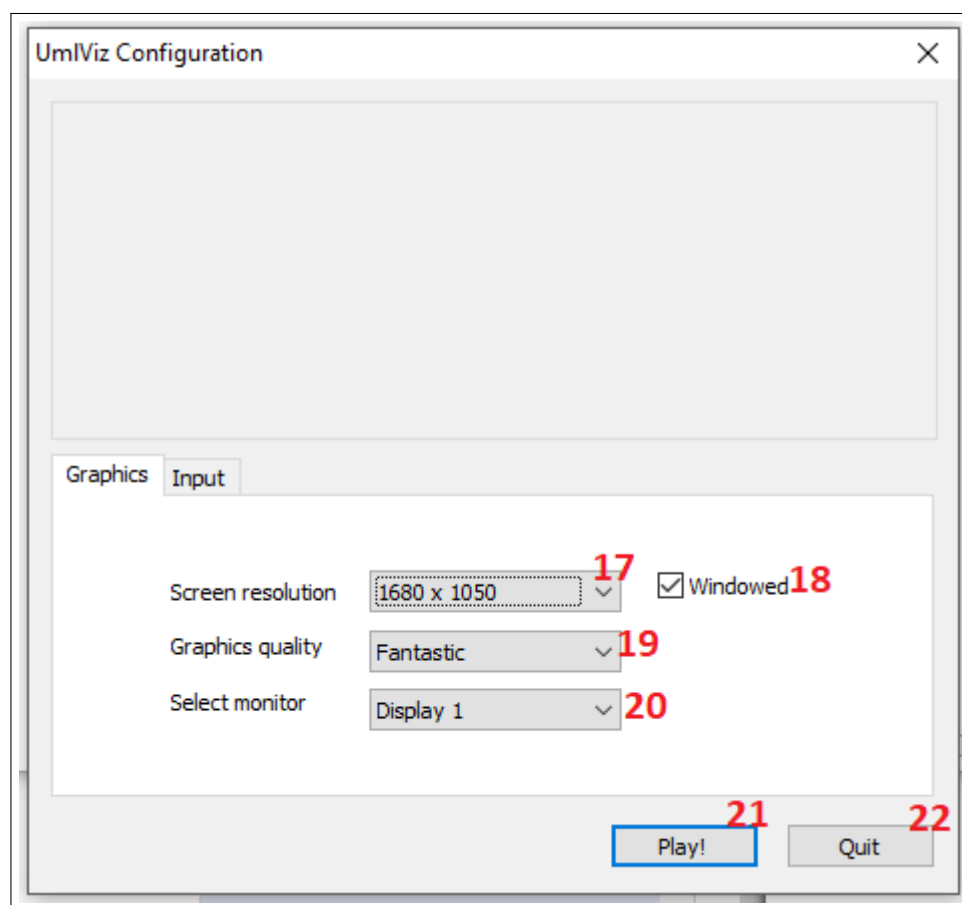
Obrázok 28: Okno pre rozšírenie UmlDiff.

V tomto okne je informácia o načítanom modeli z XMI formátu (13), pričom načítanie je možné vykonať kliknutím na možnosť *Open XMI* (14). V tomto

okne sa nachádza aj možnosť na vykonanie porovnania verzií modelov *Diff* (15). Výsledok porovnania môžeme vidieť v časti 16.

C2: Vizualizačný nástroj UmlViz

Po spustení nástroja *UmlViz* sa zobrazia možné nastavenia, čo môžeme vidieť na obrázku 29.

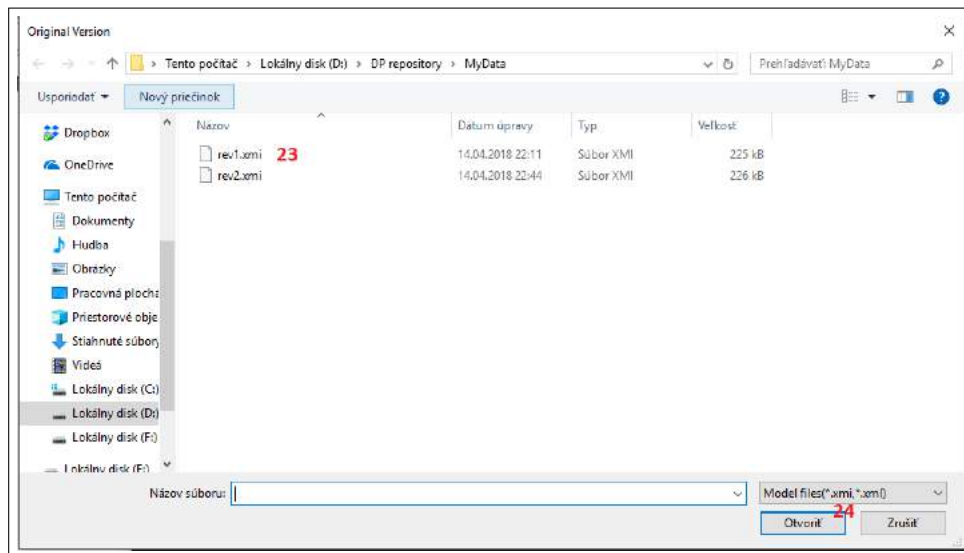


Obrázok 29: Nastavenie nástroja.

K možným nastaveniam patrí rozlíšenie (17), mód zobrazenia (18), úroveň grafickej kvality (19), výber zariadenia, na ktorom sa nástroj zobrazí (20), spustenie samotného nástroja (21) a ukončenie práce (22).

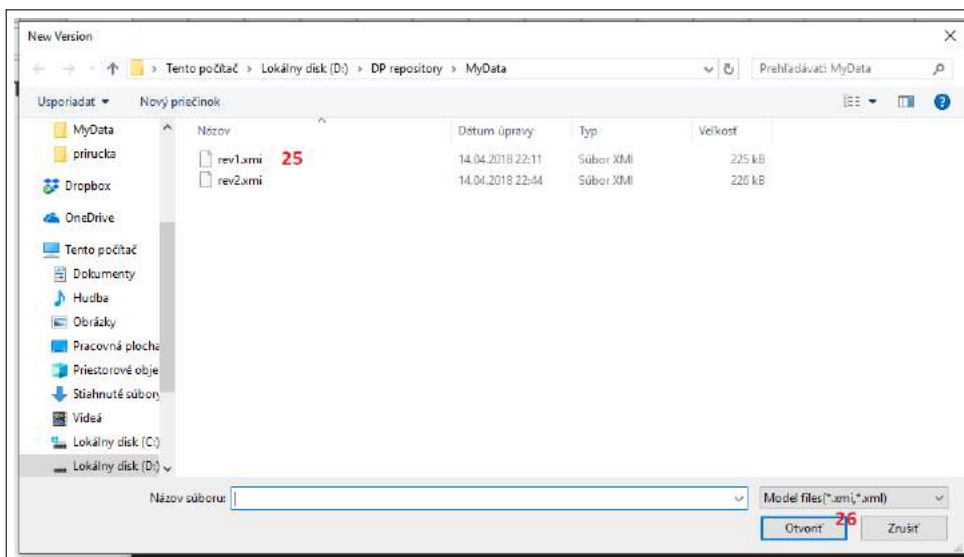
Po spustení nástroja kliknutím na možnosť *Play!* (21) sa zobrazí dialógové okno určené na výber XMI súboru s pôvodnou verziou modelu, čo môžeme

vidieť na obrázku 30.



Obrázok 30: Výber pôvodnej verzie modelu.

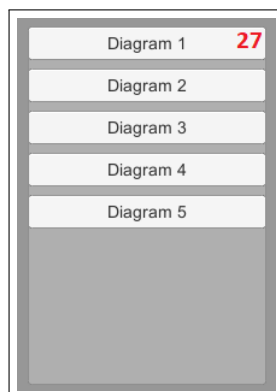
Po vybratí súboru (23) a potvrdení (24) sa analogicky zobrazí okno na načítanie upravenej verzie modelu (obrázok 31).



Obrázok 31: Výber upravenej verzie modelu.

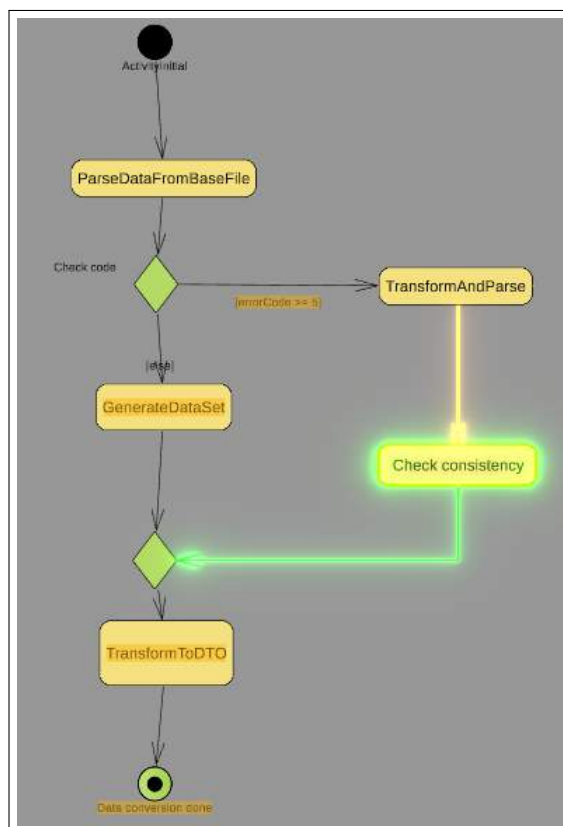
Po vybratí súboru (25) s upravenou verziou modelu a potvrdení (26) sa zobrazí okno so zoznamom zmenených diagramov v rámci modelu, ktoré môžeme vidieť

na obrázku 32.



Obrázok 32: Zoznam upravených diagramov.

Po vybratí diagramu kliknutím na jeho názov (27) sa zobrazí okno so samotnou vizualizáciou zmien medzi vybratými verziami modelu, čo môžeme vidieť na obrázku 33.

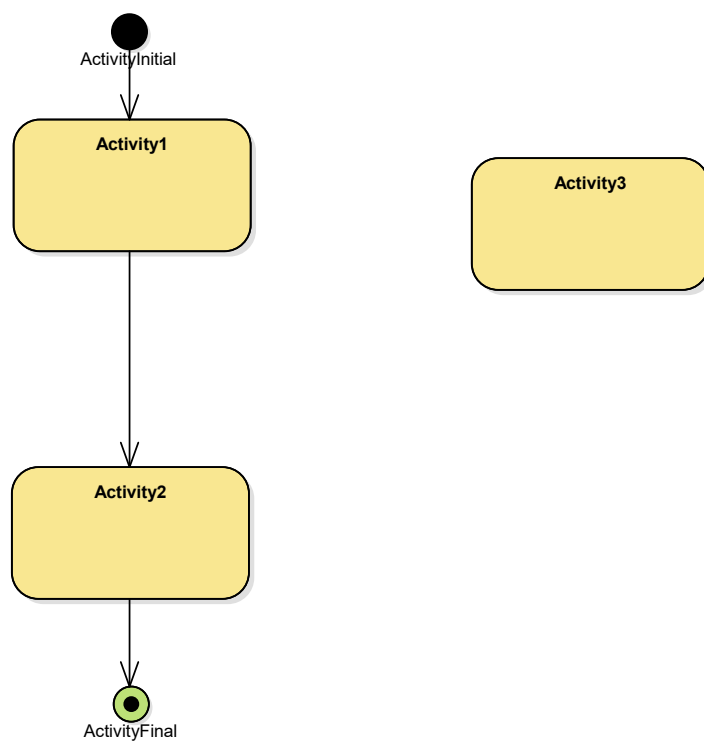


Obrázok 33: Vizualizácia zmien medzi verziami modelu.

Príloha D: Použité modely

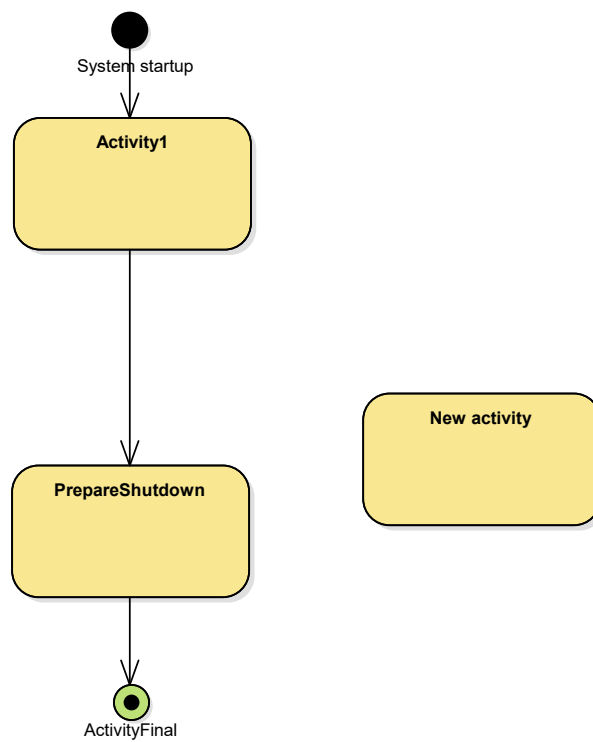
D1: Diagram 1

Pôvodná verzia



Obrázok 34: *Pôvodná verzia diagramu 1.*

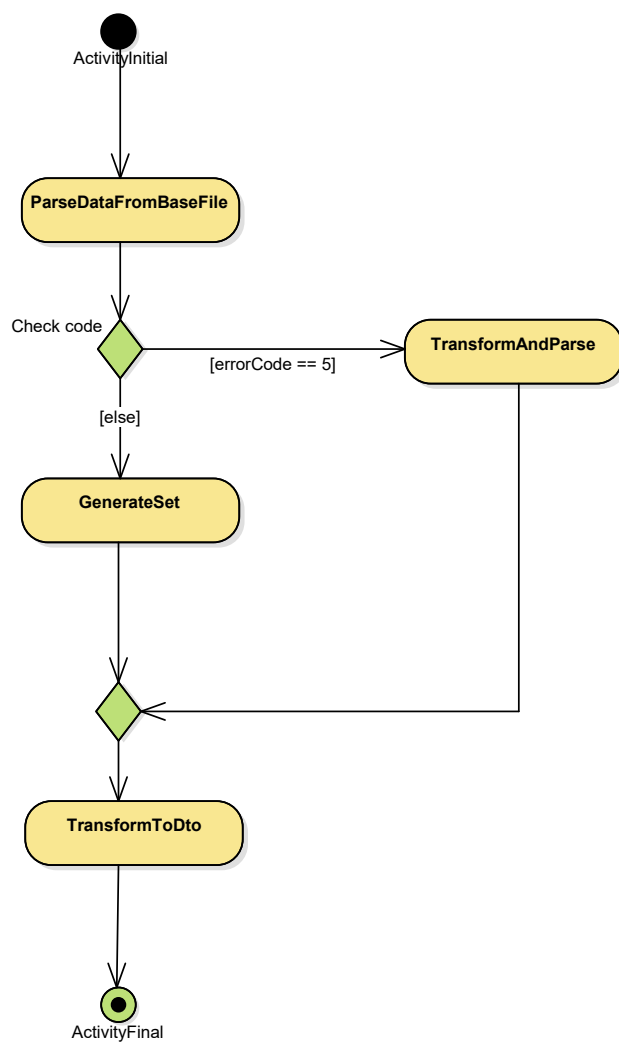
Upravená verzia



Obrázok 35: *Upravená verzia diagramu 1.*

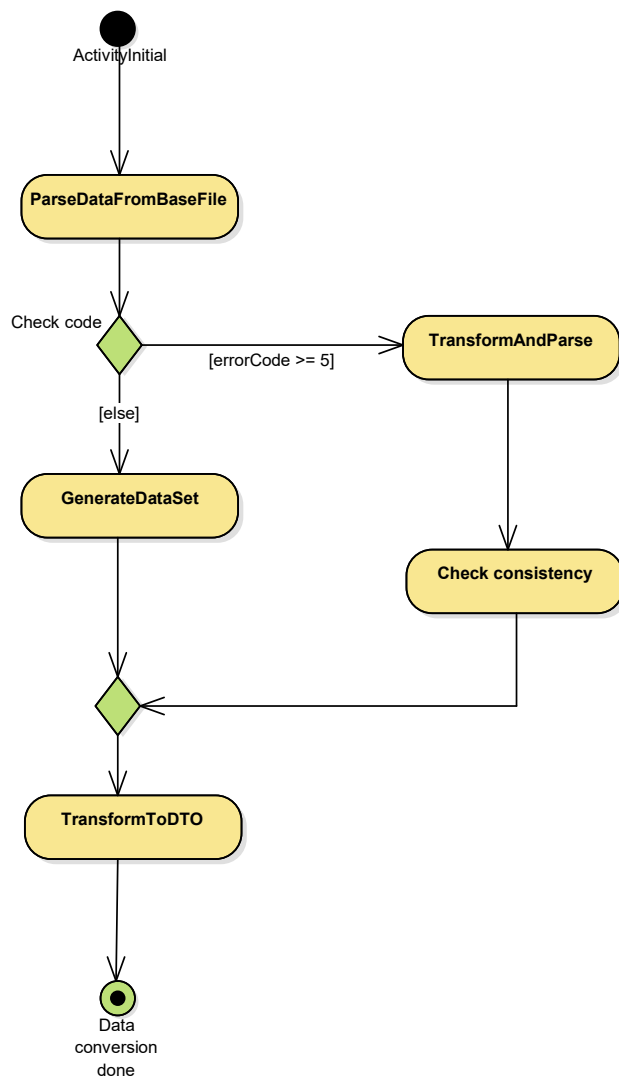
D2: Diagram 2

Pôvodná verzia



Obrázok 36: Pôvodná verzia diagramu 2.

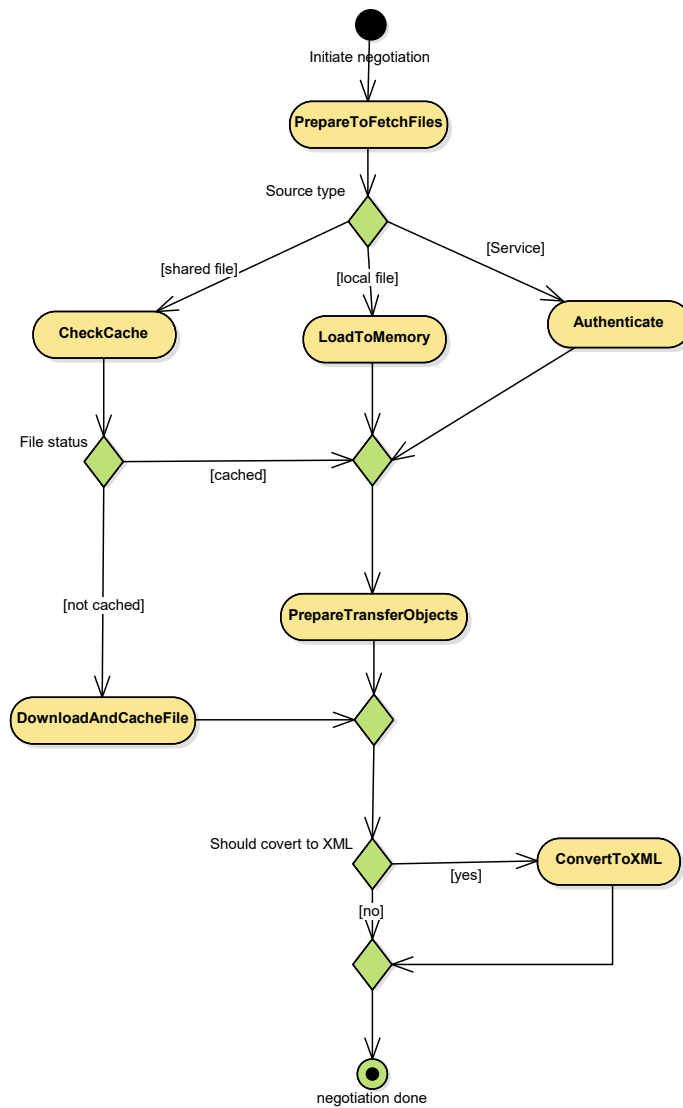
Upravená verzia



Obrázok 37: *Upravená verzia diagramu 2.*

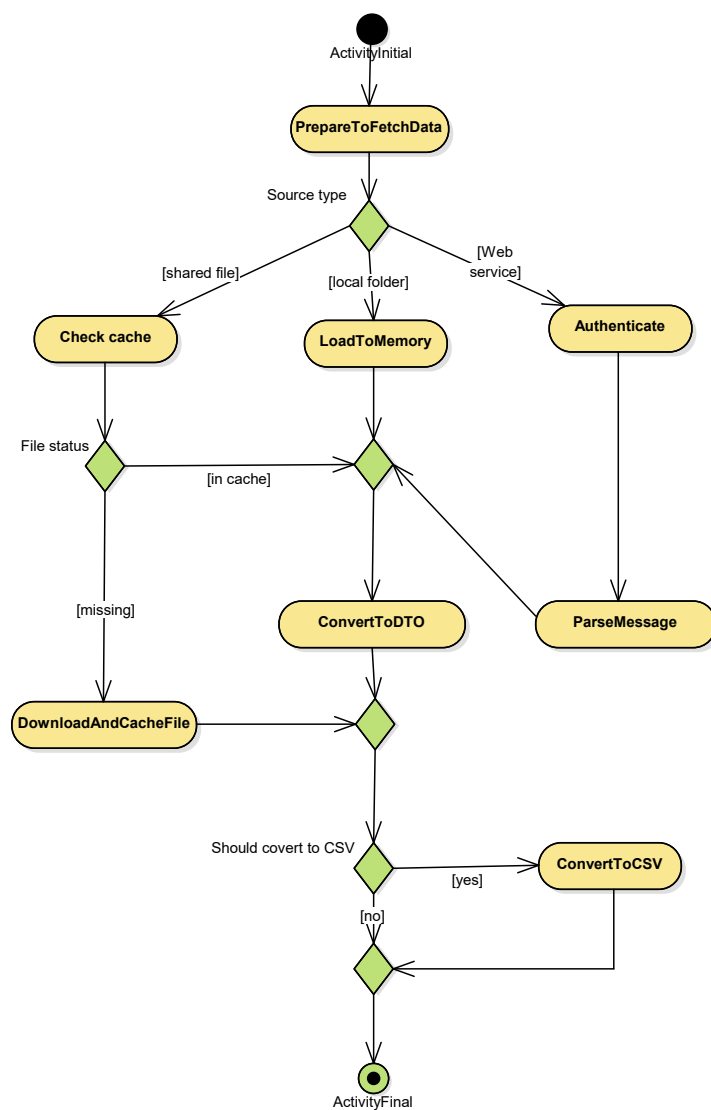
D3: Diagram 3

Pôvodná verzia



Obrázok 38: Pôvodná verzia diagramu 3.

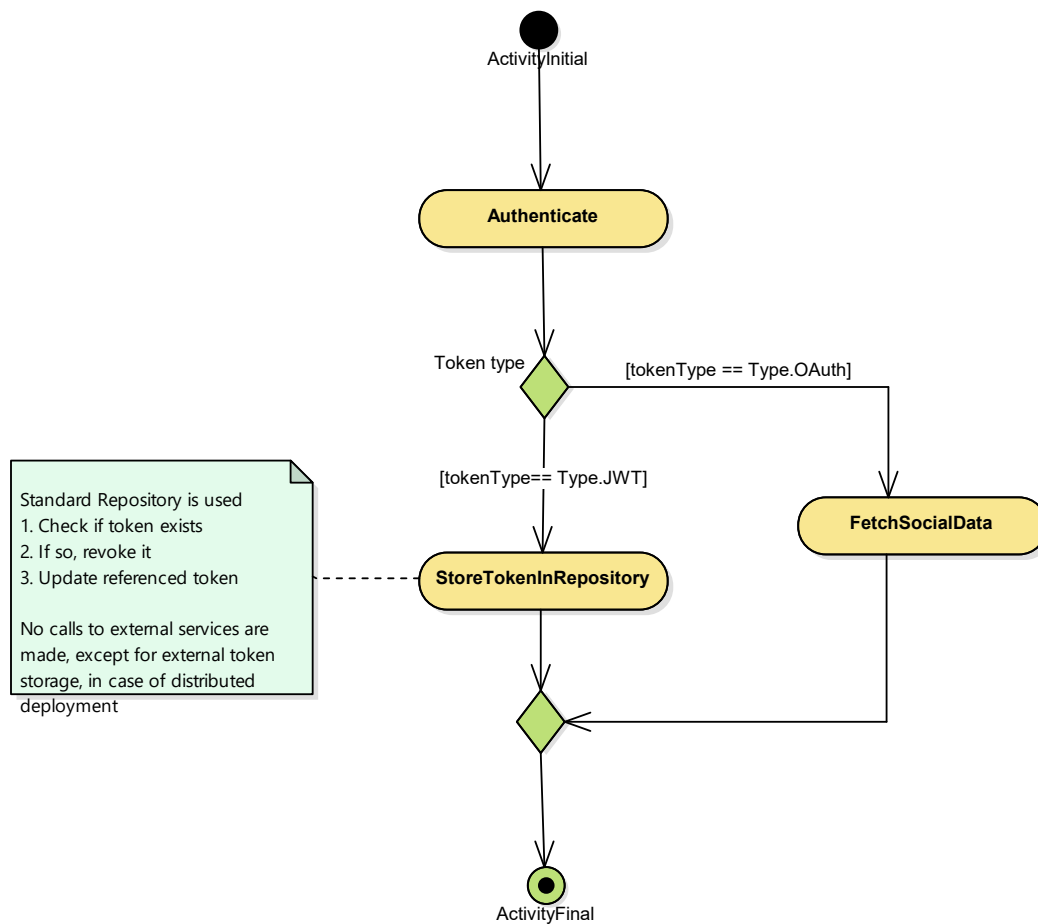
Upravená verzia



Obrázok 39: Upravená verzia diagramu 3.

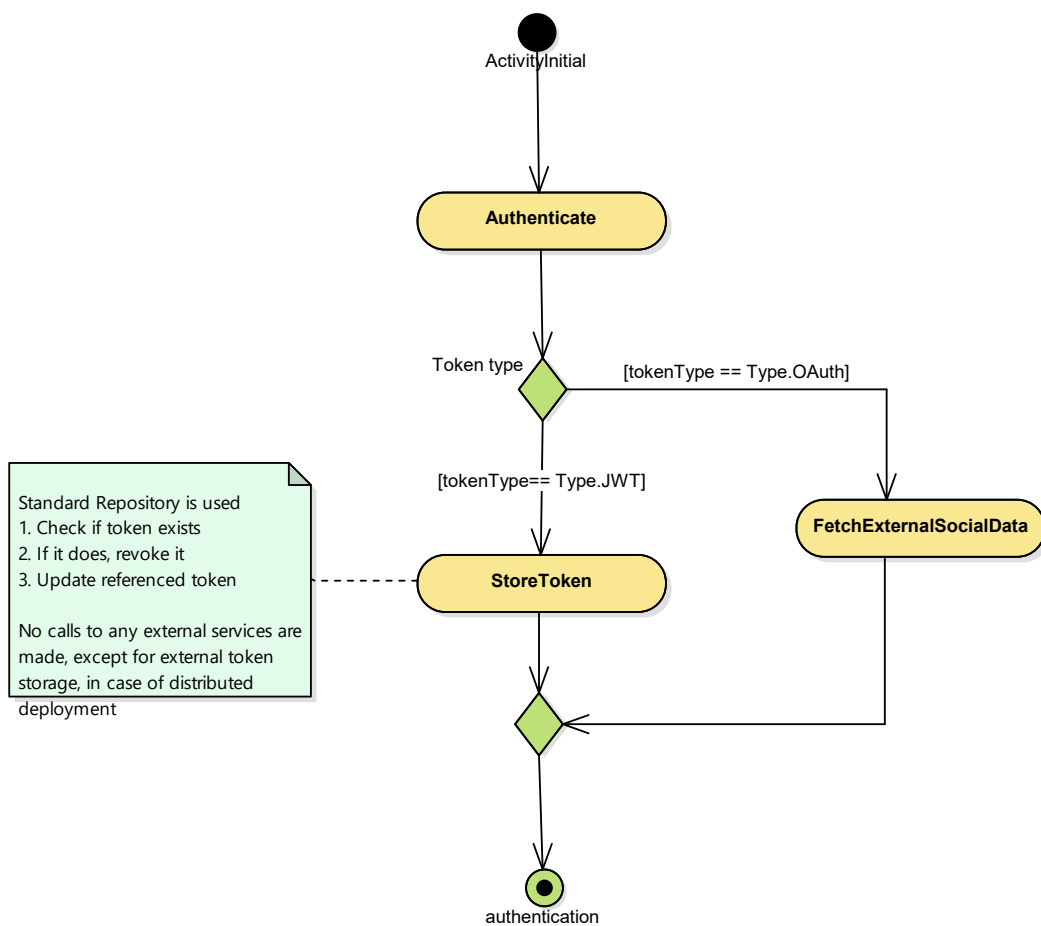
D4: Diagram 4

Pôvodná verzia



Obrázok 40: Pôvodná verzia diagramu 4.

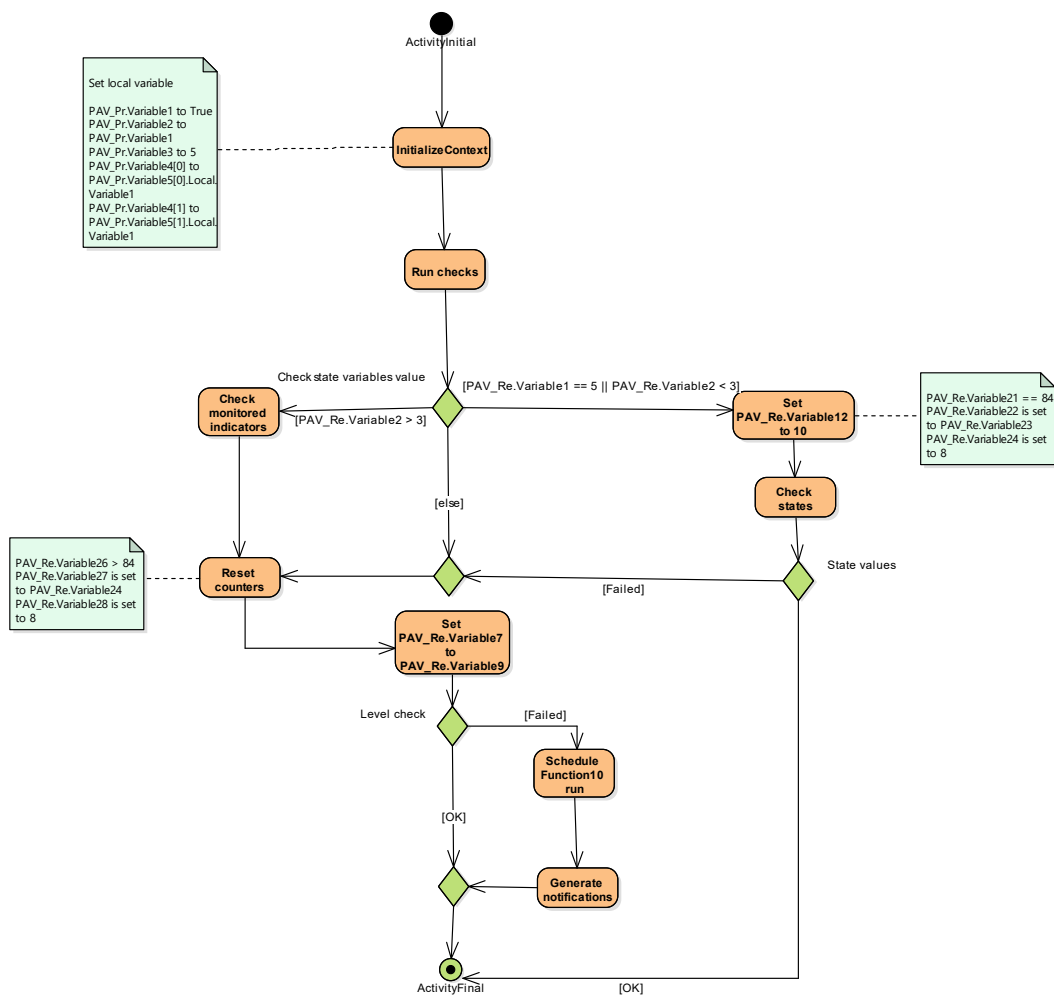
Upravená verzia



Obrázok 41: Upravená verzia diagramu 4.

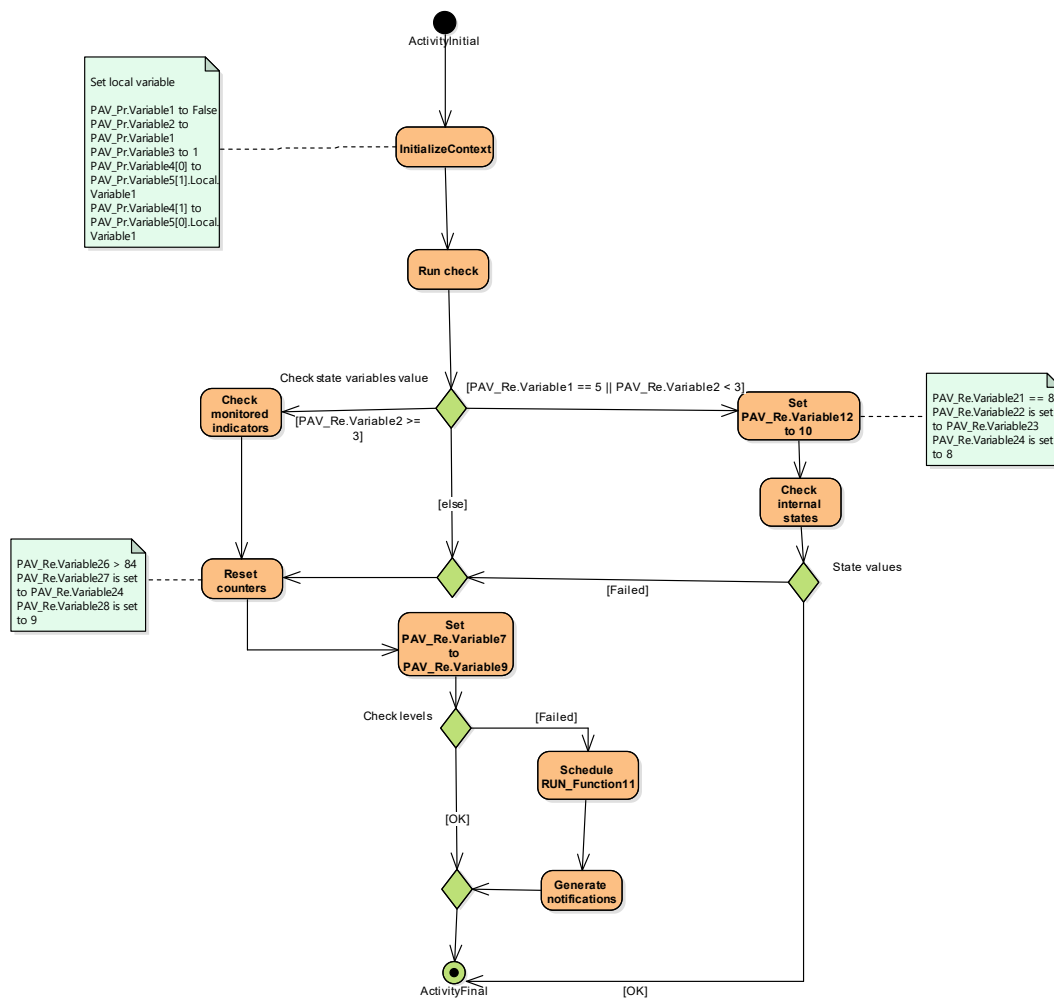
D5: Diagram 5

Pôvodná verzia



Obrázok 42: Pôvodná verzia diagramu 5.

Upravená verzia



Obrázok 43: Upravená verzia diagramu 5.

Príloha E: Plán práce

E1: Plán práce po prvej priebežnej správe

Tabuľka 2: *Plán práce v zimnom semestri*

Úloha	Termín
dokončenie návrhu metódy na vizualizáciu zmien	2. týždeň
prvotný návrh metódy na detekciu zmien	5. týždeň
implementácia prototypu vizualizácie zmien	9. týždeň
dokončenie návrhu metódy na detekciu zmien	11. týždeň
formálne úpravy práce a tvorba prezentácie na obhajobu	12. týždeň

Tabuľka 3: *Plán práce v letnom semestri*

Úloha	Termín
zpracovanie pripomienok z posudku a obhajoby	1. týždeň
podrobný návrh A/B testovania	2. týždeň
vykonanie A/B testovania	3. týždeň
spracovanie výsledkov z A/B testovania	3. týždeň
podrobný návrh používateľského testovania	4. týždeň
vykonanie používateľského testovania	5. týždeň
spracovanie výsledkov z A/B testovania	6. týždeň
implementácia prototypu detekcie zmien	8. týždeň
overenie prototypu detekcie zmien	9. týždeň
dokončenie hlavnej textovej časti práce	10. týždeň
spísanie príloh a formálne úpravy	11. týždeň
tvorba prezentácie a príprava na obhajobu	12. týždeň

E2: Zhodnotenie zimného semestra

Úlohy určené v pláne na zimný semester (tabuľka 2) sa nám podarilo splniť úplne, až na úlohu *implementácia prototypu vizualizácie zmien*. Túto úlohu sa nám podarilo dokončiť len na úrovni vizualizácie prvkov validných pre diagram aktivít bez ich vizualizácie zmien. Podarilo sa nám však implementovať

prototyp detekcie zmien, ktorého implementácia bola pôvodne plánovaná až v letnom semestri.

E3: Plán práce po druhej priebežnej správe

V pláne bol vykonaný presun implementácie prototypu detekcie zmien z letného semestra do zimného semestra. Vzhľadom na túto skutočnosť sa implementácia prototypu vizualizácie zmien presunula na koniec zimného semestra a teda plánom je na nej pracovať počas medzisemestralného obdobia. Taktiež sme sa rozhodli spojiť A/B testovanie a používateľské testovanie. V tabuľke 4 je uvedený aktuálny plán práce na letný semester.

Tabuľka 4: *Upravený plán práce v letnom semestri*

Úloha	Termín
zapracovanie pripomienok z posudku a obhajoby	1. týždeň
dokončenie prototypu vizualizácie zmien	2. týždeň
overenie prototypu detekcie zmien	2. týždeň
podrobný návrh používateľského testovania	3. týždeň
vykonanie používateľského testovania	5. týždeň
spracovanie výsledkov z používateľského testovania	6. týždeň
dokončenie hlavnej textovej časti práce	10. týždeň
spísanie príloh a formálne úpravy	11. týždeň
tvorba prezentácie a príprava na obhajobu	12. týždeň

E4: Zhodnotenie letného semestra

Úlohy určené v upravenom pláne na letný semester (tabuľka 4) sa nám podarilo splniť úplne s výnimkou posledného bodu, t.j. tvorba prezentácie a príprava na obhajobu – 12. týždeň. Dôvodom je fakt, že odovzdanie konečnej verzie tejto práce predchádza dvanástemu týždňu a primárnym cieľom bolo sfinalizovať túto diplomovú prácu.

Príloha F: Obsah elektronického média

/	
— doc	
— ondik_dp.pdf.....	text diplomovej práce
— latex.....	zdrojové súbory DP pre L ^A T _E X
— installer	
— Installer.msi.....	inštalátor pre infraštruktúru rozšírení
— UMLDiff.zip.....	inštalčné súbory pre rozšírenie UmlDiff
— UmlViz.zip....	inštalčné súbory pre vizualizačný nástroj UmlViz
— modely.....	modely použité pri overovaní metód
— src	
— DiffAddin.....	prototyp metódy detekcie zmien
— UmlViz.....	prototyp metódy vizualizácie zmien

Príloha G: Príspevok 1 prezentovaný na IIT.SRC

Visualization of Modifications in Software Models

Jakub ONDIK*

*Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xondik@stuba.sk*

Abstract. During the development of software systems, various software models are created in the software design phase. The models are being further modified not only by customers' changes in system requirements, but as well by design decisions based on problems being solved. These changes are sometimes not visible without previous familiarity with original versions of software models, which can cause loss of efficiency not only for software analysts, but also for developers, who must reflect these modifications into source code. The solution to these problems can be found in proposed method for visualization of modifications in software models.

1 Introduction

Visualization of modifications in evolving software models allows us to efficiently identify the parts of models, which are modified. Due to this fact, this visualization is important not only for software analysts, who are designing and creating the model, but also for software developers, who are reflecting these changes into source code.

Software analysts and developers use various tools for visualization of modifications in software models, both standalone tools¹ and extensions for existing modelling tools [5]. These tools are often different not only in the visualization techniques, but they also differ in supported model and element types.

In this paper we provide an overview of existing tools and approaches for visualization of modifications in software models along with their shortcomings. We also propose the method for visualization of modifications in software models, which supports multiple context-based views in a form of standalone tool and aims to overcome the shortcomings of similar existing tools.

2 Related work

2.1 Fujaba Tool Suite

One of the existing tools providing visualization of modifications in software models is an extension [4] for an UML CASE tool called *Fujaba Tool Suite* [1]. This extension, however, uses extended internal metamodel of Fujaba, which makes it non-reusable for other metamodels, such as UML metamodel. It also only supports Fujaba-specific XMI format with support for Poseidon for UML². This extension supports visualization of modifications only in class diagrams and uses two versions of software models – *reference version*, which is original version of model and *current version*.

Removals, i.e. elements that are present in reference version, but not in current version of model, are highlighted with red color. Additions, i.e. elements present only in current version of model, are highlighted with green color. Changes in elements are visualized with small *U* mark by their side, signifying an update. This extension also does not support change type of modifications for relations and does not support movements visualization at all.

2.2 UMLDiff_{cl}

UMLDiff_{cl} [2] represents an algorithm for visualization of modifications directly in class diagrams. In a case of modification, whole modified part of element, or element itself, is highlighted by color specific to modification type, e.g. if only a type of a field is modified, whole field is highlighted. UMLDiff_{cl} uses eight colors for each of supported modification type, such as successful and unsuccessful element mapping between versions of models, additions, removals, original and current positions of elements, cloning and

* Master study programme in field: Software Engineering
Supervisor: Dr. Karol Rástočný, Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

¹ <https://www.ibm.com/software/products/en/ratirhapfami/>

² <http://www.gentleware.com/>

change. This can result in insufficient clarity of modification visualization and can also result in high cognitive load for users along with high initial need to learn which color corresponds to which modification type. UMLDiff_{clid} also uses its own XML-based model format.

2.3 Sparx Enterprise Architect

Sparx Enterprise Architect³ provides, in addition to software modelling features, means to create model versions in form of snapshots⁴ and their further comparison with other model versions. This comparison is visualized in a table containing list of all properties of compared element in the first column. In the second column current values of the element's properties are shown, while in the third column are snapshot values of the element. The non-matching values are highlighted with green color.

Enterprise Architect also supports visualization of modifications directly in the diagrams⁵. However, only three types of modifications are supported: element movement in diagram, element removal from diagram and element addition to diagram. Visualization of modifications in relations or element parts, such as change of field type, is not supported directly in the diagram, only in a table view.

Unlike the previous tools, Enterprise Architect supports XMI model format for multiple UML versions along with its own UML extensions.

2.4 Shortcomings of state-of-the-art

Presented related tools and algorithms carry multiple shortcomings:

- visualization of modifications limited by diagram and model types,
- lack of visualization for movements of elements' parts and elements themselves,
- missing support for common data format,
- visualization limited to small amount of modification types,
- high cognitive load.

3 Visualization of modifications

In the previous section of this paper we described the shortcomings we want to solve with our proposed method. Therefore, we decided to use UML meta-model for the representation of elements, which allows

us to support all of the UML diagrams and elements. It also allows us to use XMI as a common data format for models. The solution to the other identified shortcomings is described in this section of the paper.

3.1 Static and detail visualization

On Figure 1 we can see the *detail view* of the proposed visualization method. Our method uses only four colors – green for addition, red for removal, orange for change and blue for movement, i.e. single color for every modification type, and only the specific modified parts are visualized. With this, we aim to overcome the shortcomings of similar described tools – visualization of whole element part, even when only its single property was modified, e.g. visualization of whole operation even when only a type of its parameter was modified. and use of a separate color assigned to this type of modification.

Addition and removal of an element is visualized by highlighting the outline of the affected element, due to the fact that the highlighting of the whole element could provide inadequate clarity.

Change, addition and removal of relations is done by visualizing directly affected part. Similar modification visualization approach is used for other elements of diagram (model), such as multiplicities, names of relations, etc., that are not visualized on Figure 1.

Our proposed method supports visualization of element descriptions, which we can see on class *EntitlementType*. This class contains a note icon highlighted with orange color, which represents change. In a case an element was removed only from diagram and not from the whole model, our method uses a visual pattern, such as diagonal greyed stripes.

3.2 Interactive visualization

Proposed method also supports interactive visualization of modifications. Original values of elements and their parts can be seen after movement of mouse cursor to the specific modification, e.g. field *id* of class *Feature* where original type can be seen in a form of *originalValue* → *currentValue*. This is also supported for other element and part types, e.g. relations (aggregation to composition), or element description. In a case of element description, similarly to modifications of other elements' parts, the modified area is highlighted. Only difference is in the shade of used color – the specific modified part is highlighted with a darker shade of color, while the paragraph is high-

³ <http://sparxsystems.com/products/ea/>

⁴ <http://sparxsystems.com/resources/baseline/>

⁵ <http://sparxsystems.com/resources/diff/>

We chose to use this technique instead of *saliency map* [7] due to better support for filters, which we want to use. Our method supports two types of heatmap:

- (i) heatmap showing all types of modifications, while heat is represented by color of modification type and
- (ii) heatmap showing only one type of modifications, which is selected by filter, while heat behaves in a standard way, i.e. shows frequency of specific type of the modification in diagram.

4 Evaluation

To evaluate the proposed method, we plan to carry out an user testing with our software prototype implementing the proposed method. Users will be shown two versions of large software model side-by-side. They will be given simple tasks, such as to identify all movements or field modifications. We will observe the fulfillment and the time these tasks take without and with the use of our method.

We will also evaluate the efficiency of this method. The prototype will be given to the team of software analysts. We will observe the time it takes them to reflect a class diagram modifications to an unchanged state machine diagram. They will also be given a qualitative survey to fill. This survey will help us to evaluate if our method affects the discussion between analysts in the team, i.e. if it supports the discussion in a positive way.

5 Conclusions

In this paper we presented the method for visualization of modifications in software models. We described existing tools providing such a visualization and we identified their shortcomings. In the further parts of this paper, we presented our proposed method for visualization of modifications in software models, which

solves the shortcomings of described existing tools. We described context-based views of modifications and discussed the impact of large software models on our method along with solution in a form of heatmap.

For the future work we plan to evaluate this method in an experimental manner. We want to carry out an user testing, where the users will be given simple tasks to solve without and with our method. We also plan to provide this method to the team of software analysts and observe the efficiency and usability of the method.

References

- [1] Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zndorf, A.: Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer*, 2004, vol. 6, no. 3, pp. 203–218.
- [2] Girschick, M., Darmstadt, T.: Difference detection and visualization in UML class diagrams. *Technical University of Darmstadt Technical Report TUD-CS-2006-5*, 2006, pp. 1–15.
- [3] Kosara, R., Miksch, S., Hauser, H.: Semantic depth of field. *Proc. InfoVis*, 2001, vol. 2001, p. 97.
- [4] Niere, J.: Visualizing differences of UML diagrams with Fujaba. *Proceedings of the 2nd International Fujaba Days 2004*, 2004.
- [5] Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. *ACM SIGSOFT Software Engineering Notes*, 2003, vol. 28, no. 5, p. 227.
- [6] Röthlisberger, D., Nierstrasz, O., Ducasse, S., Pollet, D., Robbes, R.: Supporting task-oriented navigation in IDEs with configurable HeatMaps. *IEEE International Conference on Program Comprehension*, 2009, pp. 253–257.
- [7] Torralba, A., Oliva, A., Castelano, M.S., Henderson, J.M.: Contextual guidance of eye movements and attention in real-world scenes: the role of global features in object search. *Psychological Review*, 2006, vol. 113, no. 4, pp. 766–786.

Príloha H: Príspevok 2 prezentovaný na IIT.SRC

Extendable Software Modelling Architecture

Jakub ONDIK*

*Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xondik@stuba.sk*

Abstract. Existing tools used for software modelling provide us with standard functions, sometimes enriched by means to extend them. However, even if these tools provide us with some kind of extension or plugin systems, they often limit the access to internal representation of software models. We propose the extendable software modelling architecture, which aims to eliminate common problems of similar tools. We also aim to provide a way to easily extend basic features by creating modules called addins, which can provide the user with various features to support modelling abilities of existing modelling tools.

1 Introduction

Existing tools and framework aimed on software modelling provide us only with standard and limited features. They often do not provide the means to extend and enrich their functions. Even when they provide some kind of extension framework, they greatly limit our access to internal representation of software model or underlying meta-information.

One of the frameworks, that improve the means to create various extensions, is EA Addin Framework¹ suited for Sparx Enterprise Architect². However, this framework only encapsulates the standard Enterprise Architect API³ and provides us with simplified access to standard EA API features.

In this paper we propose an extendable software modelling architecture evaluated on top of Sparx Enterprise Architect and we provide a proof of concept in a form of addin featuring difference detection between two versions of software models.

2 Architecture

As the basis for the our framework, we decided to use Sparx Enterprise Architect Automation Interface³. This interface provides us with numerous means to access and manipulate the software model. This includes means to intercept user actions, which can be used to detect defects at the time of their creation and to further improve skills of software designer [4]. The proposed architecture can be seen on Figure 1. We decided to allow the user to manipulate software model via extended Enterprise Architect interface and web interface created by us.

2.1 Local client

Local client component includes three smaller components – Enterprise Architect API, Addin Provider, and Addin Wrapper. *Enterprise Architect API* is a shared library included with Sparx Enterprise Architect. This library consists of interfaces for model manipulation and a base extension class, which is used to connect standard addins to Enterprise Architect. It also provides a way to generate XMI file from loaded model.

Addin Provider is a main interface used to connect our architecture to EA API. It uses *Addin Wrapper* to wire dynamic addins and provides features such as addin loading, unloading and reloading, addin context management (e.g. model memory management) and conversion of model formats, model loading and persistence.

2.2 Web client

Web Client (web browser) component provides us with means to manipulate the software model via web browser. It is interconnected with *Web Addin*

* Master study programme in field: Software Engineering

Supervisor: Dr. Karol Rástočný, Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

¹ <https://github.com/GeertBellekens/Enterprise-Architect-Add-in-Framework>

² <http://sparxsystems.com/products/ea/>

³ <http://sparxsystems.com/resources/developers/autint.html>

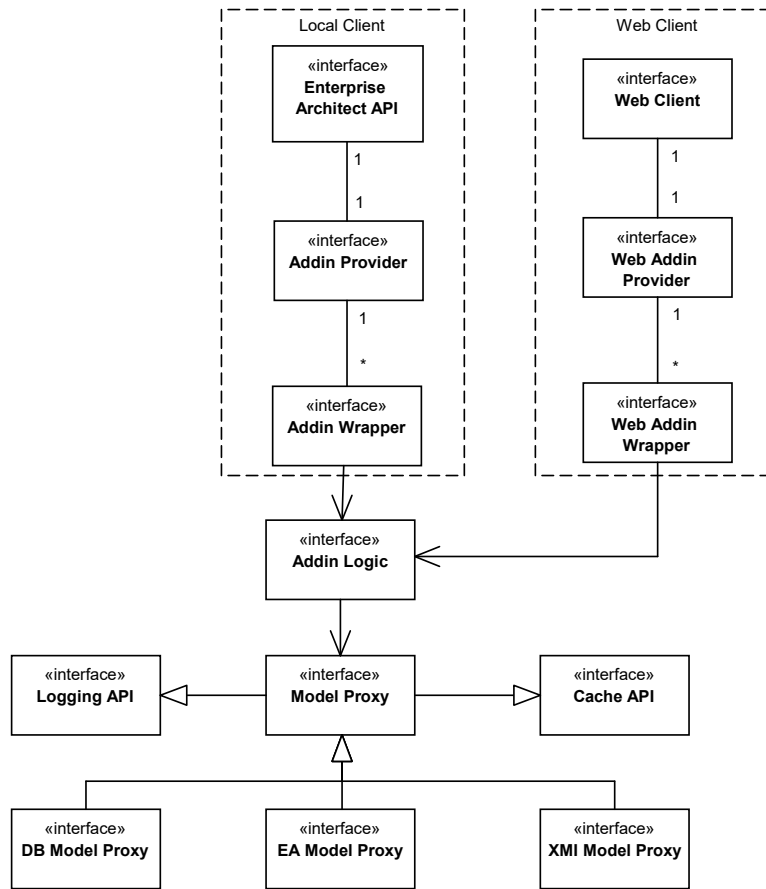


Figure 1. Architecture of the proposed framework.

Provider, which is similar to *Addin Provider* with one distinction – it manipulates the model directly and not through another layer as *Addin Provider*. *Web Addin Wrapper* serves as a proxy and a wrapper over dynamic addins.

In addition to these features, *Web Client* provides means to collaborate on the same project, which can contain multiple models, for multiple users. Users can create projects and add *Items of Interest*. *Items of Interest* can be specified by addins and represent items, that are created, further modified and used by users, e. g. software model versions.

2.3 Shared components

Remaining components of our architecture are used by both *Local client* and *Web client*. Main shared component is *Addin Logic*. This is a base interface, which all of the dynamic addins implement. This interface uses *Model Proxy* to access different types of software models, such as *Database-persisted Model*,

Enterprise Architect Model and *XMI Model*. It also extends *Logging API* and *Cache API*. *Cache API* component provides least recently used caching of model elements, which improves speed of model manipulation by lowering time needed to fetch elements from database (in case of *Database-persisted Models*) and Enterprise Architect Model repository.

3 Proof of concept

As a proof of concept, we decided to create an example addin using the proposed architecture. Existing tools for difference detection in software models are either limited to small number of diagrams, e.g. they only support class diagrams and their extension could prove time-consuming [1], they are limited to specific programming languages [5] or they require construction of initial sets for the algorithm to work [2].

This addin features difference detection between two versions of software model and provides both local Enterprise Architect and web interfaces. In case of local Enterprise Architect version, this addin com-

compares currently loaded model with model given in XMI format. In case of web version of this addin, the addin compares two XMI model versions, which are uploaded through web browser.

We differentiate between two versions of software models – *version A*, which is the reference version and *version B*, which is the current version of model. The steps of the algorithm, which uses Merkle trees [3], used by this addin are described below.

3.1 Creation of Merkle tree

In the first phase, the loaded models are traversed and modified Merkle tree instances are created according to algorithm on Figure 2.

Algorithm 1 Creation of Merkle tree

```

1: function CREATETREE(model)
2:   tree ← new Node()
3:   for every element in model do
4:     tree.Nodes.Add(GenerateNode(element))
5:   end for
6:   tree.ComputeHash()
7:   return tree
8: end function
9: procedure COMPUTEHASH
10:  for every node in Nodes do
11:    node.ComputeHash()
12:  end for
13:  generatedHash ← ComputeHashForPayload(this)
14:  LocalHash ← generatedHash.LocalHash
15:  Hashes ← generatedHash.Hashes
16:  for every node in Nodes do
17:    Hash ← Hash xor node.Hash
18:  end for
19:  Hash ← hashFunction(Hash)
20: end procedure

```

Figure 2. Modified Merkle tree creation algorithm.

We do not describe loading of XMI model nor the Sparx Enterprise Architect Model, as it is provided by *Addin Provider* component. Node hashes are calculated based on attributes of UML element types, such as *PackagedElement* type. We decided to use *xxHash*⁴ non-cryptographic hashing function due to speed and low amount of collisions.

3.2 Tree traversal

After the modified Merkle tree is created, we traverse it and detect differences in node hashes. As for the types of differences, we use states:

- *SAME*, if the hashes are identical in both versions of model,

- *REMOVED*, if the node was removed in the version B of model,
- *ADDED*, if the node was added to the version B of model and
- *INTERNAL_DIFFERENCE*, if we need to traverse attributes of currently processed node.

The detected differences are put into three buckets, based on the type of difference, namely addition bucket, removals bucket and diffs bucket, where nodes with internal differences are put.

3.3 Difference processing

In this phase we process the buckets and create data transfer objects 3. These data transfer objects are then fed into difference table, where the detected differences are visualized. Difference table is in the case of local client shown in the bottom addin window of Sparx Enterprise Architect and in the case of web client, in the main addin window. The differences are colored base on type – red in case of removal, green in case of addition and orange.

Difference	
+	Type: string
+	OriginalValue: string
+	CurrentValue: string
+	OriginalObject: string
+	CurrentObject: string
+	GetAllDifferences(): List<Difference>

Figure 3. Structure of difference Data Transfer Object.

3.4 Model versioning

Model versioning is not supported by local version of this addin – the only way to compute differences between two versions of the model is by providing one of the version in XMI format, while the other version is loaded directly from currently opened model in Enterprise Architect.

Regarding the web version of described addin, model versioning is enabled by use of *Items of Interest* described in section 2.2. Model versions are stored in the database with their respective references to XMI files stored on file system. All of the versions are linked to user projects created. They can be also linked to themselves, creating tree-like structure. This can be useful in cases, when there are multiple versions based on the same version of the model and there is a need to compare or merge them. This addin does not support merging of model versions at the moment,

⁴ <http://cyan4973.github.io/xxHash/>

but it is a part of future work.

4 Conclusions

In this paper we presented an extendable software modelling architecture. This architecture is aimed to simplify and to provide means to access and manipulate software models created in Sparx Enterprise Architect. Our architecture supports all of the UML 2.4.2 diagrams, as it internally uses UML 2.4.2 metamodel. We described the components of this architecture and provided a simple overview.

As a proof of concept we created a addin, which uses the proposed architecture. This addin provides means to detect differences between two versions of UML model and it also provides the users with simple difference visualization. The addin uses modified Merkle trees to detect differences between two versions of software models, which allows us to compare the models more effectively as it can compare the model subtrees as a whole.

For the future work, this framework can be extended by providing means to warn the user of his mistakes, which can help the learning process of new software designers, and by providing activity logging

feature as described in [4].

References

- [1] Costa, V., Monteiro, R., Murta, L.: Detecting Semantic Equivalence in UML Class Diagrams. *The 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2014, pp. 318–323.
- [2] Kessentini, M., Mansoor, U., Wimmer, M., Ouni, A., Deb, K.: Search-based detection of model level changes. *Empirical Software Engineering*, 2016, pp. 1–46.
- [3] Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In Pomerance, C., ed.: *Advances in Cryptology — CRYPTO '87: Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988, pp. 369–378.
- [4] Ondík, J., Olejár, M., Rástočný, K., Bieliková, M.: Activity-Based Model Synchronization and Defects Detection for Small Teams. In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017, Prague, Czech Republic, July 25-29, 2017*, IEEE, 2017, pp. 8–15.
- [5] Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, 2005, p. 54.