

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5220-72178

Bc. Michal Král'

IDENTIFIKÁCIA, KOMPOZÍCIA A KONFIGURÁCIA
EXISTUJÚCICH SOFTVÉROVÝCH KOMPONENTOV PODĽA
ŠPECIFIKÁCIE POŽIADAVIEK VO FORME TEXTU

Diplomová práca

Študijný program: Softvérové inžinierstvo

Študijný odbor: 9.2.5 Softvérové inžinierstvo

Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva

Vedúci práce: Ing. Michal Bystrický

Máj 2018

Zadanie diplomovej práce

Meno študenta: **Bc. Michal Král'**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Konfigurácia a kompozícia existujúcich softvérových komponentov podľa špecifikácie požiadaviek formulovanej vo voľnom texte**

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. Michal Bystrický**

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
Iľkovičova 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: Michal Kráľ, Bc.
Študijný program: Softvérové inžinierstvo
Kontakt: deliko.kral@gmail.com

Výskumník:

Meno, priezvisko, tituly: Michal Bystrický, Ing.

Projekt:

Názov: Konfigurácia a kompozícia existujúcich softvérových komponentov podľa špecifikácie požiadaviek formulovanej vo voľnom texte.
Názov v angličtine: Configuring and composing existing software components from free text requirement specification.
Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky: Programovanie pomocou voľného textu

Text návrhu zadania²

Prípady použitia neposkytujú priestor zachytiť požiadavky na nižšej úrovni abstrakcie. Napríklad oni umožňujú špecifikovať požiadavku procesu objednania v obchode, ale neumožňujú špecifikovať požiadavku ako predpovedať dátum doručenia objednávky. Preto je potrebné vytvoriť ďalšie dokumenty, ktoré toto špecifikujú vo forme voľného textu. Taký voľný text na nižšej úrovni abstrakcie už postačuje na konfiguráciu a kompozíciu existujúcich softvérových komponentov. Konfigurácia a kompozícia existujúcich komponentov podľa špecifikácie požiadaviek formulovanej vo voľnom texte by umožnila vytvárať softvér koncovým používateľom.

Analyzujte prístupy, ktoré umožňujú programovať pomocou voľného textu. Navrhnite metódu konfigurácie a kompozície existujúcich softvérových komponentov podľa špecifikácie formulovanej vo voľnom texte. Zvážte pritom možnosť rozšírenia nejakého existujúceho typu špecifikácie požiadaviek, ako sú prípady použitia alebo používateľské príbehy (user stories). Voľný text navrhnete spracovať pomocou lexikálnej analýzy slov a kľúčových slov. Implementujte podporný nástroj, ktorý toto zabezpečí. Navrhnutú metódu vyhodnoťte prostredníctvom štúdie.

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

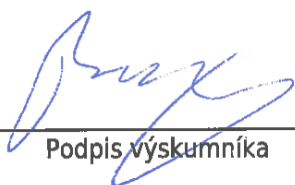
- Śmiałek, M., Jarzębowski, N., & Nowakowski, W. : Translation of use case scenarios to Java code. : Computer Science, 13(4)), 35-52. : 2012
- Franců, J., & Hnětynka, P. : Automated code generation from system requirements in natural language. : OFICZYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ, 73. : 2009

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Michal Kráľ, konzultoval(a) a osvojil(a) si ho Ing. Michal Bystrický a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 20.1.2017



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 Nehodlacie sa prečiarknite

Návrh zadania diplomovej práce

Revízia č.: 1¹

Študent:

Meno, priezvisko, tituly: Michal Kráľ, Bc.
Študijný program: Softvérové inžinierstvo
Kontakt: deliko.kral@gmail.com

Výskumník:

Meno, priezvisko, tituly: Michal Bystrický, Ing.

Projekt:

Názov: Identifikácia, kompozícia a konfigurácia existujúcich softvérových komponentov podľa špecifikácie požiadaviek vo forme textu.
Názov v angličtine: Identification, composition and configuration of existing software components from text requirement specification.
Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky: Programovanie pomocou textu

Text návrhu zadania²

Prípady použitia neposkytujú priestor zachytiť požiadavky na nižšej úrovni abstrakcie. Napríklad umožňujú špecifikovať požiadavku procesu objednania v obchode, ale neumožňujú špecifikovať požiadavku ako predpovedať dátum doručenia objednávky. Preto je potrebné vytvoriť ďalšie textové dokumenty, ktoré toto špecifikujú. Takýto text na nižšej úrovni abstrakcie už postačuje na identifikáciu, konfiguráciu a kompozíciu existujúcich softvérových komponentov. Konfigurácia a kompozícia existujúcich komponentov podľa špecifikácie požiadaviek vo forme textu by umožnila vytvárať softvér koncovým používateľom.

Analyzujte prístupy, ktoré umožňujú programovať pomocou textu. Navrhnite metódu identifikácie, kompozície a konfigurácie existujúcich softvérových komponentov, pričom identifikácia a konfigurácia bude podliehať špecifikácii formulovanej vo forme textu. Zvážte pritom možnosť rozšírenia existujúceho typu špecifikácie požiadaviek ako sú prípady použitia alebo používateľské príbehy (user stories). Text navrhnite spracovať pomocou lexikálnej, sémantickej, syntaktickej analýzy slov, viet a kľúčových slov. Implementujte podporný nástroj, ktorý to zabezpečí. Navrhnutú metódu vyhodnoťte prostredníctvom štúdie.

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

- Šmialek, M., Jarzębowski, N., & Nowakowski, W. : Translation of use case scenarios to Java code. : Computer Science, 13(4)), 35-52. : 2012
- Franců, J., & Hnětynka, P. : Automated code generation from system requirements in natural language. : OFICZYNA WYDAWNICZA POLITECHNIKI WROCLAWSKIEJ, 73. : 2009

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Michal Kráľ, konzultoval(a) a osvojil(a) si ho Ing. Michal Bystrický a súhlasí, že bude takýto projekt viesť.

V Bratislave dňa 15.2.2018



Podpis študenta



Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / nie⁴

Dňa: 26.2.2018



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 (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

⁴ Nehodiace sa prečiarknite

Čestne vyhlasujem, že som túto prácu vypracoval samostatne, na základe konzultácií a s použitím uvedenej literatúry.

V Bratislave, 01.05.2018

Bc. Michal Kráľ

PodĎakovanie

Výskum v tejto diplomovej práci by nebol uskutočnený, nebyť viacerých ľudí. Osobitne ďakujem svojmu školiteľovi Michalovi Bystrickému, lebo bez jeho odborného vedenia by práca nedosiahla svoj plný potenciál a zároveň by na jej základoch nevznikla jedna plnohodnotná vedecká publikácia a zároveň podklady pre druhú vedeckú publikáciu. Osobitá vďaka taktiež patrí aj Zuzane Oboňovej a Beate Kráľovej, pretože bez ich morálnej podpory by som prácu nedokončil. Na záver by som rád poďakoval aj všetkým tým, ktorí mi akokoľvek pomohli pri riešení diplomovej práce alebo vedeckých publikáciách.

Anotácia

Slovenská Technická Univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLOGIÍ

Študijný program: Softvérové inžinierstvo

Autor: Bc. Michal Král'

Diplomová práca: Identifikácia, kompozícia a konfigurácia existujúcich softvérových komponentov podľa špecifikácie požiadaviek vo forme textu

Vedúci diplomovej práce: Ing. Michal Bystrický

Máj 2018

Vývojári, ktorí využívajú aktuálne prístupy programovania pomocou prirodzeného jazyka sú nútení na takéto programovanie používať špeciálne jazyky a zároveň dodržiavať jednoduchosť zápisu. Avšak ani po splnení týchto dvoch podmienok nie sú schopní definovať komplexný softvér. V práci navrhujeme nový prístup komponovania softvéru pomocou detailnej špecifikácie v prirodzenom jazyku a zároveň nástroj podporujúci náš prístup. Detailná špecifikácia rozširuje prípad použitia a navrhovaný prístup z nej na základe sémantickej podobnosti textov identifikuje existujúce komponenty. Následne nájdené komponenty automaticky komponuje a konfiguruje do vyvíjaného softvéru. Výsledky identifikácie a konfigurácie komponentov sú 82,91% a 63,10%. Navrhovaný prístup prináša nový pohľad na to ako programovať pomocou prirodzeného jazyka. Zároveň vidíme veľkú perspektívu vo využití prístupu v rámci prototypovania alebo v spojení s nástrojmi, ktoré umožňujú generovanie používateľského rozhrania.

Annotation

Slovak University of Technology Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree course: Software engineering

Author: Bc. Michal Král'

Master's Thesis: Identification, composition and configuration of existing software components from text requirement specification

Supervisor: Ing. Michal Bystrický

2018, May

Developers which are using actual approaches of natural language programming are forced to use specific languages intended and at the same keep text simple. But even after fulfilling these two conditions they are not able to define complex software. In this thesis, we present a new approach of composing software based on detailed specification in nature language as well as tool to support this. The detailed specification is new part of use case and the approach employs semantic lookup to identify existing components from it. Subsequently, it automatically composes and configures found components in to software being developed. Result of components recognition is 82,91% and components configuration is 63,10%. Suggested approach opens the possibilities for extensive natural language programming support. Using the approach in prototyping or combining it with user interface generation tools is particularly promising.

Obsah

1 Úvod.....	1
2 EUSE – Softvérové inžinierstvo pre koncových používateľov	3
3 NLP – Spracovanie prirodzeného jazyka.....	7
4 Systémové špecifikácie	9
4.1 Prípady použitia	9
4.2 Používateľské príbehy	11
4.3 Výber vhodnej systémovej špecifikácie.....	12
5 CBSE – Softvérové inžinierstvo postavené na komponentoch	15
6 Identifikovanie, komponovanie a konfigurovanie komponentov	19
6.1 Rozšírenie prípadu použitia o nový artefakt – Detailnú špecifikáciu	21
6.2 Identifikovanie komponentov v prípade použitia	21
6.2.1 Spracovanie textu	22
6.2.2 Výpočet sémantickej podobnosti.....	23
6.2.3 Definovanie hodnoty podobnosti	25
6.3 Kompozícia komponentu do projektu.....	26
6.4 Konfigurácia komponentov.....	26
6.5 Rozšírenie komponentov o dodatočnú časť	27
6.6 Príklad	29
6.7 Zosumarizovanie navrhovaného prístupu	31
7 Testovanie	33
7.1 Identifikácia komponentov z prípadov použitia.....	34
7.2 Identifikácia metód konfigurovaných komponentov	35

7.3 Konfigurácia vstupných atribútov volaných metód	36
7.4 Zhodnotenie testovania.....	37
7.4.1 Hrozby validity	38
8 Zhodnotenie	39
Literatúra	43
Príloha A: Technická implementácia	I
Aplikačná vrstva.....	II
Formát prípadov použitia.....	III
Dátová vrstva.....	IV
Prezentačná vrstva.....	VI
Príloha B: Zadanie štúdie na výpočet hodnoty podobnosti.....	IX
Príloha C: Testovacie zadanie.....	XIII
Príklad prípadu použitia s detailnou špecifikáciou.....	XV
Vysvetlenie príkladu	XVI
Príloha D: Používateľská príručka	XVII
Príloha E: Inštalačná príručka	XXI
Príloha F: Obsah priloženého média.....	XXIII
Príloha G: Plán práce na riešení projektu.....	XXV
Príloha H: Vedecká publikácia prijatá na konferencii IIT.SRC.....	XXVII
Príloha I: Plagát použitý pri prezentácii vedeckej publikácie na konferencii IIT.SRC	XXXI

1 Úvod

Prípady použitia sú v dnešnej dobe jedny z najpoužívanějších notácií pri definovaní systémových špecifikácií. Vznikajú na začiatku vývoja softvéru a existujú viaceré štandardy ako zapisovať ich obsah [12, 16], rôzne štandardy vyhovujú rôznym prípadom [12, 16] a vďaka tomu majú veľký potenciál v softvérovom inžinierstve pre koncových používateľov. Práve upravovanie a rozširovanie prípadov použitia o dodatočné artefakty môže viesť k uľahčeniu samotného vývoja softvéru. Predstavte si systémovú špecifikáciu vo forme prípadov použitia, z ktorej je možné automaticky nakomponovať a nakonfigurovať softvér do finálnej podoby alebo aspoň do podoby, v ktorej už netreba robiť radikálne zásahy. Tvorba softvéru by sa značne zrýchlila a vznikol by väčší priestor pre plnenie požiadaviek klienta.

Viacero výskumníkov skúšalo prekladanie špecifikácie v prípadoch použitia na funkčný kód [17, 43]. Aby bol takýto preklad možný, musí text špecifikácie spĺňať rôzne podmienky ako napríklad použitie špecifického jazyka [43] alebo jednoduchosť zápisu [17]. Lenže vygenerovaný kód takýmto spôsobom, je málokedy automaticky funkčný a väčšinou nespĺňa to, čo bolo zadané v špecifikácii prípadu použitia na požadovanej úrovni. Dodatočná úprava takto vygenerovaného kódu sa tiež vzdáva od softvérového inžinierstva pre koncových používateľov. Zároveň takýmto spôsobom nie je možné generovať komponenty, ktoré vyžadujú komplexnejšiu špecifikáciu a to z dôvodu zatiaľ nedostačujúceho pokroku v oblasti spracovania prirodzeného jazyka [17].

Avšak myslíme si, že aktuálny stav v oblasti spracovania prirodzeného jazyka postačuje na automatickú identifikáciu, kompozíciu a konfiguráciu existujúcich softvérových komponentov. Keby sme mali v texte v prirodzenom jazyku detailne špecifikované aký komponent požadujeme a ako a s akou konfiguráciou ho chceme využiť, potom by sme na základe tohto textu boli schopní opísaný komponent automaticky identifikovať, nakomponovať a nakonfigurovať. Z tejto hypotézy nám vzniká viacero otázok. Ako podľa textu v prirodzenom jazyku správne identifikovať a nakonfigurovať požadovaný komponent. Ako odlíšiť podobné komponenty. A ako identifikované komponenty nakomponovať.

V práci navrhujeme nový prístup automatického komponovania softvéru pomocou detailnej špecifikácie v prirodzenom jazyku. Detailná špecifikácia rozširuje prípad použitia a navrhovaný prístup z nej na základe sémantickej podobnosti textov identifikuje existujúce softvérové komponenty, ktoré sú rozšírené o nami pridané údaje. Následne identifikované komponenty automaticky komponuje a konfiguruje do vyvíjaného softvéru.

Predpokladáme, že spojením nášho prístupu s technológiami, ktoré prekladajú systémovú špecifikáciu na funkčný kód [17, 43] by sa znížila a možno aj úplne odpadla semi-automatickosť týchto technológií. Použitím existujúcich softvérových komponentov sa zníži počet riadkov kódu, ktoré sa automaticky generujú z textu a tak klesá chybovosť vygenerovaného zdrojového kódu.

Navrhovaný prístup sa dá využiť počas celého vývoja softvéru, avšak jeho najväčšie uplatnenie vidíme vo fáze prototypovania po definícii systémových špecifikácií. Myslíme si, že dokáže túto fázu vývoja softvéru značne urýchliť. Zároveň vidíme perspektívu v spojení nami navrhovaného prístupu s nástrojom, ktorý dokáže generovať používateľské rozhranie. Takýmto spojením sa približujeme opäť o čosi bližšie k cieľu softvérového inžinierstva pre koncových používateľov.

Práca je rozvrhnutá nasledovne: (1) Predstavujeme problematiku práce. (2) Predstavujeme problémy EUSE¹ a vedecké publikácie, ktoré sa venujú podobnej problematike ako my. (3) Opisujeme techniky NLP² relevantné pre výskum našej práce. (4) Definujeme v súčasnosti najpoužívanejšie systémové špecifikácie a predkladáme argumenty, prečo sme sa rozhodli rozšíriť práve prípady použitia. (5) Určujeme základné formy existujúcich softvérových komponentov a rozoberáme problematiku komponovania komponentov v rozličných oblastiach. (6) Predstavujeme návrh nášho prístupu, ktorý na základe textu v prirodzenom jazyku identifikuje, komponuje a konfiguruje existujúce softvérové komponenty do vyvíjaného softvéru. (7) Testujeme navrhovaný prístup. (8) Sumarizujeme prácu a opisujeme využitie prístupu v širšom kontexte.

¹ End user software engineering

² Natural language processing

2 EUSE – Softvérové inžinierstvo pre koncových používateľov

Softvérové inžinierstvo pre koncových používateľov (odteraz EUSE) už nie je doménovou oblasťou, ktorej sa venujú len ľudia zruční v informatike. V posledných rokoch vzniklo viacero technológií, pomocou ktorých je bežný používateľ schopný vytvoriť ním požadovaný softvér bez väčších problémov. Ukážkou takýchto technológií sú napríklad Wordpress³, Joomla⁴, Drupal⁵, Wix⁶, Websitebuilder⁷ a iné. Avšak tieto technológie sú obmedzené tým, že sú zamerané vždy len na jeden konkrétny typ softvéru. Zatiaľ neexistuje technológia, pomocou ktorej by bol používateľ, ktorý nemá skúsenosti s informatikou, schopný sám vytvoriť komplexnejší softvér z ľubovoľnej domény. Práve EUSE je oblasť výskumu, ktorá sa zaoberá vývojom technológií, ktoré môžu využívať koncoví používatelia na vývoj alebo zlepšenie ich softvérov [10]. Jedným z cieľov tejto oblasti výskumu je, aby používatelia, ktorí nemajú zručnosti s informatikou boli sami schopní vytvoriť komplexný softvér podľa ich požiadaviek.

Vývoj v tejto vedeckej oblasti je oproti minulému storočiu viditeľný a existuje už viacero EUSE technológií, ktoré používa široká verejnosť. Avšak pokrok priniesol aj nové otázky a problémy, ktorých vyriešenie je kľúčové pre ďalší vývoj tejto oblasti:

- Ľudia nechcú používať EUSE technológie, lebo na oboznámenie sa s ich používaním musia obetovať svoj čas a neveria, že im tieto technológie prinesú požadovaný výsledok [27].
- Programy vytvorené pomocou EUSE technológií majú často implementačné chyby [9]. V prípade, že EUSE technológia vytvorila softvér s chybami, ich nájdenie a opravu môže vykonať len jej používateľ, problém vzniká, ak používateľ nemá dostatočné zručnosti na vykonanie týchto úkonov.
- EUSE technológie sú často semi-automatické a k dosiahnutiu finálneho zámeru sú ešte nutné dodatočné zásahy používateľa [17, 43].

Nami navrhovaný prístup inklinuje k dvom hlavným odvetviam - k softvérovému inžinierstvu pre koncových používateľov a k softvérovému inžinierstvu postavenému na komponentoch (odteraz CBSE). Myslíme si, že práve hlbšie prepojenie týchto dvoch odvetví by malo priniesť riešenia na uvedené problémy. Napríklad by to mohlo priniesť nový pohľad na to, ako generovať zdrojový kód priamo z textu v prirodzenom jazyku. Použitím existujúcich softvérových komponentov by sa znížil počet riadkov kódu, ktoré sa automaticky generujú z textu a tak by klesla chybovosť vygenerovaného zdrojového kódu. Zároveň by sa znížila a možno by aj úplne odpadla semi-automatickosť týchto technológií. Pričom používateľ ani nemusí vedieť, aké komponenty má k dispozícii, stačí ak ich v rámci textu opíše vlastnými slovami.

³ <https://wordpress.com/>

⁴ <https://www.joomla.org/>

⁵ <https://drupal.org/>

⁶ <https://www.wix.com/>

⁷ <https://www.websitebuilder.com/>

V práci sa venujeme odvetviu EUSE, ktoré využíva systémovú špecifikáciu definovanú používateľom vo forme textu k vytvoreniu funkčného softvéru. Oblasť výskumu EUSE, ktorá sa zaoberá vytvorením softvéru z textu, už bola spracovaná vo viacerých úspešných vedeckých prácach [17, 28, 31, 32, 42, 43].

Výskum *Keyword Programming in Java* poukázal na efektívnosť programovania pomocou kľúčových slov [31]. Za týmto účelom autori vytvorili aj plugin do vývojového prostredia Eclipse, vďaka ktorému sprístupnili takýto typ programovania v jazyku Java. Ich algoritmus pomocou kľúčových slov pokrýva až 90% možných výrazov a konštrukcií v tomto jazyku [31]. Takéto programovanie vyžaduje zručnosť s programovaním ako takým a zároveň znalosť možných kľúčových slov. Je teda vhodné pre koncových používateľov, ktorí majú znalosti s informatikou.

Výskum *Automated Code Generation from System Requirements in Natural Language* sa zaoberá automatickým prekladom textu v prirodzenom jazyku v prípadoch použitia na spustiteľný kód [17]. Autori vytvorili softvér Procastor, ktorý takýto preklad zabezpečuje. Procastor vyžaduje, aby vstupné prípady použitia boli v presne definovanom formáte a ich špecifikácia musí byť napísaná jednoduchou formou. V prípade, že tieto podmienky nie sú dodržané, Procastor nie je schopný vygenerovať funkčný kód [17]. Procastor zároveň nie je schopný generovať komponenty, ktoré vyžadujú komplexnejšiu špecifikáciu.

Výskum *Translation of use case scenarios to Java code* sa takisto zaoberá automatickým prekladom prirodzeného jazyka v prípadoch použitia na spustiteľný kód, avšak z prípadov použitia využíva len ich scenáre [43]. Tie musia podliehať forme zjednodušeného jazyka RSL [43]. Takto napísané scenáre sa podrobia transformačným pravidlami a zo získaných dát sa automaticky generuje kód v jazyku Java. Navrhovaný prístup je však semi-automatický a jazyk RSL nezabezpečuje, že zo scenáru bude vygenerovaný funkčný kód, takýto kód treba ešte dodatočne upravovať [43]. Jazyk RSL je zároveň obmedzujúci a je vyžadované, aby koncový používateľ ovládal jeho syntax.

Výskum *Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications* pojednáva o identifikovaní a komponovaní existujúcich komponentov na základe jednoduchej nekompletnej špecifikácie vo forme vstupno/výstupných dvojíc a jednoduchého slovného opisu čiastočných fragmentov komponentov [42]. Výskum je postavený na technológii Yahoo! Pipes [42], ktorej úlohou je zlučovanie a úprava dát, hlavne vo formáte RSS⁸, JSON⁹ a Atom¹⁰, podobne ako to je v jazyku Shell. Avšak výskum je limitovaný len na komponenty, ktoré spracovávajú textové reťazce, upravujú dáta v zoznamoch alebo robia jednoduché matematické operácie [42]. Zároveň časová náročnosť vytvorenia vstupno/výstupných dvojíc je porovnateľná s manuálnym nájdením/naprogramovaním riešenia daného problému.

⁸ Rich Site Summary

⁹ JavaScript Object Notation

¹⁰ Atom Syndication Format

Výskum *CodeGenie: a Tool for Test-Driven Source Code Search* využíva zaznamenané chyby počas automatického generovania testov technológiou JUnit [28]. Z chýb identifikuje názvy chýbajúcich metód a tried. Následne pomocou kľúčových slov obsiahnutých v identifikovaných názvoch vyhľadáva požadovanú funkcionálnu technológiou Sourcerer [28]. Prístup je priamo závislý na technológii JUnit, vďaka tomu ho nie je možné aplikovať v rámci iných ako Javových projektov.

Z vyššie spomenutých prác nám vyplýva, že automatické generovanie komplexnejšieho zdrojového kódu podľa špecifikácie zadanej vo forme textu v prirodzenom jazyku, s aktuálnym stavom v oblasti spracovania prirodzeného jazyka zatiaľ nie je možné. Preto je nutné hľadať iné prístupy, pomocou ktorých by bol koncový používateľ schopný vytvoriť komplexnejší softvér z ľubovoľnej domény. V práci prichádzame s návrhom prístupu, ktorý tvorbu takéhoto softvéru pomocou textu v prirodzenom jazyku koncovým používateľom umožňuje. Text rozširuje systémovú špecifikáciu a obsahuje detailné informácie, pomocou ktorých je navrhovaný prístup schopný automaticky identifikovať, nakomponovať a nakonfigurovať existujúce softvérové komponenty do vyvíjaného softvéru. K identifikácii komponentov využíva sémantickú podobnosť slov a viet.

3 NLP – Spracovanie prirodzeného jazyka

Navrhovaný prístup spracováva text za cieľom identifikácie a konfigurácie existujúcich softvérových komponentov. Odvetvie informatiky, ktoré sa zaoberá spracovaním textu sa volá *spracovanie prirodzeného jazyka* (odteraz NLP). Hlavný zámer NLP je extrahovanie gramatickej štruktúry a významu z textu [1]. V práci sa primárne venujeme nasledovným technikám NLP: lexikálna analýza, syntaktická analýza, sémantická analýza a určenie kľúčových slov. Samotná výskumná oblasť NLP avšak disponuje aj inými technikami [26].

Pri spracovávaní prirodzeného jazyka tvoria lexikálna a syntaktická analýza základné stavebné zložky. Bez nich by sme len ťažko vedeli správne určiť kľúčové slová alebo priradiť slovám vo vetách pravý význam. Ich hlavné úlohy sú: identifikácia tokenov¹¹, určenie slovných druhov, určenie vetných členov, identifikovanie lema¹² tvarov tokenov, identifikovanie názvoslovných entít a iné. Medzi najrozšírenejšie nástroje určené hlavne pre lexikálnu a syntaktickú analýzu patria StanfordNLP [33], NLTK [6] a OpenNLP¹³.

Vo všeobecnosti platí, že textu dávajú najväčší význam kľúčové slová, preto je dôležitá ich správna identifikácia. Kľúčové slová sa používajú už dlhší čas a osvedčili sa v nasledujúcich úlohách: automatické indexovanie, sumarizácia dokumentu, kategorizácia textu a zhlukovanie textu. Hlavné vlastnosti slova v dokumente sa môžu rozdeliť do dvoch kategórií, štatistické a sémantické. Na základe týchto kategórií sa vyvinuli rôzne metódy extrahovania kľúčových slov [15]. Medzi metódy, ktoré sú založené na štatistických vlastnostiach sa radia: frekvencia výskytu, TFIDF¹⁴, GenEx, KEA¹⁵, spoločný výskyt slov a sieťový model. Kvalita týchto metód nie je príliš veľká, pokiaľ sa extrahujú kľúčové slová z nezávislých dokumentov alebo krátkych textov. Medzi metódy, ktoré sú založené na sémantických vlastnostiach sa radia: lexikálne reťaze a KEA++. Tieto metódy sú závislé na jazyku a väčšinou vyžadujú dodatočný lexikálny slovník.

Hlavnou úlohou sémantickej analýzy je priradenie významu jednotlivým slovám vo vete. K tomu sú využívané lexikálne slovníky ako je napríklad Wordnet [34] alebo Thesaurus. V rámci práce sa venujeme jednej konkrétnej časti sémantickej analýzy - sémantickej príbuznosti a podobnosti slov. Sémantická príbuznosť a podobnosť sú dva rozdielne pojmy. Vo všeobecnosti platí, že podobné slová sú podmnožinou slov príbuzných. Sémantická príbuznosť je všeobecná, a tak príbuznosť medzi slovami nemusí byť na prvý pohľad očividná, napríklad slová *škola* a *budova* [14]. Pri sémantickej podobnosti je podobnosť medzi slovami na prvý pohľad zjavná, napríklad slová *variť* a *vrieť* [14].

¹¹ Základná časť slova.

¹² Základný tvar tokenu.

¹³ <https://opennlp.apache.org/>

¹⁴ Term frequency–inverse document frequency.

¹⁵ <http://community.nzdl.org/kea/>

Na zistenie percentuálnej sémantickej podobnosti existuje šesť celosvetovo uznávaných metód [39]. Tieto metódy sa radia do dvoch základných skupín:

1. Metódy založené na dĺžke cesty medzi pojmami v štruktúre sémantického grafu:

Metóda WUP [45]

Nájde najbližší spoločný rodičovský vrchol hľadaných slov. Týmto vrcholom je slovo, ktoré najlepšie vystihuje obe slová zároveň. Hodnota, ktorá je výsledkom tejto metódy je vypočítaná ako suma dĺžok ciest z individuálnych konceptov k rodičovskému vrcholu [39].

Metóda LCH [29]

Nájde najkratšiu vzdialenosť medzi dvoma slovami a zobrazí túto hodnotu ako maximálnu dĺžku v hierarchii podobnej rodičovskému stromu, v ktorej sa dané slová nachádzajú [39].

Metóda PATH

Výsledkom metódy je obrátený súčet vrcholov na najkratšej ceste medzi hľadanými slovami [39].

2. Metódy založené na porovnaní definícií slov v slovníkoch:

Metóda RES [41]

Výsledkom je podobnosť významov hľadaných slov, založená na informačnom obsahu ich najbližšieho spoločného rodičovského vrcholu.

Metóda JNC [24] a metóda LIN [30]

Obe metódy sú variáciami metódy RES [39].

Na zistenie percentuálnej sémantickej príbuznosti existujú tri celosvetovo uznávané metódy [39]. Sú to metódy LESK [5], HSO [22] a VECTOR [38].

Vývojári prístupov, ktoré počítajú sémantickú príbuznosť a podobnosť, zvyknú jednotlivé metódy zlučovať za cieľom dosiahnuť presnejší výsledok. Ich cieľom je vytvoriť techniku, pomocou ktorej sa bude dať určiť sémantická príbuznosť a podobnosť čo najpresnejšie a vďaka čomu najmenšiemu zdroju vedomostí. V posledných rokoch tak vznikli aj ďalšie metódy, ktoré v doméne sémantickej podobnosti a príbuznosti tiež dosahujú dobré výsledky [14, 35, 46]. Avšak tieto metódy zatiaľ nemajú vytvorený verejne dostupný softvér a práca samotná nepojednáva o overení týchto metód, preto sa nimi ďalej nebudeme zaoberať.

4 Systémové špecifikácie

Navrhovaný prístup bude rozširovať nami vybranú notáciu o dodatočný artefakt, s názvom *detailná špecifikácia*, vo forme textu v prirodzenom jazyku. Text detailnej špecifikácie bude bližšie charakterizovať komponenty, ktoré sú všeobecne definované v notácii. Dôvodom, prečo chceme rozširovať už existujúcu notáciu a nie definovať novú, je fakt, že používatelia sa často stráňa úplne novým technológiám [27], ale väčšinou sú otvorení rozšíreniam existujúcich technológií. Vďaka rozšíreniu tak bude s používaním nami navrhovaného prístupu stotožnených viacero ľudí.

Medzi najčastejšie notácie pri definovaní systémových špecifikácií v čase vývoja systému patria prípady použitia a používateľské príbehy. Každá z týchto notácií má svoje miesto vo vývoji softvéru a obsahuje určitú formu systémovej špecifikácie. Jednotlivé notácie sú opísané v nasledujúcich podkapitolách.

4.1 Prípady použitia

Prípady použitia sa väčšinou špecifikujú na začiatku vývoja softvéru a zostavujú sa na základe požiadaviek zákazníka. Primárne sa využívajú počas analýzy a návrhu softvéru a v rámci vývoja softvéru môžu do určitej miery upravovať. Aby sme sa vyhli úprave prípadu použitia v neskoršej fáze vývoja softvéru, tak sa na začiatku vývoja vytvárajú prototypy navrhovaného softvéru, ktoré vychádzajú práve z prípadov použitia.

Existuje viacero spôsobov ako prípady použitia zapisovať, v konečnom dôsledku, ale všetky vychádzajú z dvoch základných notácií a to Jacobsonova alebo Cockburnova notácia.

Cockburn definoval prípady použitia nasledovne: Prípad použitia je popis možných poradí interakcií medzi systémom a jeho externými aktérmi, vzťahujúci sa k určitému cieľu [12]. Samotný autor sa neskôr vyjadril, že táto definícia nie je úplná a chýba v nej časť, ktorá by ponímala o záväzkoch systému k zainteresovaným stranám.

Jacobson definoval prípady použitia nasledovne [23]: Prípad použitia definuje možnosti ako používať systém tak, aby sme dosiahli určitý cieľ, ku ktorému používateľ smeruje. Skupina všetkých prípadov použitia tak určuje všetky možnosti ako systém používať.

Spôsob ako prípady použitia kresliť do diagramov priniesol Jacobson (1994). Od vtedy sa diagram prípadov použitia stal súčasťou skupiny UML diagramov [16].

Špecifikácia prípadu použitia:

Existujú viaceré štandardy ako zapisovať obsah prípadu použitia [12, 16], rôzne štandardy vyhovujú rôznym projektom a firmám [12, 16]. Veľké korporácie si prípady použitia častokrát prispôbujú tak, aby lepšie vyhovovali ich metodikám. Avšak existuje pár základných elementov, ktoré prípady použitia obsahujú skoro vo vždy.

Názov: Krátky výstižný popis.

Popis: Dlhšia netechnická špecifikácia prípadu použitia. Rozširuje názov o nové poznatky a tak presnejšie opisuje, čo je cieľom prípadu použitia.

Aktéri: Aktérom môže byť čokoľvek, čo má správanie, nemusí ísť nevyhnutie o človeka, môže ním byť aj systém alebo čas [12]. Aktéri môžu byť primárni alebo sekundárni, primárni sú tí, ktorí sa snažia dostať k cieľu prípadu použitia a sekundárni sú tí, ktorí dopomáhajú dostať sa k cieľu prípadu použitia [12].

Predpoklady: Požiadavky, ktoré musia byť splnené, aby započal scenár prípadu použitia [12]. Predpoklady sú vždy brané ako pravdivé a nemusia sa na začiatku scenára prípadu použitia overovať [12].

Scenár: Postupnosť krokov, ktoré musia byť splnené, aby sa aktér dostal k cieľu prípadu použitia [7, 12]. Samotné kroky sú definované pomocou voľného jazyka. Krok je úspešný, ak sa úspešne ukončili všetky jeho časti. Prípad použitia vždy obsahuje hlavný scenár a ak je to potrebné, tak aj alternatívne scenáre. Alternatívne scenáre sú vždy podmienené akciou.

Ukážka prípadu použitia:

Názov: Kontaktovanie zákaznického servisu

Primárni aktéri: Zákazník, Zamestnanec

Sekundárni aktéri: Systém

Popis: Zákazník chce položiť zákazníkemu servisu otázku.

Scenár:

1. Zákazník chce položiť otázku.
2. Systém vyžiada od zákazníka, aby zadal otázku a jeho emailovú adresu.
3. Zákazník zadá jeho otázku a emailovú adresu a odošle požiadavku.
4. Systém upozorní zamestnanca na novú otázku.
5. Zamestnanec odpovie na otázku.
6. Systém odošle odpoveď na zákazníkovu emailovú adresu.

Alternatívne scenáre:

- 3a. Emailová adresa nie je zadaná v správnom formáte.

Napojenie: Bod 3.

Scenár:

1. Systém informuje zákazníka o zle zadanej emailovej adrese.

4.2 Používateľské príbehy

Sú to krátke, jednoduché opisy funkcionalít, zvyčajne špecifikované používateľom alebo zákazníkom softvéru. Vytvárané sú skupinou vývojárov, ktorá počas ich vytvárania diskutuje o tom, či špecifikujú vlastnosti, ktoré sa od finálneho produktu očakávajú. Ukladajú sa do produktového backlogu, z ktorého sa neskôr podľa priority vyberajú.

Benefitom používateľských príbehov je, že sa dajú písať na rôznej úrovni granularity. Od obsiahlych nazývaných aj epiky, cez menej obsiahle nazývané aj vlastnosti až po malé používateľské príbehy.

Vytvárajú sa počas celého agilného vývoja produktu. Zvyčajne sa ich najviac napíše na začiatku vývoja, kedy sa definujú ciele produktu. V prípade, že sú vytvorené aj diagramy prípadov použitia, tak hlavné používateľské príbehy, epiky, vychádzajú práve z týchto diagramov.

Špecifikácia používateľského príbehu:

Tak ako prípady použitia, tak aj používateľské príbehy majú viacero štandardov zápisu. Korporácie si ich prispôbujú tak, aby viac vyhovovali ich požiadavkám. Používateľský príbeh najčastejšie pozostáva z nasledovných elementov:

Názov: Krátky výstižný popis.

Popis: Obsiahlejšia špecifikácia úlohy. Najčastejšie sa pri definovaní popisu využíva nasledujúci formát: Ako „typ používateľa“ chcem „definovaný cieľ“ pretože „odôvodnenie“. Samotný popis neobsahuje bližšiu technickú špecifikáciu. Tú neskôr obsahujú úlohy daného používateľského príbehu.

Priorita: Dôležitosť vzhľadom na ostatné používateľské príbehy.

Úroveň obtiažnosti: Najčastejšie sa určuje pomocou story pointov.

Akceptačné kritériá: Čo je požadované od vlastníka produktu, aby akceptoval daný používateľský príbeh.

Ukážka používateľského príbehu:

Názov: Informácie o účtoch detí používateľa.

Popis: Ako používateľ chcem vo svojom internet bankingu vidieť pohyby na účtoch mojich detí, aby som vedel ako sa učia narábať s peniazmi.

Priorita: 21

Úroveň obtiažnosti: 8

Akceptačné kritériá:

Zobrazené všetky účty.

Formulár pre filtrovanie informácií obsiahnutých v pohyboch na účte.

Možnosť zablokovat' / odblokovat' manipuláciu s peniazmi na účte.

4.3 Výber vhodnej systémovej špecifikácie

Navrhovaný prístup bude rozširovať nami vybranú notáciu o dodatočný artefakt, s názvom *detailná špecifikácia*, vo forme textu v prirodzenom jazyku. Text detailnej špecifikácie bude bližšie charakterizovať komponenty, ktoré sú všeobecne definované v notácii. Úlohou navrhovaného prístupu je zjednodušenie vývoja softvéru a priblíženie sa k EUSE takým spôsobom, že používateľ nebude musieť skoro vôbec písať kód softvéru. K tomu potrebujeme, aby bol nami pridaný artefakt definovaný v ranných fázach vývoja softvéru. Preto je nutné vybrať notáciu, ktorá spĺňa tieto podmienky čo najpresnejšie.

Pri výbere vhodnej notácie, ktorú by sme rozšírili o dodatočný artefakt sme uvažovali aj nad dilatáciou rôznych druhov testov v rámci vývoja riadeného testom (odteraz TDD). Priaznivým aspektom TDD je, že testy v ňom vznikajú na začiatku vývoja softvéru. Medzi hlavné testy, nad ktorými sme uvažovali sú používateľské akceptačné testy (odteraz UAT) a systémové akceptačné testy (odteraz SAT). Avšak UAT a SAT opisujú spôsob overenia určitej funkcionality softvéru a nie princíp jej fungovania. Vďaka tomu používateľ nemusí identifikovať všetky komponenty, ktoré sa v UAT a SAT nachádzajú alebo ich vďaka nedostačujúcim informáciám o princípe fungovania funkcionality môže zle charakterizovať.

Vhodnými kandidátmi sú teda tie typy notácii, ktoré adekvátne opisujú funkcionality softvéru a zároveň vznikajú v prvých fázach vývoja softvéru. Medzi tie sa radia prípady použitia a používateľské príbehy. Oba štandardy opisujú funkcionality softvéru z určitého uhla pohľadu, ich základom sú požiadavky zákazníka, sú ľahko pochopiteľné, predstavujú vždy ucelenú časť softvéru, dajú sa písať na rôznej úrovni granularity a nemajú striktné stanovenú doktrínu, ktorú je nutné pri ich vytváraní dodržiavať.

Avšak v prípadoch použitia vidíme väčší potenciál na rozšírenie z nasledujúcich dôvodov:

1. Používateľské príbehy sa na rozdiel od prípadov použitia využívajú len pri agilnom vývoji softvéru. Nechceme, aby bol náš prístup limitovaný typom vývoja softvéru.

2. Na základe prípadov použitia sú vytvárané prototypy. Spojením rozšíreného prípadu použitia s nástrojom určeným na prototypovanie by sa využil plný potenciál navrhovaného prístupu.
3. *Detailná špecifikácia* obsahuje okrem iného aj technickejší opis funkcionalít obsiahnutých v prípade použitia. Práve tento opis núti zamyslieť sa hlbšie nad časťami prípadu použitia, a to môže viesť k objaveniu nových alternatívnych tokov. Príklad objavenia takéhoto toku sa nachádza nižšie.

Objavenie alternatívneho toku v prípade použitia pomocou detailnej špecifikácie:

Pouvažujte nad nasledovným prípadom použitia:

Názov: Kontaktovanie zákazníckeho servisu

Primárni aktéri: Zákazník, Zamestnanec

Sekundárni aktéri: Systém

Popis: Zákazník chce položiť zákazníkemu servisu otázku.

Scenár:

1. Zákazník chce položiť otázku.
2. Systém vyžiada od zákazníka, aby zadal otázku a jeho emailovú adresu.
3. Zákazník zadá jeho otázku a emailovú adresu a odošle požiadavku.
4. Systém upozorní zamestnanca na novú otázku.
5. Zamestnanec odpovie na otázku.
6. Systém odošle odpoveď na zákazníkovu emailovú adresu.

Alternatívne scenáre:

- 3a. Emailová adresa nie je zadaná v správnom formáte.

Napojenie: Bod 3.

Scenár:

1. Systém informuje zákazníka o zle zadanej emailovej adrese.

V prípade použitia zákaznícky servis odpovedá zákazníkovi prostredníctvom emailu. Avšak v čase písania prípadu použitia si častokrát neuvedomujeme všetky súvislosti, napríklad to, že

firma má vytvorenú infraštruktúru emailových serverov, ktorá môže byť využitá. Práve *detailná špecifikácia* obsahuje okrem iného aj technickejší opis komponentov obsiahnutých v prípade použitia. A práve tento opis núti zamyslieť sa hlbšie nad časťami prípadu použitia, to môže viesť k objaveniu nových alternatívnych tokov. *Detailná špecifikácia* k tomuto prípadu použitia môže vyzerat' aj nasledovne:

Detaily o komponente: Emailový klient. Email bude odoslaný z foo@ehop.com. Kópia bude odoslaná na bar@eshop.com. Pre odosielanie emailov sa využije infraštruktúra SMTP serverov firmy.

Po zamyslení sa nad komponentom, emailovým klientom, sme si spomenuli na infraštruktúru SMTP serverov, ktoré firma vlastní. A vďaka tomu sme prišli na nový alternatívny tok, ktorý zachytáva a rieši problém s odoslaním emailu.

Alternatívne scenáre:

6a. Systém nedokáže odoslať email z aktuálneho SMTP servera.

Napojenie: Bod 6.

Scenár:

1. Systém využije iný SMTP server v rámci firmy.

5 CBSE – Softvérové inžinierstvo postavené na komponentoch

Nami navrhovaný prístup inklinuje k CBSE práve tým, že využíva už existujúce softvérové komponenty a pomocou špecifikácie vo forme textu ich identifikuje, komponuje a konfiguruje. CBSE je oblasť výskumu, ktorá sa zaoberá vývojom softvéru tak, že spája už existujúce komponenty do väčších celkov. Pre kompozíciu sa využívajú rozhrania komponentov a zároveň tzv. lepidlá, ktoré môžu mať viacero tvarov [2, 11, 19, 36, 40].

Pomocou lepidla je definované, ktorý komponent a akým spôsobom sa bude komponovať. Ukážkou jedného z najjednoduchších lepidiel je shell v operačnom systéme UNIX, ktoré vďaka návrhovému vzoru pipe and filters dokáže pospájať komponenty do zložitých štruktúr. Avšak jeho nevýhodou je nutnosť znalosti pomerne zložitého jazyka. Niektoré formy lepidiel nám dovoľujú kompozíciu komponentov aj z úplne iných oblastí, ako napríklad komponent napísaný v jazyku Java s komponentom napísaným v jazyku Python [11]. Lepidlo vždy spolupracuje len s rozhraním komponentu a nie s komponentom samotným, k spolupráci môže využívať rôzne druhy konektorov [2, 37]. Z toho dôvodu vzniká požiadavka na interaktívne rozhranie, ktoré sa dá jednoducho, efektívne a automaticky využiť za účelom sprostredkovania funkcionality komponentu.

Zahrnutie CBSE vo vývoji softvéru zvyšuje produktivitu, znižuje cenu celkového produktu a zabezpečuje väčšiu konzistenciu [8, 20]. Jeho používanie však prichádza aj s pár nedostatkami [13]:

- Vytvorenie komponentov.
- Správa komponentov.
- Zabezpečenie znovu použiteľnosti komponentov.
- Je možné, že komponent nebude pokrývať všetku funkcionality, ktorú od neho aplikácia vyžaduje.

Ucelená definícia komponentu ako takého neexistuje, rôzne problémy vyžadovali odlišné definície [4, 13, 20, 25, 37]. V našej práci sa stotožňujeme s nasledovnou definíciou [4, 25]: Za komponent považujeme ucelené zdrojové kódy, ktoré majú určitú funkcionality sprostredkovanú skrz rozhranie komponentu. Komponenty sú stavané tak, aby sa dali znovu použiť a aby sa dali ľahko spojiť s inými komponentmi. Sú vyvíjané samostatne, sú navzájom nezávislé a mali by byť nezávislé na platforme.

Zároveň v rámci práce identifikujeme tieto základné vlastnosti komponentov: štruktúra (trieda, balík, knižnica, softvér, externá služba), synchronnosť (synchronný, asynchronný), uchovávanie stavu (stateless¹⁶, statefull¹⁷) a univerzálnosť (univerzálny¹⁸, neuniverzálny¹⁹).

Pre kompozíciu komponentov do väčších celkov väčšina výskumníkov a korporácií využíva ako lepidlo doménovo špecifický jazyk (odteraz DSL), jednou z mnohých inštancií DSL sú jazyky popisujúce architektúru (odteraz ADL). Medzi najrozšírenejšie ADL patria ArchiMate, ABACUS, ACME, ADML, ByADL a iné. Je však možné zadefinovať si aj vlastný ADL [36, 40] alebo rozširovať existujúce ADL [44].

Výhody jazykov DSL:

- Môžu mať aj grafickú podobu [21, 44].
- Znižujú veľkosť zdrojového kódu [3].
- Umožňujú aj netriviálne kompozície [3, 25].

Nevýhody jazykov DSL:

- Sú špecializované len na určitú doménu softvérov.
- Je nutné vedieť ich syntax.
- Je potrebné rozumieť kompozícii komponentov.
- Je nevyhnutné poznať komponenty, ktoré sú pomocou DSL dostupné.
- Ich vzájomná kombinácia je vo väčšine prípadov nemožná [18].

Medzi lepidlá sa tiež radia architektonické vzory, ktoré umožňujú korektne využívať už naprogramované komponenty [11, 19]. Ukážkou systémov, ktoré takýto vzor podporujú sú napríklad WS, CORBA, DCOM, EJB2. Pomocou takýchto architektonických vzorov sme schopní spojiť aj komponenty naprogramované úplne inými programovacími jazykmi z úplne iných oblastí. Využívanie už naprogramovaných komponentov výrazne uľahčuje vývoj softvéru [19].

V rámci práce sa venujeme jednému druhu kompozície:

¹⁶ Neuchováva si svoj stav.

¹⁷ Uchováva si svoj stav.

¹⁸ Dokáže ho využiť každý objektovo orientovaný jazyk. Napríklad REST api, SOA a iné.

¹⁹ Dokáže ho využiť len jazyk, v ktorom bol naprogramovaný. Napríklad trieda, balík, knižnica.

- Externý komponent-vyvíjaný softvér [4]: táto kompozícia vytvára interakcie medzi komponentom a vyvíjaným softvérom. Tieto interakcie pridávajú vyvíjanému softvéru funkcionality.

Navrhovaný prístup využíva pre kompozíciu komponentov lepidlo vo forme textu v prirodzenom jazyku. Text definuje požadovaný komponent aj s jeho konkrétnou funkcionality. Dôvod prečo vytvárame lepidlo vo forme textu v prirodzenom jazyku je priblíženie sa k jednému z cieľov EUSE. Vďaka takejto implementácii poskytujeme používateľovi voľnosť vyjadrenia sa a odbremeňujeme ho od znalosti nového špecifického jazyka a nutnosti vedieť, aké komponenty má k dispozícii.

Komponenty sa nemusia vždy nachádzať na internom úložisku, prístup môže využívať aj úložiská externé ako je napríklad maven²⁰, github²¹, google code²² alebo sourcerer²³. Avšak je nutné, aby mal každý komponent definované dodatočné informácie a konfiguračný súbor, ktorý v tomto prípade predstavuje rozhranie komponentu.

Prístup sa zameriava na kompozíciu stateless komponentov. Narábať so statefull komponentami dokáže len v prípade, že rozhrania týchto komponentov podporujú návrhový vzor singleton. Avšak implementácia vzoru singleton je rozličná už pri frameworkoch jedného programovacieho jazyka a jeho implementácia nie je vždy možná.

²⁰ <https://mvnrepository.com>

²¹ <https://github.com>

²² <https://code.google.com>

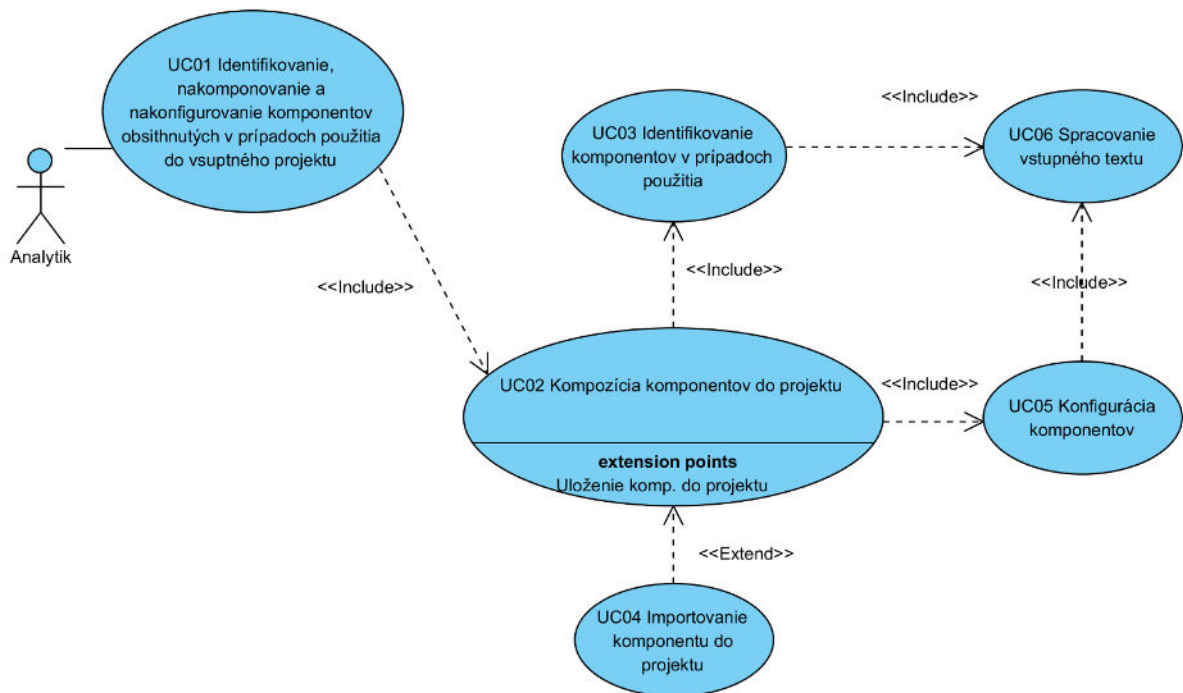
²³ <http://sourcerer.ics.uci.edu>

6 Identifikovanie, komponovanie a konfigurovanie komponentov

Navrhovaný prístup automatického komponovania softvéru pomocou textu v prirodzenom jazyku (odteraz iba prístup) uľahčuje prácu vývojárom softvéru. Dá sa využívať počas celého vývoja softvéru, avšak jeho najväčšie uplatnenie vidíme vo fáze prototypovania po definovaní systémových špecifikácií. Samotný prístup je zároveň vhodné spojiť s nástrojom, ktorý dokáže generovať používateľské rozhranie. Ich vzájomným spojením sa približujeme opäť o čosi bližšie k jednému z cieľov EUSE.

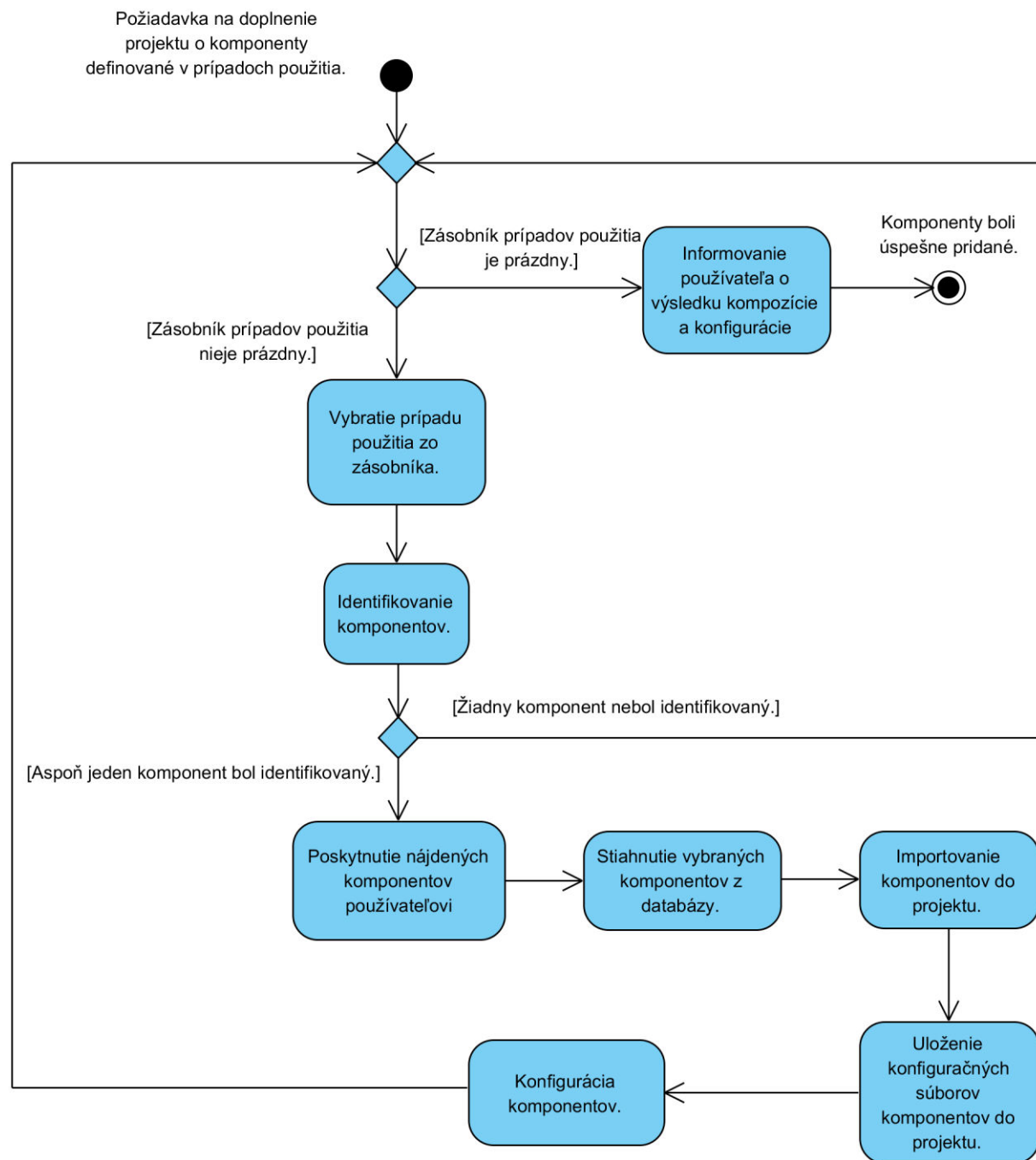
Cieľom prístupu je priblíženie sa k programovaniu pomocou textu tak, aby sme vedeli vďaka textu vytvoriť aj zložitejšie softvérové konštrukcie. Samotný prístup je založený na fakte, že existuje veľké množstvo naprogramovaných komponentov, ktoré sa dajú ľahko integrovať [19]. Práve vďaka týmto komponentom sme schopní vytvoriť komplikovanejšie kompozície s minimálnym úsilím. Komponenty sú identifikované a konfigurované na základe informácií obsiahnutých v systémovej špecifikácii a následne sú automaticky komponované do vstupného softvéru používateľa (prototypu). Súčasťou systémovej špecifikácie je aj nami definovaný artefakt *detailná špecifikácia*, ktorý slúži na lepšiu identifikáciu komponentov zo systémovej špecifikácie a zároveň na ich konfiguráciu.

Prístup je všeobecne opísaný diagramom prípadov použitia na obrázku 1 a aktivity diagramom na obrázku 2. Prístup je ďalej bližšie opísaný v nasledujúcich podkapitolách a jeho technická implementácia sa nachádza v Prílohe A.



Obrázok 1. Diagram prípadov použitia – Identifikácia, kompozícia a konfigurácia komponentov

Inicializácia hlavného toku začína v UC01. Na vstupe používateľ vloží so systémom vstupný softvér (prototyp) a prípady použitia opisujúce tento softvér.



Obrázok 2. Aktivita diagram – Doplnenie vstupného softvéru používateľa o externé komponenty

6.1 Rozšírenie prípadu použitia o nový artefakt – Detailnú špecifikáciu

Dôvody prečo sme z najčastejších notácií, ktoré sa využívajú pri definovaní systémových špecifikácií, vybrali práve prípady použitia, sa nachádzajú v kapitole 4.3. Samotný text prípadu použitia postačuje na to, aby sme dodatočné komponenty identifikovali. Avšak prípadu použitia chýba časť, v ktorej by boli dané komponenty došpecifikované o podrobnejšie informácie, ktoré sú nutné na ich konfiguráciu so vstupným softvérom. Preto navrhujeme pridať do prípadu použitia dodatočný artefakt s názvom *detailná špecifikácia* vo forme textu v prirodzenom jazyku, v ktorom budú dodefinované komponenty, ktoré sa v danom prípade použitia vyskytujú. Artefakt tak funguje ako lepidlo pri kompozícií.

Štruktúra nového artefaktu – Detailná špecifikácia:

Artefakt pozostáva z dvoch elementov:

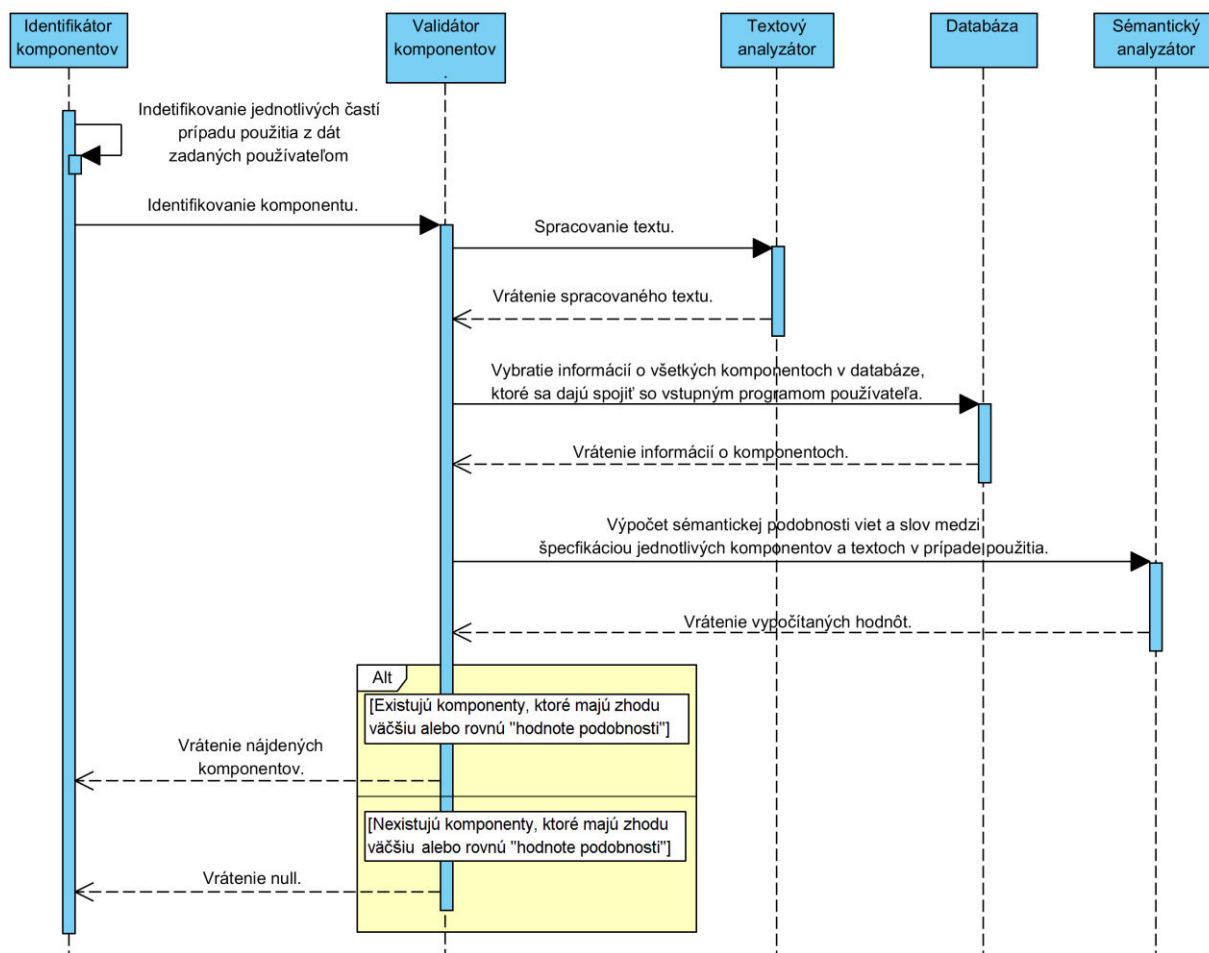
Detaily o komponente: Vety, ktoré špecifikujú komponent a funkcionality, ktorú plánuje používateľ využiť. Zároveň môže obsahovať informácie o konkrétnej konfigurácii komponentu. Používateľ nemusí o dostupných komponentoch nič vedieť, stačí intuitívna voľná špecifikácia.

Pozícia: Unikátny identifikátor umiestnenia volanej metódy komponentu v zdrojových kódach.

6.2 Identifikovanie komponentov v prípade použitia

Identifikácia komponentov z prípadu použitia je najdôležitejší proces celého prístupu. Lebo komponovať a konfigurovať môžeme len tie komponenty, ktoré boli predtým správne identifikované. Počas identifikácie sa prípady použitia vyhodnocujú samostatne. Identifikované komponenty sú následne ponúknuté používateľovi. To hlavne z dôvodu, že používateľ mohol nepresne definovať požadovaný komponent a tak sa v priebehu identifikácie komponentu identifikoval iný komponent ako používateľ požadoval. Prístup sa zároveň snaží identifikovať komponenty aj zo scenára a popisu prípadu použitia. Vďaka tomu sa nájdu aj komponenty, ktoré používateľ nedošpecifikoval v *detailnej špecifikácii*. Používateľ mohol na ich došpecifikovanie zabudnúť alebo si dané komponenty nevšimol.

Na obrázku 3 v sekvenčnom diagrame je zobrazené ako funguje identifikácia komponentov z prípadu použitia. Najdôležitejšími prvkami identifikácie komponentov je spracovanie textu a výpočet sémantickej podobnosti.



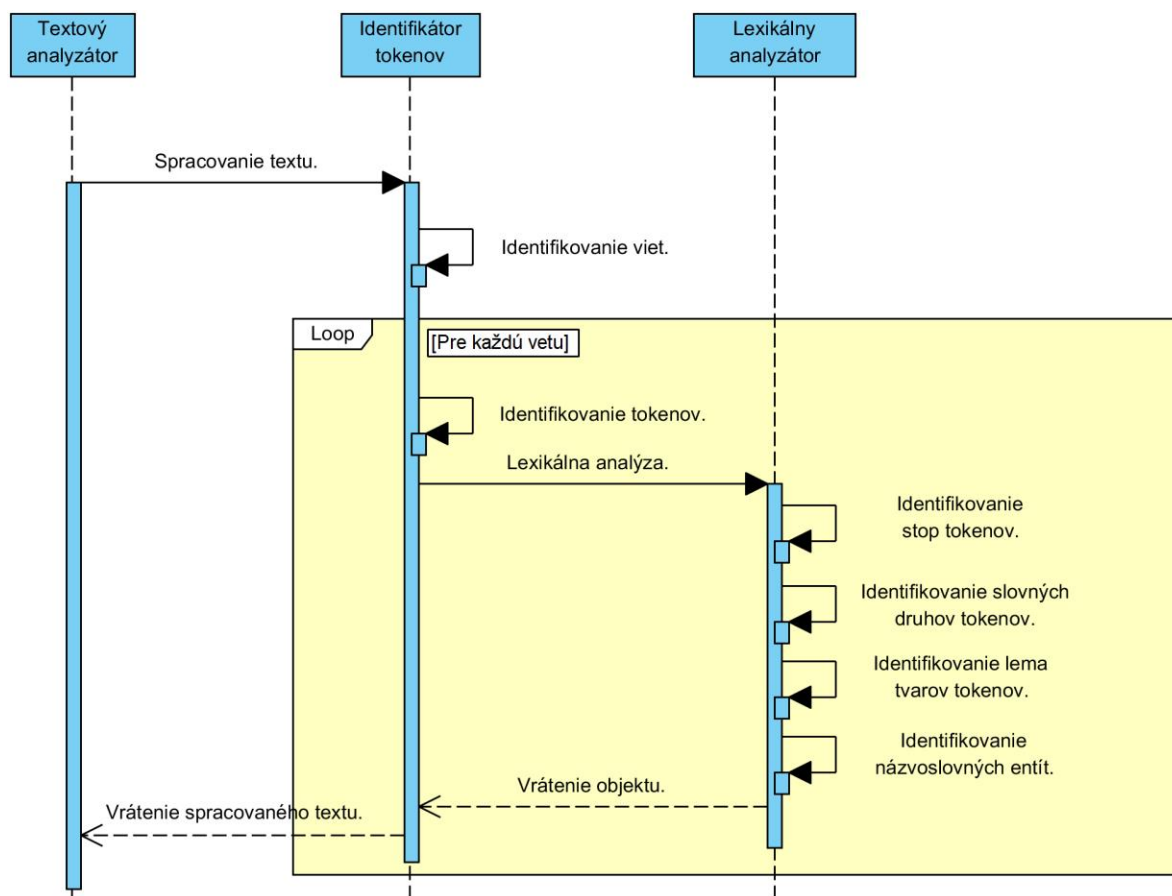
Obrázok 3. Sekvenčný diagram – Identifikovanie komponentu z textu.

K identifikácii komponentu z textu sa využívajú štyri základné komponenty.

1. Validátor komponentov – spája a vyhodnocuje dáta z jednotlivých komponentov.
2. Textový analyzátor – zodpovedá za spracovanie zadaného textu.
3. Databáza – uchováva komponenty a informácie o nich.
4. Sémantický analyzátor – zodpovedá za počítanie sémantickej podobnosti viet a slov medzi informáciami o komponentoch v databáze a spracovaným textom v prípade použitia.

6.2.1 Spracovanie textu

Texty v prípade použitia, nad ktorými sa bude počítat' sémantická podobnosť sa musia najprv pripraviť do adekvátneho formátu. Na obrázku 4 v sekvenčnom diagrame je znázornené spracovanie textu do podoby, ktorá je vhodná pre sémantickú analýzu.



Obrázok 4. Sekvenčný diagram – Spracovanie textu.

K spracovaniu textu sa využívajú dva základné komponenty.

1. Identifikátor tokenov – zodpovedá za identifikáciu viet a tokenov v texte.
2. Lexikálny analyzátor – zodpovedá za lexikálnu a syntaktickú analýzu.

6.2.2 Výpočet sémantickej podobnosti

Pre lepšie pochopenie sme výpočet sémantickej podobnosti dekomponovali do troch skupín.

1. Výpočet sémantickej podobnosti viet:

Pre výpočet sémantickej podobnosti viet sme si zadefinovali vlastný algoritmus. Náš algoritmus počíta percento sémantickej podobnosti dvoch viet ako priemer sémantických podobností slov vo vetách navzájom.

Bližšie opísanie algoritmu:

1. Vypočíta sa sémantická podobnosť lemy každého slova v základnej vete s lemov každého slova v porovnávanej vete. S tým, že lemy sa porovnávajú na základe konkrétnej metódy sémantickej podobnosti slov opísaných v kapitole 3.
2. V matici, ktorá vznikne v bode 1 sa identifikujú všetky hodnoty, ktoré sú rovné alebo väčšie ako 0. V matici sa totižto nachádzajú aj záporné čísla v prípade, že lemy sa nedali porovnať lebo neboli rovnakého slovného druhu.
3. Z hodnôt z bodu 2 sa vytvorí aritmetický priemer.
4. Percento sémantickej podobnosti je následne zvýšené o 15% v prípade, že porovnávajúca veta obsahuje aspoň jedno kľúčové slovo základnej vety.

2. Sémantická podobnosť textov:

Bližšie opísanie algoritmu:

1. Vypočíta sa sémantická podobnosť prvej vety v základnom texte s každou vetou v porovnávanom texte. Z týchto hodnôt sa vyberie tá najväčšia. Obdobne sa pokračuje pre ďalšie vety v základnom texte.
2. Vypočítame aritmetický priemer hodnôt z bodu 1.

3. Sémantická podobnosť medzi textami komponentov a textami prípadu použitia:

Pre zjednodušenie opisu nasledovného algoritmu predpokladáme, že v *detailnej špecifikácii* je definovaný práve jeden komponent.

Bližšie opísanie algoritmu:

1. Zo spracovaného prípadu použitia vyberieme jeho popis, scenár a *detailnú špecifikáciu*.
2. Vyberieme z databázy prvý komponent.
3. Vypočítame sémantickú podobnosť textov medzi:
 - 3.1. Popisom prípadu použitia a popisom komponentu.
 - 3.2. Popisom prípadu použitia a popisom metód komponentu.
 - 3.3. Scenárom prípadu použitia a popisom komponentu.
 - 3.4. Scenárom prípadu použitia a popisom metód komponentu.
 - 3.5. *Detailnou špecifikáciou* a popisom komponentu.
 - 3.6. *Detailnou špecifikáciou* a popisom metód komponentu.

4. Hodnoty z bodu 3 porovnáme s hodnotou podobnosti (viď. kapitola 6.2.3). V prípade, že aspoň jedna hodnota bola väčšia ako hodnota podobnosti, tak sa komponent v prípade použitia vyskytuje.
5. Kroky 2 až 4 opakujeme pre každý komponent v databáze.

Vďaka porovnaniam 3.1 až 3.4 dokážeme identifikovať aj komponenty, ktoré neboli zadefinované v *detailnej špecifikácii*. Tieto komponenty mohol používateľ zabudnúť došpecifikovať alebo si ich vôbec nevšimol. Zároveň dokážeme presne určiť, v ktorej časti prípadu použitia bol identifikovaný komponent definovaný. A na základe porovnania v bode 3.6 dokážeme identifikovať aj konkrétnu metódu komponentu, ktorá je v prípade použitia definovaná.

6.2.3 Definovanie hodnoty podobnosti

Hodnota podobnosti je číslo, ktoré nám určuje hranicu medzi tým, či sú dva texty sémanticky podobné. Táto hodnota sa využíva pri identifikovaní komponentu v prípade použitia (viď. kapitola 6.2.2). Pre výpočet tohto čísla sme vytvorili štúdiu, v ktorej sa zúčastnilo 6 participantov.

Priebeh štúdie bol nasledovný:

Každý participant dostal tri skupiny viet. Každá skupina obsahovala vety z inej domény. Participant mal vyhodnotiť každú skupinu samostatne. Jej vyhodnotenie spočívalo z významového (sémantického) porovnania prvej vety v danej skupine s ostatnými vetami v skupine.

Názorná ukážka jednej skupiny:

1. Mačka je domáce zviera.
2. Na ceste je pes. - vzhľadom na 1. vetu je to významovo *nepodobná* veta
3. Škrečok je domáce zviera. - vzhľadom na 1. vetu je to významovo *podobná* veta

Každá skupina obsahovala od 11 do 13 viet. V skupine sa nachádzali vety, ktoré náš prístup počítania sémantickej podobnosti viet (viď. kapitola 6.2.2) vyhodnotil s podobnosťou viac ako 85%, s podobnosťou od 40% do 85% aj vety s podobnosťou do 40%. Zadanie štúdie spolu s úplnými skupinami viet, ktoré dostali participanty k vyhodnoteniu sa nachádzajú v Prílohe B.

Vyhodnotenie štúdie:

Participanty dohromady spravili 216 porovnaní, z nich 146 vyhodnotili ako podobné vety. Skupiny viet sme vyhodnocovali samostatne. Zo skupiny sme zobrali vety, ktoré participant vyhodnotil ako podobné a vypočítali sme ich sémantickú podobnosť s prvou vetou v skupine.

Z týchto hodnôt sme vybrali dve najväčšie a spravili aritmetický priemer. Takto sme dostali subjektívny názor participanta, ktorý nám hovorí, od akej hodnoty sú si dve vety významovo podobné. Podobne sme vypočítali hodnoty pre všetky skupiny viet. Z týchto hodnôt sme spravili aritmetický priemer a tak sme dostali subjektívny názor skupiny participantov, ktorý v konečnom dôsledku môžeme považovať za objektívny.

Výsledkom štúdie je teda číslo, ktoré určuje hranicu, od ktorej si človek myslí, že dve vety sú si podobné. Hodnota podobnosti, ktorú sme dostali je rovná 72,5%. Číže náš prístup identifikuje z prípadu použitia len tie komponenty, ktoré budú mať zhodu rovnú alebo väčšiu ako 72,5%. Práve vďaka tejto hodnote sme schopní identifikovať komponenty tak, ako by ich identifikoval používateľ vo väčšine prípadov.

6.3 Kompozícia komponentu do projektu

Pod kompozíciou rozumieme importovanie komponentu do vstupného softvéru používateľa. Takúto kompozíciu je možné plne zautomatizovať a preto v rámci kompozície nie je nutný žiadny zásah od používateľa. Avšak importovanie komponentu nie je identické pre všetky programovacie jazyky a frameworky. Preto je nutné mať predom definované importovanie pre každý jazyk a framework zvlášť.

V prípade, že má komponent dostupné platformovo nezávislé rozhranie je možné ho skomponovať s ľubovoľným typom projektu. Na takúto kompozíciu je nutný len konfiguračný súbor prispôbený pre daný typ projektu. Napríklad, majme komponent implementovaný vo forme spustiteľného jar súboru. Tento komponent je platformovo nezávislý a tak sa dá skomponovať s ľubovoľným typom projektu. Používateľ má projekt, ktorý je implementovaný v jazyku Python a chce využiť náš komponent. Pre takúto kompozíciu je nutné, aby konfiguračný súbor tohto komponentu bol implementovaný v jazyku Python a tak poskytoval rozhranie, ktoré projekt zadáný používateľom vyžaduje. Z toho vyplýva, že platformovo nezávislý komponent môže mať implementovaných viacero konfiguračných súborov pre rozličné programovacie jazyky a frameworky. Pomocou tejto konfigurácie dosahujeme spojenie komponentov z rôznych oblastí.

Konfiguračné súbory importovaných komponentov a pri určitých štruktúrach komponentov aj komponenty samotné sa v projekte pridávajú na generické miesto a to z dôvodu náročného hľadania korektného umiestnenia týchto súborov.

6.4 Konfigurácia komponentov

Konfigurácia komponentov sa vykonáva priamo na základe informácií v *detailných špecifikáciách* v prípadoch použitia. Z toho vyplýva, že komponenty, ktoré nie sú týmto artefaktom dodefinované, nemôžu byť nakonfigurované.

Pod konfiguráciou sa v objektovo orientovanom systéme typicky rozumie nastavenie parametrov objektu alebo priradenie atribútov metóde. V rámci práce sa venujeme priradovaniu

atribútov metóde. Avšak pred samotným priradovaním atribútov je nutné najprv identifikovať metódu komponentu a následne identifikovať jej možné vstupné atribúty.

Identifikácia metódy komponentu prebehla počas identifikovania komponentu samotného (viď. kapitola 6.2.2). Spôsob volania identifikovanej metódy definuje konfiguračný súbor a za správne umiestnenie volania metódy v zdrojových kódach zodpovedá unikátny identifikátor, ktorý sa nachádza v *detailnej špecifikácii*. Po identifikovaní metódy a miesta v zdrojových kódach vstupného projektu sa začína identifikácia všetkých možných vstupných atribútov identifikovanej metódy. Tie sú identifikované z dvoch rozličných zdrojov:

1. Z *detailnej špecifikácie* v prípade použitia na základe identifikovania kľúčových slov, názvoslovných entít a ich kategórii. A následne ich sémantickej a syntaktickej podobnosti so vstupnými atribútmi metódy a ich synonymami.
2. Zo zdrojového súboru, do ktorého sa umiestilo volanie identifikovanej metódy, na základe sémantickej podobnosti slov medzi globálnymi premennými v danom súbore a vstupnými atribútmi metódy a ich synonymami.

V prípade, že sa nepodarilo korektne identifikovať nejaký vstupný atribút, to sa mohlo stať napríklad z dôvodu, že daný atribút nebol definovaný v *detailnej špecifikácii*, tak sa použijú defaultné hodnoty daného atribútu.

Spracovanie výstupu identifikovanej metódy ponechávame na používateľovi. To hlavne z dôvodu, že jej výstup môže byť veľmi rôznorodý a spôsob spracovania tohto výstupu nie je identický pre všetky programovacie jazyky.

6.5 Rozšírenie komponentov o dodatočnú časť

Aby bolo možné komponenty automaticky identifikovať, komponovať a konfigurovať, musia mať definovanú dodatočnú špecifikáciu. Bez tejto špecifikácie nie je možné automatické identifikovanie komponentu a tak ani jeho komponovanie a konfigurovanie s inými komponentami. Špecifikácia obsahuje:

Definovanie komponentu: Kľúčové slová a vety, ktoré vecne špecifikujú komponent.

Definovanie metód komponentu: Slová a vety, ktoré vecne špecifikujú jednotlivé metódy komponentu.

Miesto uloženia: Fyzická cesta ku komponentu. Môže to byť aj URL²⁴ pokiaľ sa nejedná o lokálny komponent.

Všeobecné informácie o komponente: Či sa jedná o univerzálny komponent, či má vlastné rozhranie, či je synchrónny, akú má formu.

²⁴ Uniform Resource Locator

Informácie o integrácii: Jazyky a frameworky, s ktorými sa dá komponent spojiť. A zároveň informácie o tom, ako sa dá daný komponent s týmito jazykmi a frameworkmi spojiť.

Konfiguračný súbor: Je to zdrojový kód, v ktorom sú implementované metódy, ktoré komponent poskytuje. Napríklad komponent *emailový klient* ako taký neobsahuje priamo implementovanú metódu *pošli email*, avšak obsahuje všetky metódy a prostriedky, ktoré takáto metóda vyžaduje. Takže pomocou komponentu *emailový klient* je programátor schopný naprogramovať metódu *pošli email*. Konfiguračný súbor uľahčuje prácu programátorovi, pretože obsahuje predimplementované metódy, ktoré je možné pomocou komponentu naprogramovať. V rámci uvedeného príkladu by konfiguračný súbor obsahoval metódu *pošli email*. Konfiguračný súbor sa tak správa ako rozhranie komponentu.

Konfiguračný súbor obsahuje okrem implementovaných metód aj spôsob volania jednotlivých metód, defaultné hodnoty pre vstupné atribúty a synonymá a vstupných atribútov - viď. hlavička metódy *sendSimpleEmail* na obrázku 5. Platformovo nezávislý komponent môže mať zadaných viacero konfiguračných súborov, pre každý programovací jazyk a framework, s ktorým ho je možné skomponovať zvlášť. Na obrázku 5 je zobrazená ukážka konfiguračného súboru v jazyku Java pre komponent *emailový klient*.

```
public class ComponentEmailClient {  
  
    /**  
     * Call: ComponentEmailClient.sendSimpleEmail( fromEmail , toEmail , host , subject , content );  
     * Default: fromEmail = fiit@gmail.com ; toEmail = stu@gmail.com ; host = localhost ; subject = Hello world ; content = Lorem...  
     */  
    * @param fromEmail from, email, mail, address, e-mail  
    * @param toEmail to, email, mail, address, e-mail  
    * @param host host, localhost, ip-address, ipaddress ; 192.168.0.1 : 0.0.0.0  
    * @param subject subject, topic, title, heading, caption  
    * @param content content, text, message  
    * @return True if everything goes without problems.  
    */  
    public static boolean sendSimpleEmail(String fromEmail, String toEmail, String host, String subject,  
        String content) {  
        Properties properties = System.getProperties();  
        properties.setProperty("mail.smtp.host", host);  
        Session session = Session.getDefaultInstance(properties);  
        try {  
            MimeMessage message = new MimeMessage(session);  
            message.setFrom(new InternetAddress(fromEmail));  
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(toEmail));  
            message.setSubject(subject);  
            message.setText(content);  
            Transport.send(message);  
            return true;  
        } catch (MessagingException e) {  
            e.printStackTrace();  
        }  
        return false;  
    }  
}
```

Obrázok 5. Ukážka konfiguračného súboru pre komponent *emailový klient*

Vďaka konfiguračným súborom sme schopní komponenty obsiahnuté v prípadoch použitia aj konfigurovať a tak pridať do projektu používateľa požadovanú funkcionality, opísanú danými prípadmi použitia, a nie len prostriedok, komponent, na jej implementáciu.

6.6 Príklad

Predpoklady:

Predpokladajme, že používateľ vytvára aplikáciu v jazyku Java, ktorej časť je opísaná nasledovným prípadom použitia:

Id: 1

Názov: Oboznámenie zákazníka o úspešnom dodaní požadovaného tovaru na predajňu.

Primárni aktéri: Systém.

Popis: Oboznámenie zákazníka emailom, že jeho tovar je pripravený na predajni.

Predpoklady: Tovar je pripravený na vyzdvihnutie.

Scenár:

1. Systém sa pripojí do databázy.
2. Systém vyhladá zákazníkov email na základe čísla objednávky.
3. Systém odošle zákazníkovi notifikačný email.

Z prípadu použitia je zrejmé, že aplikácia má obsahovať komponent, ktorý vie odosielať emaily. Používateľ dodefinuje tento komponent v *detailnej špecifikácii*:

Detaily o komponente: Mailový klient. Posielanie jednoduchých emailov. Email bude odoslaný z foo@ehop.com.

Pozícia: Marker1

V *detailnej špecifikácii* používateľ bližšie špecifikoval komponent, ktorý dokáže posilať emaily. Zároveň špecifikoval aj emailovú adresu, ktorá sa má používať pre odoslanie emailov. V elemente pozícia definoval unikátny Marker1. Marker teraz musí zadať aj do zdrojového kódu aplikácie. Marker ukazuje na konkrétne miesto v zdrojovom kóde aplikácie, kde sa má vykonať metóda definovaného komponentu.

Ďalej predpokladajme, že v našej databáze sa nachádza takýto komponent:

Názov: Emailový klient.

Popis: Posielanie emailov. Použitie SMTP servera pre odoslanie emailov. Posielanie skupiny emailov.

Kľúčové slová: Email, smtp, html, mail, spam, príloha, príjemca.

Metóda1: Odoslanie jednoduchého emailu. Odoslanie emailu s viacerými príjemcami. Odoslanie emailu z lokálneho smtp serveru.

Metóda2: Odoslanie emailu s html obsahom. Odoslanie emailu s pekným formátovaním.

Metóda3: Odoslanie emailu s prílohou.

Štruktúra: Knižnica.

Konfiguračný súbor: Rozhranie komponentu v jazyku Java. (Komponent má práve jeden konfiguračný súbor a ten je implementovaný v jazyku Java. Preto vieme, že daný komponent je možné spojiť s programom v jazyku Java.)

Kroky vedúce k doplneniu vstupného programu o externé komponenty:

- Používateľ spustí náš softvér a vloží do neho:
 - Cestu ku zdrojovým súborom jeho aplikácie - prototypu.
 - Cestu k prípadu použitia.
- Softvér identifikuje programovací jazyk, v ktorom je aplikácia používateľa napísaná.
- Softvér nájde v databáze všetky komponenty, ktoré je možné skomponovať so vstupnou aplikáciou používateľa.
- Softvér spracuje texty v prípade použitia.
- Softvér sa na základe sémantickej podobnosti snaží identifikovať komponenty aj s ich metódami v spracovanom texte prípadu použitia.
- Softvér ponúkne používateľovi identifikované komponenty aj s ich metódami zoradené od najväčšej zhody po najmenšiu.
- Používateľ buď vyberie správny komponent aj s jeho metódou (správny komponent nemusí byť ten, ktorý má najväčšiu zhodu – je prvý v poradí, viď. kapitola 6.2) alebo výber ponechá na softvéri, ten následne vyberie komponent s najväčšou zhodou.
- Softvér stiahne z databázy komponent aj s jeho konfiguračným súborom.
- Softvér uloží komponent a konfiguračný súbor na generické miesto.
- Softvér importuje komponent do aplikácie.
- Softvér nájde v aplikácii zdrojový súbor, v ktorom sa nachádza Marker1 (pozícia, ktorú definoval používateľ v *detailnej špecifikácii* v prípade použitia).

- Softvér zavolá metódu komponentu, ktorá bola identifikovaná v bode 5 na mieste, kde sa nachádza `Marker1`.
- Softvér nájde názvy globálnych premenných v zdrojovom súbore, v ktorom sa volá metóda komponentu.
- Softvér na základe sémantickej podobnosti slov porovná vstupné atribúty volanej metódy s nájdenými globálnymi premennými. V prípade zhody nahradí atribút za danú globálnu premennú.
- Softvér identifikuje v *detailnej špecifikácii* kľúčové slová, názvoslovné entity a ich kategórie. Na základe syntaktickej a sémantickej podobnosti slov porovná nájdené kľúčové slová, názvoslovné entity a ich kategórie so vstupnými atribútmi metódy. V prípade zhody nahradí atribút za kľúčové slovo / názvoslovnú entitu.

Záver:

Výsledkom je aplikácia zadaná používateľom na začiatku – prototyp, rozšírená o komponent emailový klient. Tento komponent je importovaný do prototypu, jeho metóda na odoslanie emailu je zavolaná v zdrojovom súbore na mieste `Markera1` a jeho vstupné atribúty sú nakonfigurované podľa informácii v *detailnej špecifikácii*, t.j. vstupný atribút *fromEmail* nahradený za *foo@eshop.com*.

6.7 Zosumarizovanie navrhovaného prístupu

Identifikácia a kompozícia komponentov nie sú priamo závislé na nami pridanom artefakte v prípade použitia - *detailnej špecifikácii*. Komponenty sa dajú identifikovať aj len na základe scenáru a popisu prípadu použitia a komponovanie komponentov v zmysle tejto práce sa dá plne zautomatizovať. Avšak *detailná špecifikácia* dopomáha identifikovaniu komponentov k dosahovaniu lepších výsledkov a je potrebná pre konfiguráciu komponentov, ktorá bez nej nie je možná.

Pre korektnú identifikáciu, kompozíciu a konfiguráciu komponentov do vstupného projektu je nutné, aby softvér obsluhoval používateľ, zručný s programovaním, analytik. Táto podmienka vzniká z dvoch základných dôvodov:

1. Bežný používateľ nie je schopný napísať korektné prípady použitia. To hlavne z dôvodu, že by nevedel zachytiť v diagramoch prípadov použitia všetky artefakty a spojenia, tak ako to určuje biznis plán. Zároveň však nepozná postup a princípy zápisu prípadov použitia a nepozná uml notácie.
2. Neskoršia úprava kódu vstupného projektu v prípade konfigurácie metód s návratovou hodnotou.

Ako vstup do softvéru, ktorý podporuje náš prístup, musí používateľ vložiť vstupný program (prototyp) a prípady použitia. Nami navrhovaný prístup, rozširuje vstupný program o komponenty, ktoré sú definované v prípadoch použitia. K tomu mu dopomáha nami pridaný artefakt v prípadoch použitia – *detailná špecifikácia*. Identifikácia, kompozícia a konfigurácia komponentov pozostáva zo:

1. Spracovania textov v prípadoch použitia.
2. Identifikovania komponentov a ich metód z prípadov použitia na základe sémantickej podobnosti medzi spracovaným textom prípadov použitia a informáciami o komponentoch v databáze.
3. Importovania komponentov do vstupného programu.
4. Nakonfigurovania komponentov podľa *detailnej špecifikácie*.

Softvér sa dá využívať počas celého vývoja softvéru, avšak jeho najväčšie uplatnenie vidíme vo fáze prototypovania po definovaní systémových špecifikácií. Samotný softvér je vhodné spojiť s nástrojom, ktorý dokáže generovať používateľské rozhranie. Takýmto spojením sa minimalizujú priame zásahy do zdrojových súborov programu. Navrhovaný prístup ponúka generické riešenie pre všetky programovacie jazyky a frameworky.

7 Testovanie

Navrhovaný prístup bol testovaný v troch kategóriách:

1. Identifikácia komponentov z prípadov použitia.
2. Identifikácia metód konfigurovaných komponentov.
3. Konfigurácia vstupných atribútov volaných metód.

Testovania sa zúčastnilo 15 študentov posledného ročníka inžinierskeho štúdia na Fakulte informatiky a informačných technológií v Bratislave. Každý z probantov mal napísať tri prípady použitia podľa zadania. V každom prípade použitia mal intuitívne identifikovať externé komponenty a opísať a nakonfigurovať ich v *detailných špecifikáciách*. Následne mal identifikované komponenty napojiť pomocou markerov, ktoré definoval v *detailných špecifikáciách*, do priloženého prototypu. A považovať nad deklaráciou globálnych premenných, ktoré by mohli byť využité ako vstupné atribúty do volaných metód identifikovaných komponentov. Probandi nemali žiadnu znalosť ani o jednom z komponentov v databáze a tak nevedeli aké komponenty budú testované, nepoznali ani metódy, ktoré sú poskytované týmito komponentami a ani aké vstupné atribúty dané metódy vyžadujú. Vďaka tomu externé komponenty identifikovali a opisovali intuitívne. Vypracované zadania boli vyhodnotené manuálne a výsledky boli následne spracované pomocou algoritmov macro precision²⁵, macro recall²⁶, macro F1 score²⁷. Kompletne testovacie zadanie sa nachádza v Prílohe C.

Testovacie zadanie bolo postavené tak, aby študenti v prípadoch použitia identifikovali nasledovné komponenty: emailový klient, emailový validátor, GPS konvertor - geokóder, konvertor mien, JDBC konektor do databázy (alebo objektovo relačný mapper). Uvedené komponenty boli následne testované.

Databáza obsahovala 12 komponentov, z nich sa 5 testovalo. Ostatných 7 komponentov v databáze bolo veľmi podobných tým testovacím. Tieto komponenty boli vybrané precízne so zámerom dokázania, že navrhovaný prístup počas fázy identifikovania komponentu nevyberá len prvý komponent z databázy, ale že dokáže požadovaný komponent identifikovať korektne. Konkrétne to boli nasledovné komponenty: SMTP server, Mapbox, Postgis jdbc driver, Paypal platobná brána, Jackson, rest api na predpoveď počasia. Konfiguračné súbory a popisy všetkých komponentov boli vytvorené a vložené do databázy manuálne.

²⁵ Presnosť

²⁶ Úspešnosť

²⁷ Harmonický priemer

Tabuľka 1. Dáta o probantoch a komponentoch

	Počet
Probanti	15
Vypracované prípady použitia	45
Komponenty v databáze	12
Testované komponenty	5

7.1 Identifikácia komponentov z prípadov použitia

Probanti v 45 prípadoch použitia spolu opísali 90 komponentov. Z toho 86 komponentov malo vytvorenú aj *detailnú špecifikáciu*. Ostatné 4 komponenty boli opísané iba v scenároch prípadov použitia. Tieto štyri komponenty si probanti nevšimli alebo zabudli opísať. Probanti boli schopní v ich vlastnoručne napísaných prípadoch použitia identifikovať a došpecifikovať až 95,55% komponentov. Z toho usudzujeme, že sa probanti stotožnili s novým elementom, *detailnou špecifikáciou*, v prípade použitia.

Spôsob identifikácie komponentov z prípadov použitia je opísaný v kapitole 6.2. Týmto spôsobom bolo správne identifikovaných 70,00% komponentov z celkového počtu 90 komponentov.

Tabuľka 2. Identifikácia komponentov 1

	Počet	Priama úmera
Opísané komponenty dohromady	90	-
Opísané komponenty aj v detailnej špecifikácii	86	95,55%
Opísané komponenty iba v scenári prípadu použitia	4	4,45%
Správne identifikované komponenty dohromady	63	70,00%
Správne identifikované komponenty z detailných špecifikácií	61	70,93%
Správne identifikované komponenty iba zo scenáru prípadu použitia	2	2,22%

Výsledky identifikácie komponentov z prípadov použitia sa nachádzajú v tabuľke 3. Predpokladáme, že algoritmus na identifikáciu komponentov sa dá ešte zlepšiť, návrhy na zlepšenie sú opísané v závere kapitoly 8.

Tabuľka 3. Identifikácia komponentov 2

	Recall	Precision	Priama úmera	F1 score
Správne identifikované komponenty zo všetkých (90)	68,51%	90,17%	70,00%	82,91%
Správne identifikované komponenty z tých, ktoré boli opísané v detailnej špecifikácii (86)	70,00%	90,17%	70,93%	83,93%

7.2 Identifikácia metód konfigurovaných komponentov

Konfigurovať sa dajú len tie komponenty, ktoré spĺňajú nasledovné podmienky:

1. Komponent má vytvorený konfiguračný súbor a má definované metódy, ktoré tento konfiguračný súbor sprostredkúva.
2. Komponent je definovaný pomocou *detailnej špecifikácie* a *detailná špecifikácia* má definovaný element *Pozícia*.

Z identifikovaných 63 komponentov bolo 38 vhodných na konfiguráciu, to tvorí 60,31%. Ostatných 25 komponentov bolo nevhodných, lebo nespĺňali obe vyššie spomenuté podmienky. Tieto komponenty buď neboli definované pomocou *detailnej špecifikácie* alebo *detailné špecifikácie* neobsahovali element *Pozícia* alebo komponenty nemali vytvorené konfiguračné súbory a tak ani definované metódy.

Tabuľka 4. Konfigurované komponenty

Počet opísaných komponentov dohromady	90
Počet identifikovaných komponentov	63
Počet komponentov vhodných na konfiguráciu	38
Počet % konfigurovaných komponentov z identifikovaných komponentov	60,31%
Počet % konfigurovaných komponentov zo všetkých komponentov	42,22%

Identifikácia metód prebieha obdobne ako identifikácia komponentu samotného. Správne identifikovaných metód konfigurovaných komponentov (38) bolo 76,31%, z celkového počtu komponentov (90) to tvorí 32,22%. Úspešnosť, presnosť a harmonický priemer sú v tabuľke 5 zhodné z dôvodu, že množina false positives sa rovná množine false negatives.

Tabuľka 5. Identifikácia metód konfigurovaných komponentov

	Recall	Precision	Priama úmera	F1 score
Správne identifikované metódy konfigurovaných komponentov (38)	75,71%	75,71%	76,31%	75,71%

7.3 Konfigurácia vstupných atribútov volaných metód

Robiť konfiguráciu vstupných atribútov má význam len pre konfigurované komponenty. To sú komponenty, ktoré boli korektne identifikované a zároveň bola korektne identifikovaná aj ich metóda. Takýchto komponentov sa v testovacej vzorke nachádza 29. Metódy týchto komponentov majú dohromady 102 vstupných atribútov.

Probanti mohli tieto atribúty nakonfigurovať dvomi spôsobmi:

1. Napísaním konkrétnej hodnoty do *detailnej špecifikácie*.
2. Vytvorením globálnej premennej v zdrojovom súbore, kde umiestnili unikátny marker.

Druhý spôsob mali k dispozícii z dôvodu imitovania vytvorenia prototypu. Za normálnych okolností by mohli takéto globálne premenné pridávať do zdrojových kódov prototypu pomocou prototypovacieho nástroja.

Probanti však nevedeli, aké všetky vstupné atribúty nimi definované metódy vyžadujú. Konfiguráciu vstupných atribútov tak robili len intuitívne. Prvým spôsobom nakonfigurovali 6 atribútov a druhým spôsobom 31 atribútov, to dohromady tvorí 36,27% atribútov. Z týchto dát je zrejmé, že probanti mali problém predstaviť si presnejšie volané metódy a zároveň sa nevedeli stotožniť s ich konfiguráciou pomocou prvého spôsobu.

Tabuľka 6. Konfigurácia vstupných atribútov 1

	Počet	Priama úmera
Konfigurované komponenty	29	-
Vstupné atribúty metód konfigurovaných komponentov	102	-
Definované vstupné atribúty pomocou <i>detailnej špecifikácie</i>	6	5,88%
Definované vstupné atribúty pomocou globálnych premenných	31	30,39%

Správne sa nakonfigurovalo 67,56% vstupných atribútov z definovaných vstupných atribútov (37), z celkového počtu vstupných atribútov (102) to tvorí 24,51%. Výsledky konfigurácie vstupných atribútov metód konfigurovaných komponentov sa nachádzajú v tabuľke 7. Avšak vzorka dát, z ktorej bola úspešnosť a presnosť počítaná je pomerne malá. To je z dôvodu, že

probandi mali problém predstaviť si presnejšie volané metódy a tak nevedeli aké vstupné atribúty by mali zadeklarovať. Preto tieto výsledky môžeme považovať len za orientačné.

Tabuľka 7. Konfigurácia vstupných atribútov 2

	Recall	Precision	Priama úmera	F1 score
Z definovaných (37)	73,15%	59,81%	67,56%	63,10%

7.4 Zhodnotenie testovania

Do testovania sa zapojilo 15 študentov inžinierskeho štúdia na STU FIIT. V rámci vypracovaných zadaní študenti vytvorili 45 prípadov použitia, ktoré slúžia ako základ pre testovanie navrhovaného prístupu. V prípadoch použitia dohromady opísali 90 komponentov a 86 z nich opísali aj v *detailnej špecifikácii*. Z toho usudzujeme, že študenti porozumeli zadaniu a stotožnili sa s novým elementom, *detailnou špecifikáciou*, v prípade použitia.

Navrhovaný prístup sa testoval v troch kategóriách, identifikácia komponentov z prípadov použitia, identifikácia metód konfigurovaných komponentov, konfigurácia vstupných atribútov volaných metód.

Prístup identifikoval komponenty z prípadov použitia s úspešnosťou 68,51%, presnosťou 90,17% a harmonickým priemerom 82,91%. Výsledky identifikovania komponentov sú priaznivé avšak stále vidíme priestor na vylepšenie algoritmu identifikácie komponentov a ich metód (viď. záver kapitoly 8).

Identifikácia metód je možná len z tých komponentov, ktoré sú vhodné na konfiguráciu. Medzi identifikovanými komponentami bolo takýchto komponentov 60,31%. Ostatných 39,69% komponentov bolo nevhodných z dôvodu nedostačujúcej špecifikácie. Identifikácia metód prebiehala obdobne ako identifikácia komponentov a dosahovala úspešnosť, presnosť aj harmonický priemer 75,71%. Výsledky sú zhodné z dôvodu identicky veľkých množín false positives a false negatives.

Konfigurácia vstupných atribútov volaných metód je možná len z konfigurovaných komponentov. Takýchto komponentov sa nachádza v testovacej vzorke 29 a metódy týchto komponentov majú dohromady 102 vstupných atribútov. Z týchto vstupných atribútov študenti intuitívne nakonfigurovali 36,27%. Z toho usudzujeme, že študenti mali problém si presnejšie predstaviť metódy komponentov. Výsledky konfigurácie vstupných atribútov sú nasledovné, úspešnosť 73,15%, presnosť 59,81% a harmonický priemer 63,10%. Avšak z dôvodu nízkeho počtu nakonfigurovaných vstupných atribútov od používateľov považujeme tieto výsledky iba za orientačné.

Kompozíciu identifikovaných komponentov do projektu sa podarilo plne zautomatizovať, vďaka tomu boli všetky identifikované komponenty korektne nakonponované. Celkovo z výsledkov testovania usudzujeme, že pomocou navrhovaného prístupu je možná korektná

identifikácia, konfigurácia aj kompozícia existujúcich softvérových komponentov do vyvíjaného softvéru. Uplatnenie prístupu tak má veľký potenciál vo vývoji softvéru v softvérovom inžinierstve.

7.4.1 Hrozby validity

Identifikovali sme nasledovné hrozby validity.

Interné:

Komponenty v databáze: Počas testovania sa nachádzalo v databáze približne dvakrát viacej komponentov ako sa testovalo. Väčšie množstvo komponentov v databáze by umožnilo väčšiu voľnosť vo výbere komponentu, či už používateľovi alebo algoritmu, ktorý identifikuje komponenty. To by mohlo mať dopad na výsledky testovania.

Testovacia skupina: Testovanie prebiehalo na študentoch inžinierskeho štúdia, ktorí majú skúsenosti v rôznych oblastiach vývoja softvéru, ako napríklad v programovaní, navrhovaní abstraktných riešení problémov, písaní korektných prípadov použitia a podobne. Zmena testovacej skupiny, by mohla mať dopad na výsledky testovania. Napríklad, keby sme do testovacej skupiny vybrali programátorov z praxe bez vysokoškolského vzdelania. Takýto tester by písali prípady použitia neodborne a na inej úrovni granularity, čo by mohlo ovplyvniť identifikáciu komponentov.

Externé:

Hawthornský efekt: Testovacia skupina vedela o tom, že je podrobená experimentu. Toto mohlo viesť k tzv. Hawthornskému efektu. Ten hovorí o tom, že pokiaľ si je testovaný objekt vedomý toho, že je podrobený testovaniu tak to môže zlepšovať jeho výkonnosť. Probandi tak mohli napísať testované prípady použitia lepšie a presnejšie ako za normálnych okolností, čo by mohlo mať dopad na výsledky testovania.

8 Zhodnotenie

EUSE už nie je doménovou oblasťou, ktorej sa venujú len ľudia zruční v informatike. V posledných rokoch vzniklo viacero technológií, pomocou ktorých je bežný používateľ schopný vytvoriť ním požadovaný softvér bez väčších problémov. Avšak tieto technológie sú obmedzené tým, že sú zamerané vždy len na jeden konkrétny typ softvéru. Zatiaľ neexistuje technológia, pomocou ktorej by bol používateľ, ktorý nemá skúsenosti s informatikou, schopný sám vytvoriť komplexnejší softvér z ľubovoľnej domény. Tomuto cieľu sa zatiaľ najviac približujú technológie, ktoré prekladajú špecifikáciu v prípadoch použitia na funkčný kód [17, 43]. Aby bol takýto preklad možný, musí text špecifikácie spĺňať rôzne podmienky ako napríklad použitie špecifického jazyka [43] alebo jednoduchosť zápisu [17]. Ale kód vygenerovaný takýmto spôsobom je málokedy automaticky funkčný a väčšinou nespĺňa to, čo bolo zadane v špecifikácii prípadu použitia na požadovanej úrovni. Dodatočná úprava takto vygenerovaného kódu sa vzdáva od softvérového inžinierstva pre koncových používateľov a zároveň takýmto spôsobom nie je možné generovať komponenty, ktoré vyžadujú komplexnejšiu špecifikáciu a to z dôvodu zatiaľ nedostačujúceho pokroku v oblasti spracovania prirodzeného jazyka [17].

V práci navrhujeme nový prístup automatického komponovania softvéru pomocou detailnej špecifikácie v prirodzenom jazyku. Zároveň prácou ukazujeme, že aktuálny stav v oblasti spracovania prirodzeného jazyka postačuje na automatickú identifikáciu, kompozíciu a konfiguráciu existujúcich softvérových komponentov. Detailná špecifikácia rozširuje prípady použitia a navrhovaný prístup z nej na základe sémantickej podobnosti textov automaticky identifikuje existujúce softvérové komponenty, ktoré sú rozšírené o nami pridané údaje. Následne identifikované komponenty automaticky komponuje a konfiguruje do vyvíjaného softvéru. Prístup tak umožňuje pomocou textu v prirodzenom jazyku tvorbu komplexného softvéru z ľubovoľnej domény, čím sa snaží naplniť vyššie spomenutý cieľ EUSE.

Navrhovaný prístup bol testový na 45 prípadoch použitia, ktoré boli vytvorené študentami inžinierskeho štúdia, v troch kategóriách:

Tabuľka 8. Sumarizácia výsledkov testovania

	Recall	Precision	F1 score
Identifikácia komponentov z prípadov použitia	68,51%	90,17%	82,91%
Identifikácia metód konfigurovaných komponentov	75,71%	75,71%	75,71%
Konfigurácia vstupných atribútov volaných metód	73,15%	59,81%	63,10%

Študenti v prípadoch použitia dohromady opísali 90 komponentov a 86 z nich opísali aj v *detailnej špecifikácii*. Z toho usudzujeme, že študenti porozumeli zadaniu a stotožnili sa s novým elementom, *detailnou špecifikáciou*, v prípade použitia. Výsledky testovania sú priaznivé avšak stále vidíme priestor na zlepšenie aktuálneho prístupu, napríklad vo vylepšení algoritmu identifikácie komponentov a ich metód (viď. záver kapitoly 8). Celkovo z výsledkov testovania usudzujeme, že pomocou navrhovaného prístupu je možná korektná identifikácia,

konfigurácia aj kompozícia existujúcich komponentov do vyvíjaného softvéru. Uplatnenie prístupu tak má veľký potenciál vo vývoji softvéru v softvérovom inžinierstve. Na záver opisujeme rôzne uskutočniteľné rozšírenia a vylepšenia nami navrhovaného prístupu:

- V práci viackrát hovoríme o využití nášho prístupu vo fáze prototypovania. Práve v tejto fáze vidíme jeho najväčší potenciál. Myslíme si, že dokáže túto fázu vývoja softvéru značne urýchliť. Pri vytváraní prototypov sa častokrát využívajú nástroje, pomocou ktorých vieme generovať používateľské prostredie. Tieto nástroje umožňujú tvorbu frontendu prototypu, avšak nedávajú mu žiadnu funkcionality. Funkcionalita je následne pridávaná manuálne samotnými programátormi a je pridaná len do takej miery, aby sme zákazníkom ukázali základnú myšlienku projektu v praxi. Za normálnych okolností je táto funkcionality imitovaná napríklad statickými dátami, kúzlíkom na pozadí alebo inými praktikami. Využitím nami navrhovaného prístupu vo fáze prototypovania sme schopní pridať do vygenerovaného frontendu prototypu cieľnú funkcionality vo forme komponentov automaticky. Vďaka tomu bude prototyp obsahovať viacej funkčných častí a čas na ich implementáciu bude značne menší. Zároveň spojením s nástrojom, ktorý vie generovať používateľské rozhranie minimalizujeme priame zásahy programátora do zdrojových kódov.
- V kapitole 2 sme predstavili niekoľko prístupov, ktoré pojednávajú o programovaní pomocou textu [17, 31, 32, 43]. Zároveň sme predložili fakty, z ktorých vyplýva, že automatické generovanie komplexnejšieho zdrojového kódu podľa špecifikácie zadanej vo forme textu v prirodzenom jazyku, s aktuálnym stavom v oblasti NLP zatiaľ nie je možné. Avšak na automatické vygenerovanie základných štruktúr zdrojových kódov softvéru aktuálny stav oblasti NLP postačuje [17, 43]. Myslíme si, že spojenie týchto prístupov [17, 43] s prístupom, ktorý opisujeme v tejto práci je možné. Náš prístup by najprv identifikoval komponenty, ktoré sa v prípade použitia nachádzajú. Následne by sa pomocou spomínaných prístupov vygeneroval ostatný zdrojový kód. Tento kód by sa doplnil o identifikované komponenty. A v poslednom, kroku by sa identifikované komponenty nakonfigurovali podľa *detailnej špecifikácie*. Takýmto spojením, by sa znížili a možno aj úplne odstránili nedostatky, ktoré tieto prístupy obsahujú.
- Nami definovaný prístup sa venuje len jednému druhu kompozície: externý komponent-vyvíjaný softvér (viď. kapitola 5). Avšak myslíme si, že pokiaľ by sa vhodne zadefinovali konfiguračné súbory jednotlivých komponentov, tak by bola možná aj kompozícia: externý komponent-externý komponent [2, 37]. Prínosom takejto kompozície je vytvorenie zložitejšej funkcionality, ktorá vedie k efektívnejšej práci s vytváraným softvérom a v konečnom dôsledku naplneniu požiadaviek klienta. Typický príklad takejto kompozície je spojenie komponentov JDBC a ORM, jedným z cieľov tejto konkrétnej kompozície je zjednodušenie vytvárania dopytov do databázy. Aby bolo možné vykonať takúto kompozíciu automaticky musel by existovať medzi týmito komponentami konfiguračný súbor, ktorý by prepájal ich funkcionality.

- K identifikovaniu miesta, kde sa zavolá funkcionálna prídavaného komponentu sa v našej implementácii využíva nami pridaný artefakt v prípade použitia. Avšak v prípade, že by sme boli schopní identifikovať zo scenáru prípadu použitia všetky akcie, ktoré sa v ňom nachádzajú a zároveň by sme vedeli týmto akciám priradiť ich miesto v zdrojovom kóde, tak by identifikácia tohto miesta bola možná priamo zo scenáru prípadu použitia. Napríklad jeden z krokov scenáru by bol nasledovný: „Po prihlásení používateľa načítaj z databázy jeho osobné informácie.“. V tomto kroku sa nachádza akcia „prihlásenie používateľa“ a komponent „databáza“. Zároveň vieme, že dáta sa majú z databázy vybrať po úspešnej akcii „prihlásenie používateľa“. Ak by sme teda vedeli priradiť akcii „prihlásenie používateľa“ jej miesto v kóde tak miesto, kde by sa vyberali osobné informácie používateľa z databázy, by sa nachádzalo hneď za úspešnou akciou „prihlásenie používateľa“.
- Pre sémantické porovnávanie textov sme navrhli vlastný algoritmus, ktorý využíva metódu WUP [45]. Algoritmus dosahuje dobré výsledky pri identifikácii komponentov z lokálnej databázy, kde sú tieto komponenty riadne opísané. Lenže komponenty, ktoré sa nachádzajú na externých databázach častokrát obsahujú žiadnu alebo len čiastočnú špecifikáciu. Vidíme priestor na vylepšenie nášho algoritmu tak, aby aj čiastočná špecifikácia komponentov postačovala na ich identifikáciu. Jedným z možných vylepšení daného algoritmu by bolo jeho spojenie s algoritmom cosine similarity.
- Konfiguračné súbory a popisy komponentov boli vytvorené a vložené do databázy manuálne. Takéto napĺňanie databázy je zdĺhavé a drahé na zdroje. Avšak určitá forma konfiguračných súborov sa nachádza na webových portáloch ako je www.github.com alebo www.stackoverflow.com. Zároveň sa na repozitároch komponentov ako je www.mvnrepository.com nachádzajú základné popisy komponentov. Preto vidíme priestor na vytvorenie inteligentného crawlera, ktorý bude schopný nájsť a spracovať popisy a konfiguračné súbory komponentov. A tak zautomatizovať proces napĺňania lokálnej databázy.

Literatúra

- [1] Ahmad, S.: Tutorial on Natural Language Processing. Artificial Intelligence, 2007, vol. 810, no. 161.
- [2] Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Proceedings of the 24th international conference on Software engineering, ACM, 2002, pp. 187–197.
- [3] Anlauff, M., Kutter, P.W., Pierantonio, A., Sünbül, A.: Using domain-specific languages for the realization of component composition. In: International Conference on Fundamental Approaches to Software Engineering, Springer, 2000, pp. 112–126.
- [4] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical concepts of component-based software engineering. Technical report, CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, 2000.
- [5] Banerjee, S., Pedersen, T.: Extended gloss overlaps as a measure of semantic relatedness. In: Ijcai. Volume 3., 2003, pp. 805–810.
- [6] Bird, S., Loper, E.: NLTK: the natural language toolkit. In: Proceedings of the ACL 2004 on Interactive poster and demonstration sessions, Association for Computational Linguistics, 2004, p. 31.
- [7] Bittner, K.: Use case modeling. 2002. ISBN 9780201709131.
- [8] Brown, A.W.: Large-Scale, Component-Based Development. 1 edn. 2000. ISBN 9780130887207.
- [9] Burnett, M.: What is end-user software engineering and why does it matter? In: International Symposium on End User Development, Springer, 2009, pp. 15–28.
- [10] Burnett, M.M., Myers, B.A.: Future of end-user software engineering: beyond the silos. In: Proceedings of the on Future of Software Engineering, ACM, 2014, pp. 201–211.
- [11] Cai, X., Lyu, M.R., Wong, K.F., Ko, R.: Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In: Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific, IEEE, 2000, pp. 372–379.
- [12] Cockburn, A.: Writing effective use cases. 1 edn. 2000. ISBN 9780201702255.

- [13] Crnkovic, I.: Component-based software engineeringnew challenges in software development. CIT. Journal of computing and information technology, 2003, vol. 11, no. 3, pp. 151–161.
- [14] Duong, P.H., Nguyen, H.T., Nguyen, V.P.: Evaluating semantic relatedness between concepts. In: Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication, ACM, 2016, p. 20.
- [15] Feng, J., Xie, F., Hu, X., Li, P., Cao, J., Wu, X.: Keyword extraction based on sequential pattern mining. In: proceedings of the third international conference on internet multimedia computing and service, ACM, 2011, pp. 34–38.
- [16] Fowler, M.: UML distilled: a brief guide to the standard object modeling language. 3 edn. 2004. ISBN 9780582894440.
- [17] Franců, J., Hnětynka, P.: Automated generation of implementation from textual system requirements. In: IFIP Central and East European Conference on Software Engineering Techniques, Springer, 2008, pp. 34–47.
- [18] Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: CASCON First Decade High Impact Papers, IBM Corp., 2010, pp. 159–173.
- [19] Gomez, J.M., Alor-Hernandez, G., Posada-Gomez, R., Rivera, I., Mencke, M., Chamizo, J., Sanchez, F.G., Toma, I.: An approach for component-based software composition. In: Electronics, Robotics and Automotive Mechanics Conference, 2008. CERMA'08, IEEE, 2008, pp. 195–200.
- [20] Hasselbring, W.: Component-based software engineering. Handbook of software engineering and knowledge engineering, 2002, vol. 2, pp. 289–305.
- [21] He, Y., Chen, X., Lin, G.: Composition of monitoring components for on-demand construction of runtime model based on model synthesis. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware, ACM, 2013, p. 20.
- [22] Hirst, G., St-Onge, D., et al.: Lexical chains as representations of context for the detection and correction of malapropisms. WordNet: An electronic lexical database, 1998, vol. 305, pp. 305–332.
- [23] Jacobson, I., Spence, I., Bittner, K.: Use Case 2.0: The guide to succeeding with use cases. Ivar Jacobson International, 2011, vol. 49, p. 50.
- [24] Jiang, J.J., Conrath, D.W.: Semantic similarity based on corpus statistics and lexical taxonomy. arXiv preprint cmp-lg/9709008, 1997.

- [25] Jifeng, H., Li, X., Liu, Z.: Component-based software engineering. In: International Colloquium on Theoretical Aspects of Computing. Volume 3722., Springer, 2005, pp. 70–95.
- [26] Jones, K.S.: Natural language processing: a historical review. University of Cambridge, 2001, pp. 2–10.
- [27] Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., et al.: The state of the art in end-user software engineering. ACM Computing Surveys (CSUR), 2011, vol. 43, no. 3, p. 21.
- [28] Lazzarini Lemos, O.A., Bajracharya, S.K., Ossher, J.: CodeGenie:: a tool for test-driven source code search. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, ACM, 2007, pp. 917–918.
- [29] Leacock, C., Chodorow, M.: Combining local context and WordNet similarity for word sense identification. WordNet: An electronic lexical database, 1998, vol. 49, no. 2, pp. 265–283.
- [30] Lin, D., et al.: An information-theoretic definition of similarity. In: Icml. Volume 98., 1998, pp. 296–304.
- [31] Little, G., Miller, R.C.: Keyword programming in Java. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, 2007, pp. 84–93.
- [32] Liu, H., Lieberman, H.: Metafor: visualizing stories as code. In: Proceedings of the 10th international conference on Intelligent user interfaces, ACM, 2005, pp. 305–307.
- [33] Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The Stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations, 2014, pp. 55–60.
- [34] Miller, G.A.: WordNet: a lexical database for English. Communications of the ACM, 1995, vol. 38, no. 11, pp. 39–41.
- [35] Nguyen, H.T.: Computing Semantic Similarity for Vietnamese Concepts Using Wikipedia. In: Some Current Advanced Researches on Information and Computer Science in Vietnam. Springer, 2015, pp. 91–106.
- [36] Nunn, I., Deugo, D.: Automated assembly of software components based on XML-coded instructions. In: Proceedings of the 2002 ACM symposium on Applied computing, ACM, 2002, pp. 937–942.

- [37] Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, 1998, pp. 177–186.
- [38] Patwardhan, S., Banerjee, S., Pedersen, T.: Using measures of semantic relatedness for word sense disambiguation. In: CICLing. Volume 2588., Springer, 2003, pp. 241–257.
- [39] Pedersen, T., Patwardhan, S., Michelizzi, J.: Word-Net:: Similarity: measuring the relatedness of concepts. In: Demonstration papers at HLT-NAACL 2004, Association for Computational Linguistics, 2004, pp. 38–41.
- [40] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E.: Knit: Component composition for systems software. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, USENIX Association, 2000, p. 24.
- [41] Resnik, P.: Using information content to evaluate semantic similarity in a taxonomy. arXiv preprint [arXiv preprint cmlg/9511007](https://arxiv.org/abs/1905.11007), 1995.
- [42] Stolee, K.T.: Finding suitable programs: Semantic search with incomplete and lightweight specifications. In: Software Engineering (ICSE), 2012 34th International Conference on, IEEE, 2012, pp. 1571–1574.
- [43] Śmiałek, M., Jarzębowski, N., Nowakowski, W.: Translation of use case scenarios to Java code. Computer Science, 2012, vol. 13, no. 4, pp. 35–52.
- [44] Voelter, M.: A family of languages for architecture description. In: 8th OOPSLA Workshop on Domain-Specific Modeling (DSM), 2008, pp. 86–93.
- [45] Wu, Z., Palmer, M.: Verbs semantics and lexical selection. In: Proceedings of the 32nd annual meeting on Association for Computational Linguistics, Association for Computational Linguistics, 1994, pp. 133–138.
- [46] Zesch, T., Gurevych, I.: Wisdom of crowds versus wisdom of linguists—measuring the semantic relatedness of words. Natural Language Engineering, 2010, vol. 16, no. 1, pp. 25–59.