Edícia výskumných textov
informatiky a informačných technológií

Odporúčanie pre softvérových inžinierov

Pavol Návrat, Miroslav Blšták, Michal Bystrický,
Tomáš Frťala, Ondrej Kaššák, Martin Konôpka,
Peter Laurinec, Marek Lóderer

Odporúčanie
pre softvérových inžinierov

Slovenská technická univerzita v Bratislave
2015

Edícia výskumných textov informatiky a informačných technológií

Odporúčanie pre softvérových inžinierov
Štúdie vybraných tém programových a informačných systémov (6)

Technickí redaktori: Ing. Martin Konôpka, Ing. Ondrej Kaššák, Ing. Miroslav Blšták
Grafika na obálke: Ing. Tomáš Frťala, Mgr. Alena Kovárová, PhD.
Návrh obálky: Ing. Peter Kaminský

Kniha vznikla a bola vydaná s podporou projektov Vedeckej grantovej agentúry Ministerstva školstva, vedy, výskumu a športu Slovenskej republiky a Slovenskej akadémie vied (VEGA):

- VG 1/1221/12 Pokročilé metódy v evolúcii softvéru: varianty, kompozícia a integrácia
- VG 1/0752/14. Inteligentná analýza veľkých údajových korpusov sémanticky-orientovanými a bio-inšpirovanými metódami v paralelnom prostredí.

Publikáciu podporili: GRATEX IT INŠTITÚT v rámci fondu GraFIIT.

Fakulta informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave
Ilkovičova 2, 842 16 Bratislava
http://www.fiit.stuba.sk/, pavol.navrat@stuba.sk

Vydala Slovenská technická univerzita v Bratislave
v Nakladateľstve STU, Bratislava, Vazovova 5.

ISBN 978-80-227-4495-9

PREDHOVOR

Táto knižka je výsledkom doktorandského seminára, ktorý som viedol v akademickom roku 2014/2015. Na Fakulte informatiky a informačných technológií máme šikovných študentov schopných naplniť aj náročné predstavy. Jednou takou predstavou je, aby zo seminára vznikol monotematický výskumný text, ktorý dopracujeme do podoby, pripravenej na tlač. V oblasti programových a informačných systémov sa takéto štúdie podarilo vydať už niekoľkokrát. Zatiaľ čo v prvom zväzku *Štúdií* sme podchytili seminár venovaný návrhovým vzorom a v druhom seminár venovaný webovej inteligencii, v treťom sa seminár sústreďoval na podstatu softvérovej architektúry a v štvrtom zväzku sme spracovali témy seminára, venovaného softvérovým paradigmám. Zatiaľ posledný, piaty zväzok sa venuje webovede, vznikajúcej vedeckej disciplíne, ktorá chce študovať web v rôznych aspektoch.

Tento v poradí už šiesty zväzok sa zameriava na odporúčanie v softvérovom inžinierstve. Metódy odporúčania informácií sa intenzívne študovali v predošlých zhruba dvadsiatich rokoch najmä v súvislosti s odporúčaním informácií na webe. V posledných rokoch dochádza k čoraz intenzívnejšiemu uvedomeniu, že tieto alebo podobné metódy môžu byť užitočné aj v softvérovom inžinierstve. Ide najmä o odporúčania, ktoré sa poskytnú softvérovému inžinierovi. Ukázalo sa, že je až prekvapujúco veľa možností, čo by bolo vhodné odporúčať niekomu, kto sa podieľa na vývoji softvéru. Druhou stránkou je pestrosť metód, ako odporúčania robiť.

Celú problematiku som rozdelil do trinástich častí medzi študentov seminára. Východiskovým literárnym zdrojom pre štúdium bol monografický zborník [1]. Po prednesení príspevkov a diskusii na seminári spracovali autori témy aj písomne. Prvotnú zodpovednosť za kapitoly sme si podelili takto: Blšták za kapitoly 6 a 7, Bystrický za kapitoly 10 a 12, Frťala za kapitoly 9 a 13, Kaššák za kapitoly 1 a 4 , Konôpka za kapitoly 3 a 11, Laurinec za kapitolu 8 a Lóderer za kapitoly 2 a 5. Spomenutý zborník sa ukázal byť v mnohom aj cennou inšpiráciou pri písaní, čo s vďakou priznávame. Autori však preštudovali množstvo ďalšej súčasnej vedeckej literatúry o príslušnej problematike, o čom svedčia aj zoznamy literatúry pripojené na koniec každej kapitoly. Vedomosti z nich získané tiež využili pri písaní textu.

Písanie som koordinoval a texty som aj (trochu) redakčne upravoval. Vytvorenie konečnej podoby si vyžadovalo aj mnoho ďalšej práce technického charakteru, podobne ako aj fungovanie seminára. Účastníci seminára si zaslúžia poďakovanie nielen za príspevky, ale aj za samosprávne fungovanie seminára a technickej prípravy tejto knižky. Obzvlášť chcem poďakovať Martinovi Konôpkovi.

O návrh titulnej grafiky som požiadal A. Kovárovú. Využila nápad, s ktorým prišli samotní autori, účastníci seminára. Za návrh aj za jej ochotu, s akou k veci pristúpila, jej veľmi ďakujem.

Dúfam, že táto knižka bude užitočná pre všetkých, ktorí sa chcú stať softvérovými inžiniermi alebo už ako softvéroví inžinieri pracujú a majú záujem o to, čo sa deje v jednom aktuálnom smere výskumu, ktorý môže priniesť významné inovácie do ich práce.

V Bratislave, november 2015 Pavol Návrat

Literatúra

[1] Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.): Recommendation Systems in Software Engineering. Springer, Heidelberg (2014).

Obsah

1 Odporúčacie systémy založené na zdrojovom kóde

Softvérové systémy je možné opísať pomocou rozličných artefaktov. Príkladmi sú zdrojový kód, komentáre, testy, či dokumentácia. Typicky najaktuálnejším, a teda aj najrelevantnejším zdrojom dát spomedzi artefaktov je práve samotný zdrojový kód. Ostatné artefakty sa typicky viažu na skôr vytvorený kód, a teda sú v najlepšom prípade rovnako aktuálne. Zdrojový kód poskytuje bohatý, a zároveň dobre štruktúrovaný zdroj informácií, na základe ktorého odporúčacie systémy dokážu vytvárať relevantné odporúčania. Odporúčače využívajúce zdrojový kód slúžia najmä na podporu vývojára pri práci s aplikačným rámcom alebo s API, poskytujú mu odporúčania a rady pre doplnenie chýbajúcich častí zdrojového kódu, pre opravu chýb, prípadne pomáhajú novým vývojárom naučiť sa orientovať v kóde projektu, programovacej paradigme či jazyku. Kapitola opisuje dôležité rozhodnutia potrebné urobiť pri návrhu a vývoji odporúčacieho systému založeného na zdrojovom kóde.

Odporúčacie systémy v softvérovom inžinierstve – OSSI (angl. Recommendation systems in software engineering – RSSEs) sú vo všeobecnosti softvérové nástroje, ktoré pomáhajú vývojárom v širokom spektre aktivít ako znovupoužitie zdrojového kódu či písanie efektívnych hlásení o chybách [20]. Kapitola sa zameriava na odporúčacie systémy založené na zdrojovom kóde (OSZnZK), teda odporúčacie systémy produkujúce výsledné odporúčania na základe analýzy zdrojového kódu softvérového systému a pridružených artefaktov. Tieto systémy sa zameriavajú na podporu korektného používania programovacích rozhraní (API) [8, 16, 25], knižníc, aplikačných rámcov [4], tvorbu nových metód, opravu či dokumentáciu chýb a na znovupoužiteľnosť

kódu. V kapitole okrem princípov uvádzame konkrétne príklady existujúcich odporúčacích systémov založených na zdrojovom kóde.

1.1 Fázy budovania systému

Proces vývoja odporúčacieho systému založeného na zdrojovom kóde zahŕňa sériu dôležitých rozhodnutí, ktoré spoločne formujú vlastnosti budúceho systému. Pokiaľ rozhodnutia rozdelíme v závislosti od cieľa, na ktorý sa zameriavajú, dostaneme dve jasne odlíšiteľné skupiny. Do prvej zaraďujeme rozhodnutia týkajúce sa spôsobu odporúčania [2, 21, 22, 23], do druhej rozhodnutia týkajúce sa interakcie systému s používateľom [18].

Obe tieto skupiny spoločne podliehajú štyrom štandardným fázam vývoja akéhokoľvek softvéru, teda analýze, návrhu, implementácii a následne validácii procesu. Tieto fázy prebiehajú sekvenčne, pričom v každom kroku sú postupne uvažované rozhodnutia z oboch skupín.

Vizualizácia procesu ako celku je znázornená v Tabuľka 1.1, riadky predstavujú skupiny rozhodnutí a stĺpce fázy vývoja odporúčacieho systému. Konkrétne kroky, teda bunky tabuliek sú chronologicky označené a. – h. Jednotlivé kroky podrobne rozpisujeme.

Tabuľka 1.1.Typy rozhodnutí realizovaných počas vývoja OSZnZK.

	Analýza	**Návrh**	**Implementácia**	**Validácia**
Odporúčací prístup	a. Zámer	c. Korpus	e. Metóda	g. Podpora
Interakcia s používateľom	b. IČP	d. Všeobecný V/V	f. Detailný V/V	h. Interakcia

1.1.1 Zámer

Prvé rozhodnutie realizované v rámci tvorby odporúčacieho systému založeného na zdrojovom kóde sa týka analýzy zámeru vytváraného systému. Konkrétne nás zaujíma, pre koho presne systém vytvárame, teda či bude cieľovým používateľom vývojár, tester, tvorca dokumentácie či výskumník [9]. Dôležité je taktiež stanoviť si, či máme primárne záujem odporúčať novým, neskúseným používateľom alebo chceme naopak podporiť skôr používanie metód, ktoré bežný vývojár neovláda a teda odporúčať skôr na úrovni experta.

Ďalšou dôležitou otázkou je, prečo chceme odporúčať a čo bude výsledkom odporúčania. Tu si treba premyslieť, či chceme podporiť napr. znovupoužitie existujúcich metód systému, a zabrániť tak redundancii kódu, či chceme odporúčať externé knižnice vhodné na riešenie aktuálnej úlohy používateľa, prípadne či máme záujem novému používateľovi odporúčať súvisiace časti zdrojového kódu, aby sa rýchlejšie oboznámil so systémom. Ďalšími typickými zámermi je správne použitie API, knižníc či aplikačných rámcov, podpora opravy zdrojového kódu alebo zmena zdrojového kódu [7].

Treťou významnou otázkou, ktorú je vo fáze zámeru nutné zodpovedať, je stanovenie vhodnej doby, kedy odporúčať. Teda rozhodnúť sa napr. medzi automatickým odporúčaním,

ktoré sa generuje priebežne pri písaní zdrojového kódu a medzi odporúčaním, ktoré sa vykoná až v momente, keď oň používateľ explicitne požiada. Tiež je potrebné rozhodnúť, či sa odporúčanie vytvorí pridaním nových informácií, konkrétnym návrhom metódy na použitie, alebo napr. ukážkou podobnej existujúcej funkcionality.

1.1.2 Interakcia človeka s počítačom

Druhé rozhodnutie sa robí taktiež vo fáze analýzy, avšak tu nás zaujíma spôsob interakcie používateľa so systémom. Pri type odporúčača rozhodujeme typicky medzi zásuvným modulom do IDE, prípadne samostatnou či konzolovou aplikáciou. Na základe predchádzajúcej fázy rozhodovania taktiež musíme prispôsobiť spôsob interakcie zvolenému typu odporúčača. Typicky môže ísť o vyhľadávač, radcu, či validátor.

V tejto fáze rozhodujeme tiež o miere zapojenia používateľa, teda, či mu budeme odporúčať automaticky alebo ako reakciu na ručne zadaný vstupný dopyt. V prípade ručného zadania rozhodujeme tiež o tom, čo všetko budeme pokladať za dopyt vyvolávajúci odporúčanie [5, 11].

1.1.3 Korpus

Po analýze možností poskytovanej funkcionality systému v doméne, a spôsobu interakcie s ním, nasleduje fáza návrhu. Vyberáme, ktoré z možností najlepšie vyhovujú konkrétnym požiadavkám.

Z pohľadu návrhu odporúčacieho systému nás primárne zaujíma dátový korpus. Tu je jednak potrebné vybrať zdroj dát, ktoré sa budú využívať pri modelovaní znalostí používateľa a pri následnom odporúčaní. Zvyčajne ním je zdrojový kód, pomocou ktorého vieme najlepšie identifikovať oblasť záujmu používateľa a mieru jeho znalostí. Po identifikácii „čo“ použiť nás zaujíma „ako“ to získať a priebežne zberať. Tu sa teda rozhodujeme, či nás budú zaujímať konkrétne modifikované riadky v zdrojovom kóde, prípadne či budeme zbierať záznamy o interakcii na úrovni metód, knižníc alebo podobne.

Po návrhu toho, aké dáta budeme zbierať z vybraného dátového korpusu, je taktiež potrebné rozhodnúť sa, či budeme brať do úvahy dáta výhradne z aktuálne vyvíjaného systému alebo či budeme napr. uvažovať tiež znalosti daného používateľa získané pri vývoji predchádzajúcich softvérových systémov v minulosti.

Okrem výberu primárneho korpusu slúžiaceho ako zdroj dát pre modelovanie a odporúčanie je možné využívať tiež doplnkové informácie. Typickými príkladmi sú stopy vykonania programu [1], záznamy o vývoji a zmenách zdrojového kódu v čase zachytené systémami na riadenie verzií zdrojového kódu [1, 6, 7, 26], záznamy o chybách [1, 6], ďalej komunikácia medzi vývojármi [6] či história správania používateľa.

1.1.4 Všeobecný vstup-výstup

Z pohľadu interakcie človeka s počítačom vo fáze návrhu riešime najmä výber a upresnenie spôsobu komunikácie používateľa so systémom. Určujeme tu, či bude odporúčací systém odpovedať na akcie používateľa automaticky alebo či bude čakať na explicitnú požiadavku. V prípade au-

tomatizovanej odpovede treba určiť tiež, čo bude spúšťačom odporúčania. Príkladom môže byť určitý konkrétny čas či akcia používateľa.

Ak je spúšťačom odporúčania čas, typicky ide o danú časovú jednotku, kedy sa pravidelne generuje odporúčanie. V závislosti od systému môže ísť napr. o odporúčanie každých niekoľko minút ak odporúčame knižnicu, ktorá by používateľovi mohla uľahčiť jeho aktuálnu úlohu, alebo napr. o odporúčanie generované raz denne, ak ide o návrh dokumentácie k triedam či knižniciam, pri ktorých používateľ dlhodobo robí chyby.

Inou možnosťou je zobrazenie odporúčania po tom, čo používateľ vykonal nejakú akciu. Tu však nemusí ísť len o odporúčanie na základe používateľovej explicitnej požiadavky. Spúšťačom môže rovnako dobre byť používateľovo správanie, v ktorom systém identifikuje potrebu odporúčania, napr. používateľ začne výrazne *posúvať* (angl. scrolling) po zdrojovom kóde, pravdepodobne je stratený a odporúčanie má predpoklad mu pomôcť. Podobne ak začne písať duplicitný kód, ktorý už systém obsahuje.

Ďalšou nezanedbateľnou súčasťou opisovaného kroku návrhu odporúčacieho systému je zodpovedanie otázky, či systém dokáže identifikovať položky vhodné na odporúčanie v čase, kedy vznikne potreba ich zobraziť, alebo či systém obsahuje zložitejší algoritmus odporúčania, ktorý odporúčanie počíta dlhšie ako rádovo pár milisekúnd, čím vyžaduje skoršie predpočítanie.

V tomto kroku tiež navrhujeme spôsob prezentácie odporúčania. Typickými možnosťami sú textová reprezentácia [4, 12, 14, 15, 16], linka smerujúca na cieľ odporúčania, zvýraznenie dôležitých častí systému – napr. vybranej časti zdrojového kódu, či grafický zápis – napr. diagram vysvetľujúci vlastnosť vyvíjaného systému.

Poslednou otázkou, ktorú treba v rámci tejto fázy vývoja softvérového systému vyriešiť, je samotný predmet odporúčania, napr. softvérové artefakty [1, 6], entity zdrojového kódu [4, 14], doplnený zdrojový kód [15] alebo mapovanie entít [5, 7].

1.1.5 Metóda

Vo fáze implementácie sa zaoberáme pri vývoji odporúčacieho prístupu samotnou metódou odporúčania. Presnejšie vyjasnením implementačných detailov metódy. V tejto fáze môžeme špecifikovať granularitu zbieraných dát. Kvalitné odporúčanie totiž vyžaduje dostatočnú presnosť, na druhej strane je však nutné vyhnúť sa prílišnému zahlteniu modelu používateľa detailmi. Zvyčajne sa preto dáta zbierajú na úrovni metód [4, 12, 17, 24] a relácií medzi nimi [3, 8, 14, 15, 16], teda komentárov, súborov, vstupných informácií [8, 12, 16] či zmien zdrojového kódu.

V tomto kroku sa musíme rozhodnúť, aký typ analýzy v rámci vytváraného odporúčacieho systému aplikujeme, napr. či využijeme textovú, lexikálnu, syntaktickú alebo sémantickú analýzu.

Rovnako musíme stanoviť požiadavky na programovaciu paradigmu, ktorú pri vývoji metódy využijeme, aký jazyk zvolíme, či akým spôsobom budeme analyzovať dáta. Pri analýze zvažujeme techniku, jej cieľ – teda položky, používateľov, vlastnosti prípadne ich rozličné kombinácie. Okrem analýzy treba používané dáta taktiež prefiltrovať, teda vyčistiť. Uvažujeme prefiltráciu, teda nájdenie entít, ich kontextu a ich spôsob čistenia. Uvažujeme aj postfiltráciu, čiže

výber podmnožiny relevantných dát z prvej fázy, určenie ich podobnosti, frekvencie či skóre z hľadiska vhodnosti odporúčania [4, 6, 12, 17, 24].

1.1.6 Detailný vstup/výstup

Z hľadiska interakcie používateľa s odporúčacím systémom nás vo fáze implementácie zaujíma primárne zvolenie konkrétnych vstupov a výstupov metódy.

Zvolenie konkrétneho vstupu sa skladá z voľby spôsobu, akým z vyhľadávacieho dopytu používateľa vyberieme entity zdrojového kódu, ktoré sú relevantné pre používateľovu aktuálnu úlohu a ktoré sa následne dajú dobre použiť pre samotné odporúčanie.

Voľbu konkrétneho výstupu chápeme tak, že nás zaujíma konkrétny jednotlivý výsledok. Teda neodporúčame všetky potenciálne vhodné položky, ale len niekoľko najrelevantnejších. Takto sa vyhneme hrozbe zahltenia používateľa informáciami, kedy by výsledky odporúčania v nevhodne zvolenom množstve mohli spôsobovať ešte väčšie zahltenie, ako bez ich použitia. Riskujeme síce vynechanie niektorého z relevantných výsledkov, avšak vyplývajúce výhody sú výrazne prijateľnejšie. Do množiny výsledkov totiž zahrnieme skôr relevantné odporúčania. Tu v praxi riešime skôr problémy s prioritizáciou položiek ako so zahltením používateľa.

1.1.7 Podpora

Po dokončení implementácie odporúčacieho systému nasleduje štandardne validácia vytvoreného produktu. Pri výbere spôsobu overenia jeho vlastností nás z hľadiska samotného odporúčacieho systému zaujíma prínos používateľovi, ako mu nejakým spôsobom napomohol pri riešení úloh.

V prvom rade v tomto kroku hľadáme samotný spôsob overenia, ktorým vieme kvantifikovať, ako dobre systém vykonáva určenú úlohu. Na to sa typicky využívajú prípadové štúdie [11, 15, 19] či porovnania s inými systémami [3, 7, 11, 12, 13], respektíve voči situácii bez odporúčača [12]. Taktiež je možné využiť A/B testovanie, simulácie použitia systému či kontrolované experimenty so závislými a nezávislými premennými.

Po identifikovaní spôsobu overenia meriame rozličné vlastnosti, pomocou ktorých stanovujeme kvalitu odporúčania [4, 15, 24]. Meriame a sledujeme užitočnosť odporúčania, teda mieru, akou rada prispela k vyriešeniu úlohy [6, 12, 16]. S tým súvisí tiež často meraná schopnosť dokončiť úlohu na základe odporúčania a taktiež rýchlosť vyriešenia úlohy [16]. Odporúčanie však môže napomáhať aj iným spôsobom a preto je v určitých prípadoch vhodné merať skôr mieru intelektuálnej námahy používateľa, kedy odporúčací systém používateľovi pomáha znížiť rozsah námahy nutnej pre dokončenie úlohy.

Vyriešenie úlohy sa vo väčšine prípadov dá dosiahnuť rozličnými spôsobmi a rovnako finálne riešenie sa môže značne líšiť. Riešenia tiež môžu dosahovať rozdielnu mieru kvality či správnosti. Oprava chyby môže spôsobiť chybné správanie v inej situácii, prípadne môže neúmerne spomaliť výkon systému. Preto je v istých situáciách vhodné validovať mieru správnosti riešení dosiahnutých na základe odporúčaní. Okrem správnosti je možné merať tiež relevanciu odporúčania. V tomto prípade však ide o subjektívnu metriku, ktorá sa nedá vyhodnotiť kvantitatívne.

Štandardnými kvantitatívnymi metrikami, ktoré sú pri validácii odporúčacích systémov široko používané a to dokonca aj mimo domény OSSI, sú pokrytie, presnosť a ich kombinácia zvaná F-skóre [6, 17, 24]. Pomocou pokrytia je možné jednoznačne identifikovať, ako komplexne odporúčanie pokrýva množinu existujúcich riešení. To znamená zistiť, akú časť z možných riešení dokáže systém pre používateľa nájsť a odporučiť. Presnosť vyjadruje mieru relevantných odporúčaní voči všetkým odporučeným. To znamená, že metrika hovorí o tom, koľko z odporúčaní bolo v danej chvíli vhodných. Pokrytie a presnosť sa v praxi javia ako nepriamoúmerné. Ak jedna rastie, druhá klesá. Z tohto dôvodu bola zavedená metrika F-skóre, ktorá vyjadruje pomer medzi nimi a dokáže presne vyjadriť, ktoré nastavenie systému poskytuje lepší výsledok – teda vhodnejšiu kombináciu dosahovaných výsledkov oboch metrík.

1.1.8 Interakcia

Validácia vytvoreného odporúčacieho systému sa z pohľadu jeho interakcie s používateľmi robí pomocou rozličných typov interakcie. Typ zvolený pre konkrétny odporúčací systém závisí od zamerania používateľov daného systému.

V prípade, že sú používateľmi systému vývojári, sledujeme podporu ich práce pri vývoji softvéru, čas odozvy odporúčača [1, 5, 13], mieru náročnosti jeho použitia (preferovaná je čo najnižšia), či stručnosť a výstižnosť výsledkov [12, 16, 19].

Druhou a značne odlišnou skupinou cieľových používateľov sú výskumníci. Tu nás zaujímajú najmä možnosti porovnania s inými riešeniami – teda inými odporúčačmi. Dôležité vlastnosti interakcie so systémom, ktoré meriame nezávisle od typu jeho používateľov sú najmä dostupnosti systému ako takého a tiež dát, s ktorými pracuje. V prípade systému (vyvíjaného, nie odporúčača) nás zaujímajú jeho softvérové artefakty, napr. či majú používatelia k dispozícii celý jeho zdrojový kód alebo len jeho časť, či len binárne súbory [7, 8, 11, 19]. Tiež je dôležité, či majú k dispozícii doplňujúce informácie, napr. opis systému, dokumentáciu alebo vôbec nič. Z pohľadu dostupnosti dát rozlišujeme najmä, či majú používatelia k dispozícii úplné dáta alebo len výsledky [14]. Na základe dát sa totiž dajú robiť ďalšie experimenty a sledovať nové vlastnosti systému. Hotové výsledky sú na ďalšiu prácu zväčša použiteľné v značne obmedzenejšej miere.

1.2 Existujúce systémy

V predchádzajúcej podkapitole sme opísali 8 oblastí otázok, ktoré treba pri vývoji odporúčacieho systému založeného na zdrojovom kóde uvažovať. V tejto podkapitole prinášame prehľad existujúcich odporúčacích systémov z domény, pričom pre jednotlivé systémy opisujeme spôsob, akým odporúčajú a rovnako ako riešia ďalšie vyššie opisované rozhodnutia, napr. nás zaujíma ich účel a vlastnosti. Opísané odporúčacie systémy spoločne pokrývajú v praxi používané prístupy k návrhu takýchto systémov v doméne odporúčania založeného na zdrojovom kóde.

1.2.1 RASCAL

Prvý odporúčací systém [17], ktorý predstavujeme, sa zameriava na predikciu nasledujúcej metódy, ktorú môže vývojár najbližšie pri vývoji novej časti zdrojového kódu zavolať. Systém využíva tradičnú metódu kolaboratívneho odporúčania [10] vychádzajúcu z predpokladu, že triedy sa môžu zhlukovať podľa podobnosti volaných metód. Voči tradičnému kolaboratívnemu prístupu sú v tomto prípade používatelia nahradení triedami a položky metódami. Metóda sa zakladá na hľadaní triedy volajúcej podobné metódy ako používateľom aktuálne vyvíjaná trieda.

Odporúčač RASCAL sa delí na 4 časti, ktoré sa postupne využívajú v jednotlivých fázach procesu odporúčania. Prvá časť zvaná *aktívny používateľ* slúži na identifikáciu triedy, ktorú používateľ aktuálne vyvíja. Druhá časť nazývaná *zberač histórie použitia* slúži na automatické dolovanie záznamov o vývoji tried a ich metód. V tejto fáze sa vytvára *matica využívania metód triedou* a *usporiadaný zoznam metód využívaných triedou* (Obrázok 1.1) pre všetky triedy systému, nad ktorým sa odporúča. Každá bunka v matici reprezentuje počet volaní danej metódy triedou.

Triedy (v roli používateľov CF)	**Metódy (v roli položiek CF)**			
	SetX()	SetY()	Copy()	Display()
BookingGUI	2	0	1	3
RemoteDB	1	0	2	1
CompDig	1	1	3	0

BookingGUI

SetX()
Display()
Copy()
SetX()
Display()

Obrázok 1.1. Ukážka záznamov z databázy systému RASCAL zložená z matice využívania metód triedou a usporiadaného zoznamu metód využívaných triedou.

Tretia fáza nazývaná *repozitár kódu* ukladá vydolované dáta prostredníctvom *zberača histórie*. Konečná štvrtá, fáza volaná *odporúčací agent* odporúča metódu, ktorú by mal používateľ pri vývoji použiť najbližšie. Aktuálne vyvíjaná metóda sa identifikuje v IDE podľa polohy kurzora.

Class A (active)
```
class A{
  void method1() {
    Button b;
    b.setText(“Button”);
    ---
  }
}
```

Class X (active)
```
class X{
  void method1() {
    Button b;
    b.setText(“Button”);
    b.setAlignmentX(10);
    b.setAlignmentY(10);
  }
}
```

Chceme odporučiť metódu setAligmnentX() skôr ako setAlignmentY().

Obrázok 1.2. Ukážka usporiadania odporúčania v systéme RASCAL [24].

Samotné odporúčanie v systéme sa robí pomocou textovej rady v jasne určenom poradí. Po nájdení najpodobnejšej triedy systému k aktuálne vyvíjanej triede a jej metódy, sa vygenerujú odporúčania podľa obsahu podobnej metódy – Obrázok 1.2.

1.2.2 FrUiT

Druhý odporúčací systém, ktorý predstavujeme, sa nazýva FrUiT [4]. Ide o zásuvný modul do vývojového prostredia Eclipse, ktorý slúži na podporu použitia aplikačných rámcov. Zobrazuje vzťahy v zdrojovom kóde, ako aj časti systému vyskytujúce sa v podobnom kontexte ako aktuálne vyvíjaná časť.

Odporúčanie v systéme sa vytvára v 3 krokoch. V prvom FrUiT extrahuje štrukturálne vzťahy z množiny aplikácií využívajúcich zvolený aplikačný rámec alebo aplikačné programovacie rozhranie. Tieto vzťahy zahŕňajú:

- Využívanie vzťahov *rozšírenia* (angl. extends) – trieda *A rozširuje* triedu *B*
- *Implementácie* (angl. implements) – trieda *A implementuje* rozhranie *B*
- *Prekonania* (angl. override) – trieda *A prekonáva* zdedenú metódu *m* od triedy *B*
- *Volanie* (angl. call) – ľubovoľná metóda triedy *A* volá metódu *m* triedy *B*
- *Vytvorenie inštancie* (angl. instantiate) konštruktor triedy *A* sa volá z inej metódy triedy *A*

Druhým krokom odporúčacieho procesu je dolovanie asociačných pravidiel typických pre zvolený rámec alebo API. Pravidlá sa zapisujú v tvare *ak-tak* formúl (angl. if-then) založených na predpoklade, dôsledku a pravdepodobnosti, s akou dôsledok nastáva, ak je splnený predpoklad. Všeobecný zápis pravidla uvádza vzťah 1, ukážku konkrétneho pravidla uvádza vzťah 2.

$$predpoklad \xrightarrow{dôvera} dôsledok \tag{1.1}$$

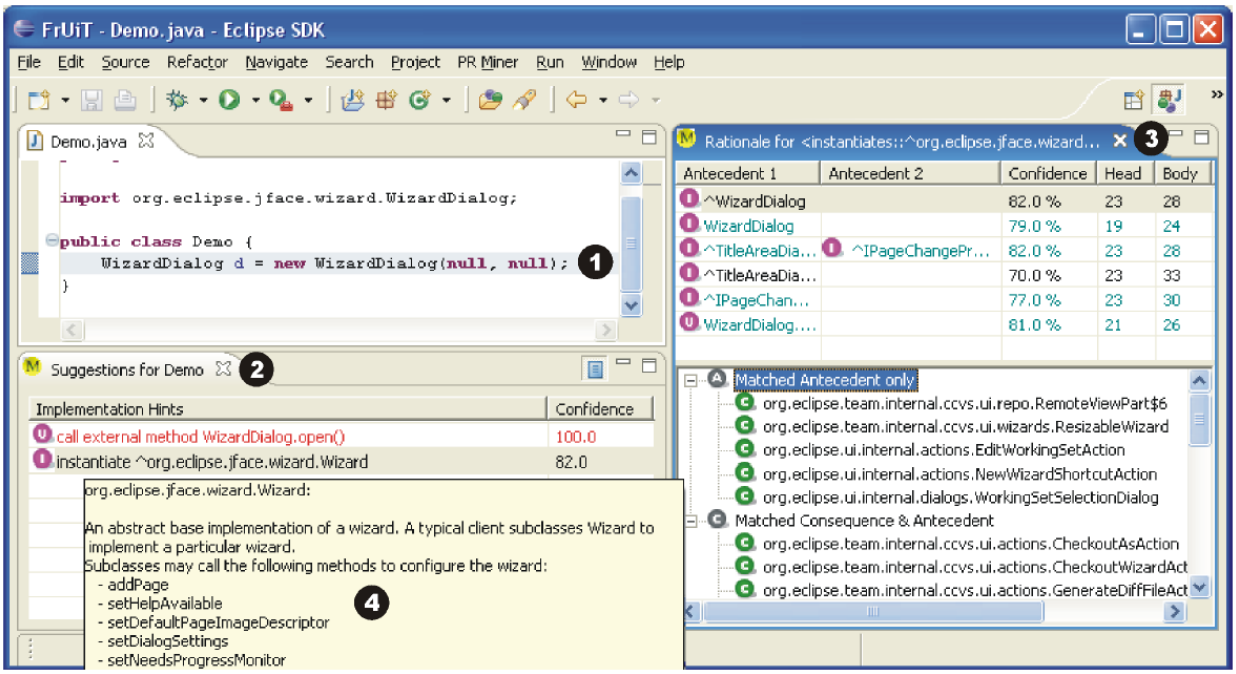
$$volanie: m \xrightarrow{80\%} inštancia: A \tag{1.2}$$

Systém FrUiT rozlišuje niekoľko typov pravidiel:

- *Minimálny výskyt* – existuje minimálne *x* prípadov, kedy pravidlo platí, teda po splnení predpokladu nastáva dôsledok.
- *Minimálna dôvera* – v prípade, že nastal predpoklad, dôsledok platí aspoň v *x*% prípadov.
- *Zavádzajúce pravidlá* – ak existuje viacero pravidiel s rovnakým dôsledkom, ktoré vychádzajú zo súvisiacich predpokladov (napr. $x \xrightarrow{d1\%} z$ a $x \wedge y \xrightarrow{d2\%} z$) vyberieme pravidlo s jednoduchším predpokladom. Dôvodom je, že pravidlá so zložitejšími predpokladmi majú nižšiu pravdepodobnosť ich splnenia, čím znižujú početnosť výskytu dôsledku pravidla oproti pravidlám so súvisiacim jednoduchším predpokladom.
- *Prekonanie pravidiel* – v prípade, že v predchádzajúcom príklade dosahuje druhé pravidlo vyššiu dôveru *d2* ako prvé pravidlo s dôverou *d1*, použijeme stále prvé pravidlo, pokiaľ platí, že rozdiel dôver neprekonáva výhody pri použití jednoduchšieho pravidla.

- *Špecifické pravidlá* – v prípade, že existujú dve pravidlá s rovnakým dôsledkom a dôverou ($x \xrightarrow{d\%} z$; $y \xrightarrow{d\%} z$), spolu s tretím pravidlom, ktoré nastáva v 100% prípadov ($y \xrightarrow{100\%} x$), tak môžeme druhé pravidlo vynechať, pretože je obsiahnuté v prvom pravidle.

V treťom kroku odporúčacieho procesu systém odporúča všetky vygenerované pravidlá pre triedu, ktorú vývojár práve vytvára – Obrázok 1.3. Pravidlá však musia vychádzať z predpokladov, ktoré v systéme skutočne nastali.



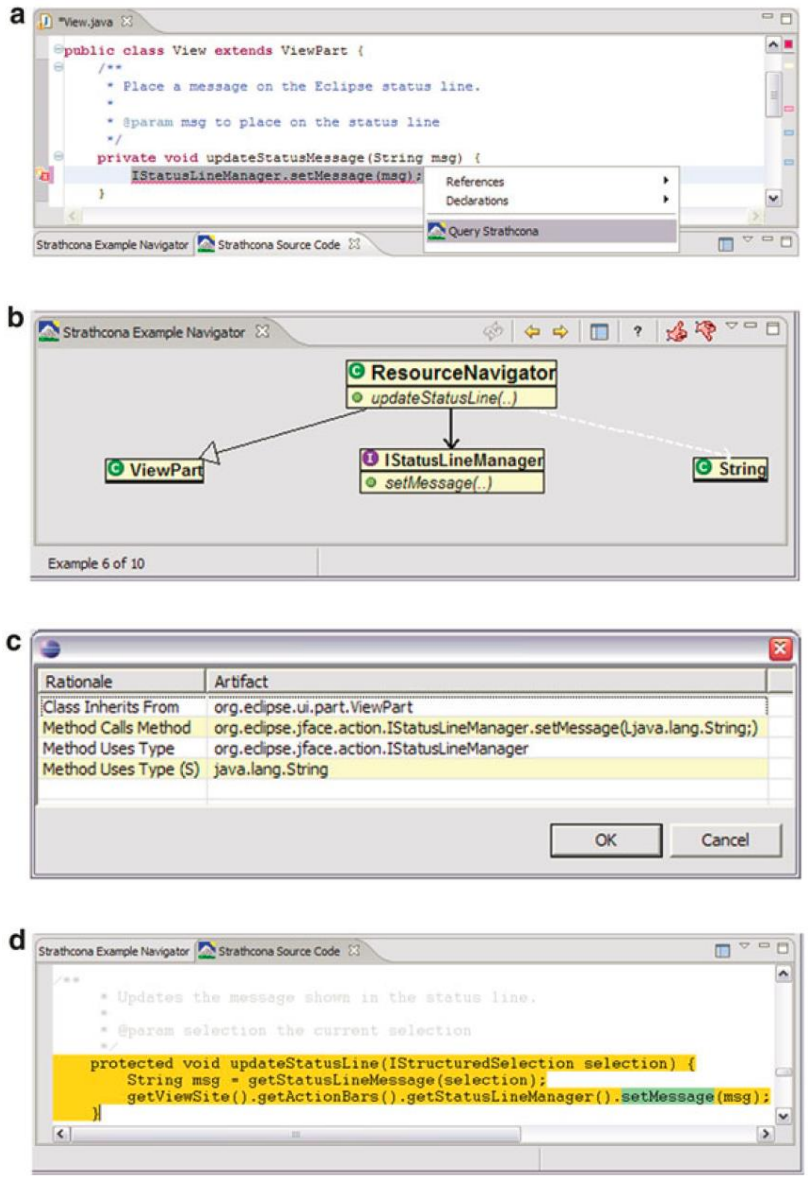
Obrázok 1.3. Ukážka odporúčania v systéme FrUiT [4].

1.2.3 Strathcona

Tento odporúčací systém navrhli taktiež ako zásuvný modul do vývojového prostredia Eclipse. Slúži na odporúčanie skutočných príkladov využitia knižníc tretích strán, ktoré používateľ použil v súčasne vytváranej metóde [12]. Pre odporúčanie používateľ označí zdrojový kód, v ktorom využíva knižnicu a zadá explicitnú požiadavku o ukážky existujúcich riešení využívajúcich danú knižnicu. Takto sa dozvie, akým spôsobom s knižnicou pracujú iní používatelia.

Systém pracuje na princípe extrakcie *štruktúrovaného kontextu* zdrojového kódu entít. Tento kontext zahŕňa signatúru metódy, deklarovaný typ, typ rodiča, volané metódy a využívané atribúty. Štruktúrovaný kontext sa dá využiť dvoma spôsobmi. Jednak ako automatický dopyt opisujúci fragment kódu, pre ktorý vývojár podporu vyvolal, jednak vytvorenie databázy štruktúrálnych kontextov tried využívajúcich knižnice, ktoré nástroj podporuje.

Ukážku jednotlivých vstupných a výstupných rozhraní systému zobrazuje Obrázok 1.4.



Obrázok 1.4. Ukážky systému Strathcona [14]: a) explicitný dopyt od používateľa, b) grafické zobrazenie odporučeného príkladu, c) ukážka odporučeného zoznamu štruktúrovaných faktov pre dopyt, d) ukážka odporučeného zdrojového kódu.

1.2.4 Hipikat

Systém Hipikat je určený pre nováčikov v rámci určitého softvérového projektu [6]. Odporúčania systému slúžia na nájdenie relevantných informácií o fungovaní a údržbe daného systému. Princíp Hipikat-u tkvie v zbere a poskytovaní relevantných častí histórie projektu na základe zdrojového kódu, e-mailovej komunikácie, správach o chybe, záznamov o zmenách v zdrojovom kóde a v dokumentácii.

Odporúčanie sa používateľovi zobrazí po tom, čo explicitne požiada o radu. Odporúčač mu vtedy zobrazí všetky dostupné artefakty k aktuálnej úlohe, napr. k oprave chyby. Systém sa skladá z dvoch častí. Prvá je implementovaná ako zásuvný modul do vývojového prostredia Eclipse. Tento modul slúži na zber dát a ich preposielanie druhej časti. Druhá časť systému na základe

prijatých dát vytvára relačný graf softvérových artefaktov. Na rozdiel od odporúčaní v skôr opísaných systémoch založených na zdrojovom kóde, odporúčanie v Hipikat-e sa zakladá na prepojeniach medzi rozdielnymi artefaktmi a ich vzájomnej podobnosti.

1.2.5 CodeBroker

CodeBroker (Obrázok 1.5) uplatňuje princíp proaktívneho monitorovania aktuálne vyvíjanej metódy, pre ktorú hľadá vo vyvíjanom systéme podobné metódy, ako odporúčanie vzorové riešenia [24]. Cieľom je podpora vývoja nových metód hľadaním metód, ktoré už majú aspoň čiastočne implementovanú vývojárom požadovanú funkcionalitu.

Systém identifikuje odporúčania pomocou monitorovacej deklarácie a ich opisu ihneď ako vývojár začne písať. Systém sa delí na prezentačnú a systémovú vrstvu, ktoré spoločne komunikujú. Prezentačná vrstva monitoruje polohu kurzora a informuje systémovú vrstvu o aktuálnom kontexte a taktiež zobrazuje konečné odporúčania. Systémová vrstva sa opäť delí do dvoch častí. V prvom rade ide o *model prejavu* (angl. discourse model), ktorý ukladá a aktualizuje komentáre a signatúry metód z množiny knižníc, API či aplikačných rámcov na znovupoužitie. Druhá časť systémovej vrstvy pokrýva *model používateľa* určený na zjednodušenie vytvoreného *modelu prejavu* odstránením tried a balíkov zo zoznamu odporúčania, pričom rozhodnutie o odstránení môže byť platné pre aktuálne sedenie (odstraňovaný element je irelevantný pre aktuálnu úlohu) alebo pre všetky sedenia (odstraňovaný element je irelevantný pre používateľa všeobecne).

Aktuálne vyvíjaná metóda sa identifikuje podľa polohy kurzora. Odporúčač vo fáze hľadania vhodných metód hľadá tiež metódy s podobnými komentármi a signatúrami.

1.3 Zhrnutie

Zdrojový kód predstavuje štandardne najaktuálnejšiu časť vyvíjaného systému. Je bežné, že metóda sa najprv naimplementuje a až následne sa komentuje, či testuje. Existujú síce výnimky v podobe vývoja založeného na testovaní (angl. test-driven development) a podobne, ale v prípade väčšiny vyvíjaných systémov sa stále presadzuje štandardný spôsob vývoja.

Z tohto predpokladu vychádza aj časť odporúčacích systémov z domény softvérového inžinierstva, ktoré sú založené na zdrojovom kóde. Tieto odporúčače pri modelovaní vlastností a preferencií jednotlivých používateľov vychádzajú primárne zo zdrojového kódu a iné zdroje dát používajú nanajvýš ako doplnok. Na základe modelov potom systémy generujú odporúčanie, ktoré sa môžu zamerať na návrh metódy alebo knižnice na použitie, ukážku metód podobných k vyvíjanej aj pre nového používateľa na efektívnejšie zorientovanie sa vo vyvíjanom systéme.

V kapitole predstavujeme budovanie odporúčacieho systému založeného na zdrojovom kóde a opisujeme najdôležitejšie rozhodnutia, ktoré treba pri vývoji takéhoto systému učiniť v závislosti od cieľa odporúčania a ďalších vstupných či výstupných faktorov. V druhej časti kapitoly predstavujeme existujúce odporúčacie systémy opisovaného typu. Zameriavame sa najmä na priblíženie ich predmetu odporúčania, cieľového používateľa, či spôsobu, akým modelujú a odporúčajú.

Obrázok 1.5. Ukážka obrazoviek systému CodeBroker [ľč]: a) odporúčanie metód – prvá položka odporúčania metóda *getInt* je označená, b) kliknutie ľavým tlačidlom myši na odporúčanú položku zobrazí dokumentáciu (JavaDoc) danej metódy, c) kliknutie pravým tlačidlom myši na odporúčanie umožňuje jej odstránenie zo zoznamu, d) zmena dopytu.

Literatúra

[1] Ashok, B., Joy, J., Liang, H., Rajamani, S.K., Srinivasa, G., Vangala, V.: DebugAdvisor: A recommender system for debugging. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 373–382 (2009). doi: 10.1145/1595696.1595766

[2] Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J.: Dimensions and metrics for evaluating recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 10. Springer, New York (2014)

[3] Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 213–222 (2009). doi: 10.1145/1595696.1595728

[4] Bruch, M., Schäfer, T., Mezini, M.: FrUiT: IDE support for framework understanding. In: Proceedings of the Eclipse Technology eXchange, pp. 55–59 (2006). doi: 10.1145/1188835.1188847

[5] Cottrell, R., Walker, R.J., Denzinger, J.: Semi-automating small-scale source code reuse via structural correspondence. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 214–225 (2008). doi: 10.1145/1453101.1453130

[6] Čubranić, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: A project memory for software development. IEEE Trans. Software Eng. 31(6), 446–465 (2005). doi: 10.1109/TSE.2005.71

[7] Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. ACM Trans. Software Eng. Meth. 20(4), 19:1–19:35 (2011). doi: 10.1145/2000799.2000805

[8] Duala-Ekoko, E., Robillard, M.P.: Using structure-based recommendations to facilitate discoverability in APIs. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 79–104 (2011). doi: 10.1007/978-3-642-22655-7_5

[9] German, D.M., Čubranić, D., Storey, M.A.D.: A framework for describing and understanding mining tools in software development. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 7:1–7:5 (2005). doi: 10.1145/1082983.1083160

[10] Goldberg, D., Nichols, D., Oki, B.M, Terry, D.: Using collaborative filtering to weave an information tapestry. Commun. ACM 35, 12, pp. 61-70, (1992). doi=10.1145/138859.138867

[11] Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the neighborhood with Dora to expedite software maintenance. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 14–23 (2007). doi: 10.1145/1321631.1321637

[12] Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: An approach to recommend relevant examples. IEEE Trans. Software Eng. 32(12), 952–970 (2006). doi: 10.1109/TSE.2006.117

[13] Long, F., Wang, X., Cai, Y.: API hyperlinking via structural overlap. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 203–212 (2009). doi: 10.1145/1595696.1595727

[14] Lozano, A., Kellens, A., Mens, K.: Mendel: Source code recommendation based on a genetic metaphor. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 384–387 (2011). doi: 10.1109/ASE.2011.6100078

[15] Lozano, A., Kellens, A., Mens, K., Arévalo, G.: MEntoR: Mining entities to rules. In: Proceedings of the Belgian–Netherlands Evolution Workshop (2010a)

[16] Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: Helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 48–61 (2005). doi: 10.1145/1065010.1065018

[17] McCarey, F., Ó Cinnéide, M., Kushmerick, N.: RASCAL: A recommender agent for agile reuse. Artif. Intell. Rev. 24(3–4), 253–276 (2005). doi: 10.1007/s10462-005-9012-8

[18] Murphy-Hill, E., Murphy, G.C.: Recommendation delivery: Getting the user interface just right. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 9. Springer, New York (2014)

[19] Robillard, M.P.: Topology analysis of software dependencies. ACM Trans. Software Eng. Meth. 17(4), 18:1–18:36 (2008). doi: 10.1145/13487689.13487691

[20] Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software 27(4), 80–86 (2010). doi: 10.1109/MS.2009.161

[21] Said, A., Tikk, D., Cremonesi, P.: Benchmarking: A methodology for ensuring the relative quality of recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 11. Springer, New York (2014)

[22] Tosun Mısırlı, A., Bener, A., Çaglayan, B., Çalıklı, G., Turhan, B.: Field studies: A methodology for construction and evaluation of recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 13. Springer, New York (2014)

[23] Walker, R.J., Holmes, R.: Simulation: A methodology to evaluate recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 12. Springer, New York (2014)

[24] Ye, Y., Fischer, G.: Reuse-conducive development environments. Automat. Software Eng. Int. J. 12(2), 199–235 (2005). doi: 10.1007/s10515-005-6206-x

[25] Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J., Ou, P.: Automatic parameter recommendation for practical API usage. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 826–836 (2012). doi: 10.1109/ICSE.2012.6227136

[26] Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Software Eng. 31(6), 429–445 (2005). doi: 10.1109/TSE.2005.72

2 Dolovanie v záznamoch chýb

Softvér ovplyvňuje v súčasnosti takmer každú ľudskú aktivitu. Využíva sa v oblasti telekomunikácii, zdravotníctva, výroby, bankovníctva, poisťovníctva, obchodu, vzdelávania a výskumu. Takmer všetko sa riadi softvérom, preto je veľmi nepríjemné, keď sa v ňom objavia chyby. Chyby v softvéri znižujú jeho spoľahlivosť, kvalitu a tým pádom aj dôveru a spokojnosť používateľov. Okrem toho môžu chyby spôsobiť materiálne a finančné škody, stratu informácii či dokonca ujmu na zdraví. Preto je nevyhnutné minimalizovať vznik a výskyt chýb. Testovanie softvéru je náročné na čas, ľudské zdroje a nemusí vždy odhaliť všetky chyby. Pomoc prinášajú systémy na sledovanie chýb v softvéri. Tieto systémy zhromažďujú záznamy o vzniknutých chybách. Ich následná analýza môže pomôcť pri identifikácii softvérových komponentov a súčastí, ktoré sú náchylnejšie na chyby a vďaka tomu sa môžu prijať potrebné rozhodnutia a zefektívniť proces testovania. Vzhľadom na veľký počet dát v systémoch na sledovanie chýb treba použiť metódy dolovania v dátach.

Softvér sa stáva neustále zložitejší a jeho vývoj je náročný. Preto sa vývojári môžu dopúšťať omylov, ktoré vyvolávajú chyby v kóde. Chyby v softvéri vznikajú najmä z dôvodu, že ľudia sú omylní, pracujú pod časovým tlakom, v komplikovanej štruktúre so zložitejšími systémami a meniacimi sa technológiami. V niektorých prípadoch dochádza k chybám, ktoré sú zapríčinené prostredím, napr. chyby vo firmvéri, poruchami hardvéru alebo zmenou hardvérových podmienok.

V súčasnosti sa pri vývoji softvérových produktov využívajú systémy na sledovanie chýb (angl. bug-tracking systems). Systémy slúžia na uchovávanie relevantných informácii o objavených chybách. Systémy sledujú vývoj chyby od jej objavenia (nahlásenia), priradenia konkrétnej osobe, ktorá ju má riešiť, až po vyriešenie, uzatvorenie, prípadne znovu otvorenie, ak sa ukáže, že navrhnuté riešenie nebolo správne. Systémy na sledovanie chýb neslúžia len ako

databáza, ale aj ako komunikačný prostriedok medzi vývojármi a testermi, prípadne zákazníkmi, ak majú prístup do daného systému.

Výsledky dolovania v dátach zo systémov na sledovanie chýb sa môžu použiť v odporúčacích systémoch, ktoré môžu následne poskytovať návrhy ako zlepšiť kvalitu kódu, znížiť výskyt chýb alebo identifikovať miesta, ktoré by sa mali testovať dôkladnejšie.

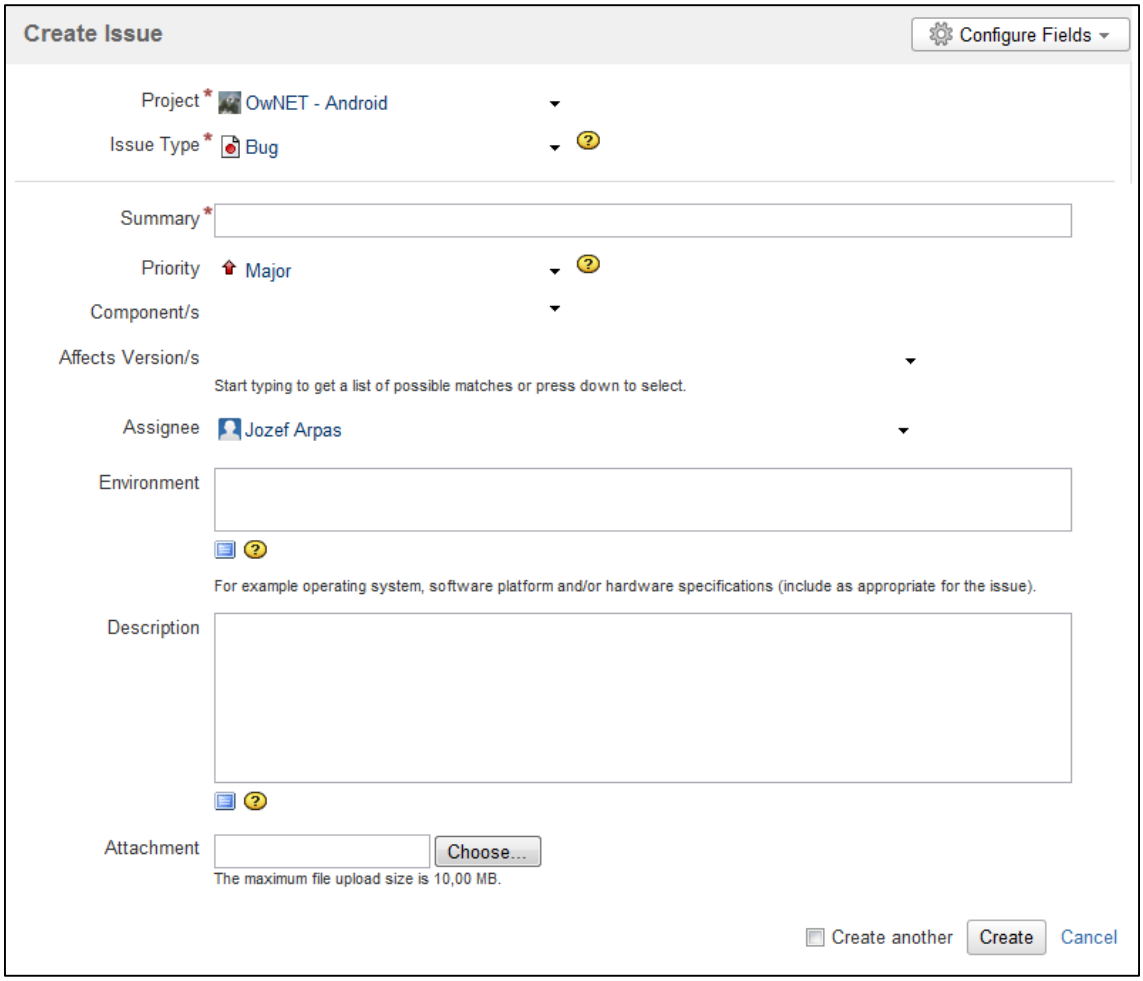
2.1 Záznam o chybe

Záznam o chybe sa tiež nazýva správa o chybe. Vo všeobecnosti obsahuje informáciu o správaní softvéru, ktoré nezodpovedá očakávanému stavu. Záznam o chybe má väčšinou podobu formulára, ktorý obsahuje povinné a nepovinné polia. Jednotlivé polia slúžia na informovanie vývojárov i čitateľov o jednotlivých vlastnostiach danej chyby. Hodnota každého poľa zvyčajne slúži na zatriedenie chyby s ohľadom na danú vlastnosť alebo prispieva k opisu či diskusii k danej chybe. Obrázok 2.1 znázorňuje typický záznam o chybe. Záznam obsahuje niekoľko typov informácii:

- Informácie o produkte, verzii a prostredí, ktoré hovoria vývojárom, v ktorom produkte a v akom prostredí nastala chyba.
- Opis a názov chyby, kľúčové slová a inštrukcie ako replikovať objavenú chybu.
- Polia typu *status*, *issue type*, *assignee*, *due date* a *priority*, ktoré pomáhajú manažmentu riadiť kto a dokedy má danú chybu opraviť.
- Diskusia, v ktorej vývojári informujú o vykonaných krokoch a zisteniach. Do diskusie sa môže zapájať viacero vývojárov a navrhovať svoje riešenia.
- Prílohy, napr. logy, vstupy a výstupy systému, snímku obrazovky a iné súbory, ktoré môžu pomôcť pri riešení problému.

Všeobecne platí, že všetky záznamy o chybách, nezávisle na ich pôvode, majú rovnaký cieľ, ktorým je dokumentovať objavené chyby. Rôzne systémy na sledovanie chýb však odrážajú odlišné procesy a základnú myšlienku. Preto je dôležité pochopiť, že správy o chybách z rôznych systémov a od rôznych tímov a projektov budú rozdielne. Je nutné identifikovať tieto rozdiely, pretože sú dôležité pre pochopenie, ako správy vznikli a ako ich menili.

V súčasnosti existuje viacero komerčných aj voľne dostupných systémov na sledovanie chýb. Systémy sa líšia v zozname povinných a nepovinných polí, v zozname preddefinovaných polí a ich hodnotách, v prídavnej funkcionalite, dátovom modele a interoperabilite s inými systémami. Medzi najznámejšie systémy patrí Jira, Bugzilla, Redmine, Team Foundation Server, YouTrack a ďalšie.



Obrázok 2.1 Ukážka záznamu o chybe. Záznam má podobu formulára s povinnými a nepovinnými poliami. Niektoré polia majú preddefinvané hodnoty, iné slúžia na zápis textu.

2.2 Kvalita záznamov

Kvalita záznamov je kľúčová pre poskytovanie vhodných odporúčaní a zvyšovanie kvality softvéru. Ovplyvňuje ju niekoľko faktorov:

Správna identifikácia – Záznamy o chybách vytvárajú rôzne skupiny používateľov, napr. vývojári, testeri, zákazníci. Každá skupina má rôznu prax a vedomosť o tom, ako softvér funguje, či ide o skutočnú chybu, novú zatiaľ nezdokumentovanú funkcionalitu, nezrovnalosť s dokumentáciou alebo chybné vstupné údaje, ktoré spôsobujú neočakávané správanie systému. Pri analýze záznamov preto treba skontrolovať správnosť údajov. Podľa [3], kde autori ručne preskúmali viac ako 7000 záznamov o chybách v 5 softvérových projektoch, bolo v priemere až 33,8% záznamov nesprávne klasifikovaných ako chyba, hoci išlo o novú funkcionalitu, zlepšenia, refaktorovanie kódu alebo nezrovnalosť s dokumentáciou.

Neúplnosť záznamov – Ďalším problémom, ktorý ovplyvňuje kvalitu záznamov je ich neúplnosť. Tvorcovia záznamov nemusia mať pri objavení chyby všetky dostupné informácie, alebo

ich zabudnú uviesť do záznamu. Chýbajúce hodnoty v poliach môžu výrazne ovplyvňovať kvalitu vydolovaných informácii a tým pádom aj kvalitu samotných odporúčaní.

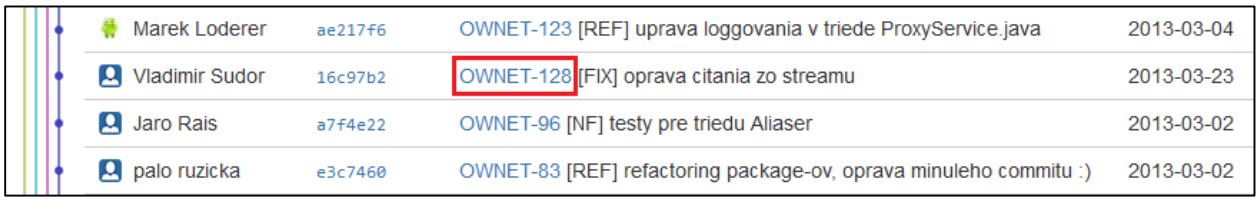
Duplicita – Je pravdepodobné, že chybu objaví viacero používateľov. Základná chyba môže byť rovnaká, ale prejavuje sa rôznymi spôsobmi (prejaví sa v rôznom čase v závislosti od práce, ktorú používateľ vykonal) a preto je ťažké identifikovať, že ide o duplicitu. Škodlivosť duplicít v záznamoch je diskutabilná. Vývojári ju vo všeobecnosti nepokladajú za vážny problém, práve naopak. Duplicitné záznamy obsahujú viac informácii, ktoré môžu pomôcť pri riešení. Len málo systémov na sledovanie chýb má v sebe implementovaný odporúčač, ktorý by upozorňoval spravodajcu chyby na prítomnosť podobných chýb, aby sa predišlo vzniku potenciálnych duplicít. Duplicity v záznamoch spôsobujú komplikácie počas analýzy a dolovania, pretože zapríčiňujú skreslenie výsledkov. Existuje viacero prístupov, ktoré sa venujú odhaľovaniu duplicít [6,7]. Sú založené na hľadaní podobností jednotlivých polí záznamov pomocou rôznych porovnávacích algoritmov a metód spracovania prirodzeného jazyka. Následne sa záznamy označujú ako duplicitné a môžu sa spojiť do jedného super záznamu.

Ručné upravenie a predspracovanie záznamov môže výrazným spôsobom prispieť k zvýšeniu kvality finálnych odporúčaní.

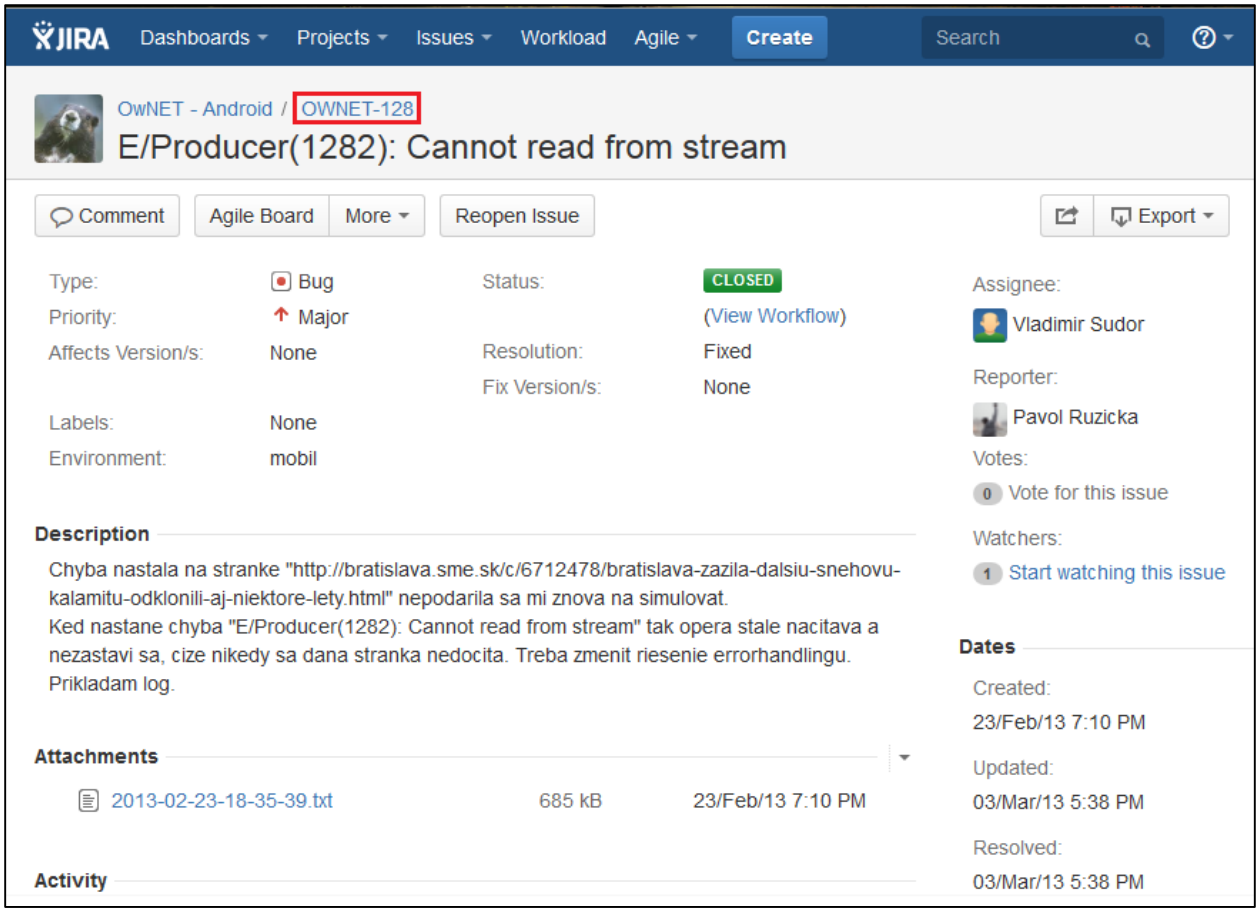
2.3 Mapovanie chýb

Záznamy o chybách sú kľúčom k údržbe a ďalšiemu vývoju softvéru. Preto je o ne veľký záujem zo strany vývojárov a manažérov. V mnohých spoločnostiach a projektoch sú systémy na sledovanie chýb oddelené od systémov pre správu verzii zdrojového kódu (VCS, angl. version control systems) a preto neumožňujú okamžitú analýzu kvality kódu a odporúčania na jej zlepšenie (napr. identifikácia komponentov, ktoré sa prejavili ako chybné). Preto sa musia vytvoriť vzťahy medzi záznamami o chybách s ich riešeniami v podobe zmien vo verziovacích systémoch.

Napriek tomu, že mapovanie chýb na zmeny v zdrojovom kóde je bežnou úlohou v procese dolovania znalostí v tejto oblasti, existuje prekvapivo málo zásadne odlišných mapovacích stratégií a ešte menej výskumných prác, ktoré by overovali správnosť daných mapovacích stratégií a ich vplyv na odporúčania a vytvárané predikčné modely. Väčšina mapovacích stratégií je založená na hľadaní referencii v správach o odovzdaní (angl. commit messages) pomocou regulárnych výrazov. Presnosť mapovania sa môže zvýšiť aplikáciou filtrov.



Obrázok 2.2 Správa z odovzdania softvérového artefaktu obsahuje referenciu na záznam o chybe. Ukážka je zo systému na správu verzií Bitbucket.



Obrázok 2.3 Ukážka záznamu o chybe v systéme Jira.

2.3.1 Mapovanie pomocou regulárnych výrazov

Fischer a kol. [2] a Čubranič a kol. [1] boli medzi prvými výskumníkmi, ktorí sa venovali hľadaniu referencií medzi zmenami v kóde vo VCS a záznamami v systémoch na sledovanie chýb. Ich prístup je priamočiary. Správa z odovzdania softvérového artefaktu by mala obsahovať referenciu na príslušný záznam o chybe v systéme na sledovanie chýb (napr. "*Fixes bug 1234*" alebo "*fixing #2367*"), ktorý sa dá ľahko rozpoznať pomocou regulárnych výrazov. Výsledkom je množina dvojíc záznamov o chybe a správ o zmenách v kóde.

Tento prístup môže vytvoriť falošné dvojice. Regulárne výrazy sú vynikajúce na odhaľovanie vzorov v textoch, no neberú do úvahy kontext, napr. regulárny výraz: `(bug|issue|fix):?\s*#?\s?(\d+)` nájde zhodu s textom "*fixing copyright issue: 2002 → 2003.*" Extrahované číslo 2002 však nezodpovedá identifikátoru chyby, ale roku.

2.3.2 Aplikácia filtrov

Mnoho dátových analytikov používa sadu filtrov, aby sa odstránili potenciálne falošné dvojice. Najčastejšie aplikované filtre:

- *Poradie vytvorenia* – záznam o chybe by sa mal vytvoriť pred aplikovaním zmeny v kóde

- *Poradie vyhodnotenia* – zmena v kóde by sa mala vykonať predtým, než sa chyba označí za vyriešenú ("*Resolved*").
- *Autori* – osoba, ktorá vykoná zmenu v kóde, by mala byť tá istá osoba, ktorá označí chybu na vyriešenú.
- *Verzia softvéru* – filter môže porovnať verziu softvéru, ktorá je uvedená v zázname o chybe s verziou softvéru vo vetve, kde bola aplikovaná zmena.
- *Rozsah identifikátorov* – niektoré rozsahy číselných identifikátorov (ID) predstavujú s väčšou pravdepodobnosťou falošné prípady, napr. malé hodnoty ID môžu predstavovať rok, dátum, verziu alebo číslo riadku, než skutočné ID záznamu.

Použitie regulárnych výrazov a filtrov je veľmi závislé od individuálnych softvérových projektov, použitých nástrojoch a zadefinovaných procesov vývoja.

V poslednom kroku sa vykonáva mapovanie chýb s príslušnými časťami kódu. Každá zmena kódu sa týka jedného alebo viacerých softvérových artefaktov, a tak je možné vytvoriť mapovanie medzi záznamami o chybách s jednotlivými artefaktmi, ktoré boli týmito chybami ovplyvnené.

Na obrázkoch 4 a 5 sa nachádza ukážka prepojenia repozitára zdrojového kódu so systémom na sledovanie chýb. Pomocou regulárnych výrazov sa v správach o odovzdaní vytvárajú referencie na príslušné záznamy o chybách.

Po vytvorení mapovania sa môže pristúpiť k predikciám chýb a poskytovaniu odporúčaní.

2.4 Predikcia chýb

Vedomosť o tom, kde a ako boli chyby opravené, umožňuje skúmať, prečo boli artefakty náchylné na chyby a ktoré procesy a okolnosti viedli k vzniku týchto chýb. Preskúmané chyby dokonca umožňujú získavať vedomosti, ktoré sa môžu využiť v budúcom vývoji. Jedným zo spôsobov využitia minulých chýb je odhadnúť a predvídať budúce chyby.

2.4.1 Predikčný model

Vhodným spracovaním historických dát s cieľom predpovedať budúcnosť vzniká predikčný model. Cieľom modelu je odhadnúť počet potenciálnych chýb a v niektorých prípadoch aj ich polohu v zdrojovom kóde. Chyby nie sú v kóde rovnomerne rozmiestnené, predikčný model sa preto snaží odhadnúť tie miesta vo vyvíjanom systéme, ktoré sú náchylnejšie na výskyt chýb.

V priebehu rokov vedci a inžinieri navrhli množstvo softvérových metrík, ktoré sa môžu použiť na tvorbu predikčných modelov. Softvérové metriky obsahujú meta-informácie o jednotlivých vlastnostiach softvérových artefaktov (napr. počet riadkov kódu zdrojového súboru, počet metód v súbore alebo počet autorov, ktorí zmenili zdrojový súbor). Metriky umožňujú oddeľovať artefakty náchylné na chyby od artefaktov, ktoré sú bez chýb. Meta-informácie môžu obsahovať tiež opis procesu (napr. kto a akým spôsobom vyvinul kód) a informácie o závislostiach medzi jednotlivými časťami kódu (napr. zviazanosť komponentov, volania metód).

Na základe vybraných metrík vypočítava predikčný model pre softvérový artefakt rizikové faktory [1], ktoré indikujú:

- pravdepodobnosť, že sa v danom artefakte nachádza chyba (klasifikácia)
- počet očakávaných chýb, ktoré sa nachádzajú v artefakte (predikcia).

Aby sa mohli metódy strojového učenia naučiť a otestovať, ktoré metriky najviac korelujú s artefaktmi náchylnými na chyby, vyžadujú prítomnosť premennej, ktorá obsahuje informáciu o skutočnom stave jednotlivých artefaktov (napr. true/false – artefakt obsahuje/neobsahuje chybu, alebo číslo vyjadrujúce skutočný počet identifikovaných chýb v danom artefakte). Tieto informácie sa môžu získať z verziovacích systémov zdrojových kódov.

Tabuľka 2.1 obsahuje ukážku datasetu z projektu Rhino[1], ktorý sa môže použiť na vytvorenie a otestovanie predikčného modelu. Každý záznam obsahuje názov zdrojového súboru, množinu softvérových metrík a v poslednom stĺpci informáciu o počte identifikovaných chýb v súbore.

Tabuľka 2.1 Ukážka datasetu obsahujúceho zoznam zdrojových súborov. Pre každý súbor je vypočítaná množina softvérových a sieťových metrík.

filename	size	sizeOut	sizeIn	density	...	numBugs
optimizer/ClassCompiler.java	12	11	2	0.2651	...	1
optimizer/Codegen.java	30	29	2	0.1678	...	37
JavaAdapter.java	23	22	3	0.2094	...	11
ast/AstRoot.java	11	7	6	0.4	...	0
Parser.java	73	71	5	0.0778	...	33
ast/FunctionNode.java	20	7	17	0.2710	...	1
IRFactory.java	69	67	3	0.0837	...	23
CompilerEnvirons.java	14	5	11	0.2197	...	3
ast/ScriptNode.java	24	10	18	0.2536	...	0
ScriptRuntime.java	98	51	72	0.0842	...	41
Scriptable.java	122	2	11	0.0529	...	0
IdFunctionObject.java	37	8	32	0.1876	...	0
Context.java	148	46	130	0.0484	...	19
ast/AstNode.java	67	9	64	0.0961	...	1

2.4.2 Trénovanie predikčných modelov

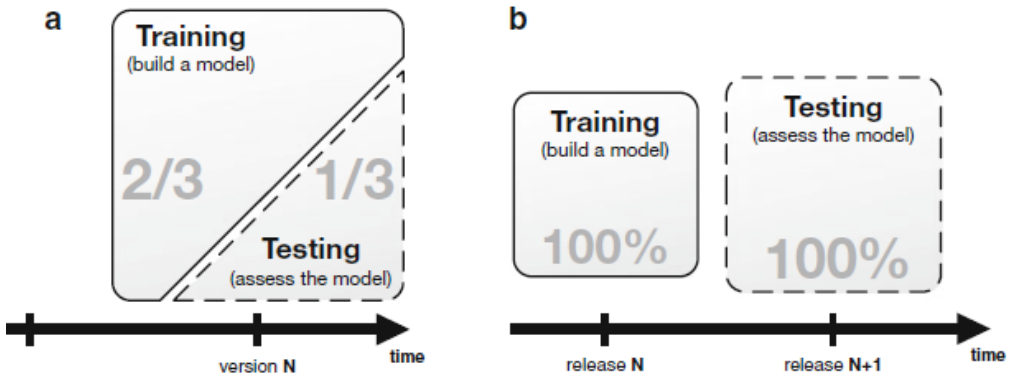
Na trénovanie a evaluáciu predikčných modelov je treba trénovaciu a testovaciu množinu. Obrázok 2.4 uvádza dva najčastejšie prístupy delenia datasetu na uvedené množiny.

Prvým prístupom je náhodné rozdelenie datasetu na dve množiny, v prípadoch, keď existuje iba jedna verzia softvérového produktu. Dataset sa rozdelí na trénovaciu množinu, ktorá zvyčajne obsahuje dve tretiny záznamov a testovaciu množinu, ktorá obsahuje zvyšnú tretinu záznamov.

Druhý prístup sa uplatňuje v prípadoch, keď existujú aspoň dve verzie softvérového produktu. Predchádzajúce verzie produktu slúžia na trénovanie modelu, zatiaľ čo neskoršie verzie

[1] https://github.com/mozilla/rhino

slúžia na testovanie. Tento prístup sa podobá situáciám v reálnom svete, kde dáta a dokumentácia z minulých projektov slúžia na odhalenie potenciálne chýb v nasledujúcich verziách softvéru.



Obrázok 2.4 Dva spôsoby rozdelenia datasetu na trénovaciu a testovaciu množinu: a) náhodné rozdelenie, b) rozdelenie s využitím viacerých verzii projektu [5].

Po rozdelení sa trénovacia množina odovzdá algoritmom strojového učenia (napr. metódam podporných vektorov, rozhodovacím stromom, alebo logistickej regresii), ktoré vytvoria príslušný model. Vytvorený model prijíma nové inštancie artefaktov a predikuje ich hodnotu. Predikčné modely sa môžu natrénovať ako klasifikačné alebo regresné modely. Klasifikačné modely spájajú inštancie s určitou kategóriou (napr. náchylný alebo nenáchylný ku chybe), zatiaľ čo regresné modely predpovedajú presný počet chýb, ktoré možno očakávať v príslušnom softvérovom artefakte.

2.4.3 Od predikcie k odporúčaniu

Mnoho predikčných modelov a ich predikované hodnoty sa môžu interpretovať ako odporúčania, keďže predikované hodnoty odhadujú budúce udalosti a situácie. Tieto znalosti môžu slúžiť na prijatie podporných alebo protipodporných akcií voči predpokladanému trendu a výsledkom.

Príkladom sú chybové predikčné modely, ktoré predpovedajú očakávaný počet chýb v stanovených softvérových artefaktoch. To znamená, že výsledky predikcie týchto modelov sa môžu použiť na vytvorenie zoznamu artefaktov, ktoré sa budú testovať a vyhodnocovať oveľa dôkladnejšie. Zapojenie predikčného modelu do odporúčacieho systému zvyčajne vyžaduje výklad predikovaných hodnôt a postrehy, ktoré umožňujú vyhodnotiť prípadné dôsledky.

2.5 Príklad predikcie

Na jednoduchom príklade ukážeme postup dolovania informácii v dátach zo systému na sledovanie chýb. Dáta čerpáme z projektu Rhino[2] spoločnosti Mozilla Foundation. Chyby

[2] https://github.com/mozilla/rhino

v zdrojovom kóde sa sledujú v systéme Bugzilla[3]. Na dolovanie, extrakciu a parsovanie informácii z daného systému použili rámec Mozkito. Postup a ukážka konfiguračných súborov, dátového modelu, potrebných Java tried a regulárnych výrazov je uvedený v práci [5]. Autori práce zverejnili na internete vytvorený dataset, na ktorom sa môžu trénovať a testovať predikčné modely.

V ďalších častiach uvádzame postup jednoduchej predikcie (v tomto prípade klasifikácie), ktorej cieľom je určiť či vyvíjaná trieda v projekte Rhino, obsahuje alebo neobsahuje chybu.

Krok 1: Načítanie knižníc

Skript využíva funkcie z viacerých knižníc. Predovšetkým funkcie z balíčka *caret* [4] v jazyku R. Posledný riadok nastavuje počiatočné náhodné číslo na ľubovoľnú hodnotu (v našom prípade je to 1). Toto zabezpečí reprodukovateľnosť výsledkov príkladu.

```
> rm(list = ls(all = TRUE))
> library(caret)
> library(gdata)
> library(plyr)
> library(reshape)
> library(R.utils)
> set.seed(1)
```

Krok 2: Načítanie dát

Pomocou príkazu `read.table()` sa potrebné dáta načítajú priamo z internetu. Po vykonaní príkazu obsahuje premenná `data` zoznam vybraných tried projektu *Rhino,* závislostné metriky [8] a počet identifikovaných chýb pre jednotlivé triedy (stĺpec s názvom `numBugs`).

```
> data <- read.table(http://rsse.org/book/c06/sampleset.csv,header=T,
  row.names=1, + sep=",")
```

Krok 3: Rozdelenie datasetu

Najprv treba rozdeliť dataset data na trénovaciu a testovaciu množinu pomocou metódy stratifikovaného *vzorkovania (stratified sampling).*

Prvé dva riadky R kódu slúžia na oddelenie premennej, ktorá obsahuje informáciu o počte identifikovaných chýb od zvyšných premenných obsahujúcich závislostné metriky, podľa ktorých sa budú trénovať vybrané predikčné modely.

V treťom riadku sa upraví informácia o počte chýb. Budú sa rozlišovať iba dve hodnoty: “One“ ak trieda obsahuje aspoň jednu chybu a “Zero“ ak trieda neobsahuje ani jednu chybu.

Nakoniec sa použije funkcia `createDataPartition`, ktorá rozdelí dataset na trénovaciu a testovaciu množinu. Testovacia množina obsahuje dve tretiny záznamov. Zostávajúca tretina záznamov je zaradená do trénovacej množiny.

```
> dataX <- data[,which(!colnames(data) %in% c("numBugs"))]
> dataY <- data[, which(colnames(data) %in% c("numBugs"))]
```

[3] https://bugzilla.mozilla.org

```
> dataY <- factor(ifelse(dataY > 0, "One", "Zero"))
>
> inTrain <- createDataPartition(dataY, times = 1, p = 2/3)
>
> trainX <- dataX[inTrain[[1]], ]
> trainY <- dataY[inTrain[[1]]]
> testX <- dataX[-inTrain[[1]], ]
> testY <- dataY[-inTrain[[1]]]
```

Po vykonaní kódu obsahuje premenná trainX závislostné metriky tried, ktoré boli zaradené do trénovacej množiny. V premennej trainY sa nachádza údaj o prítomnosti chýb ("One" alebo "Zero") pre triedy v trénovacej množine. V premenných testX a testY sa nachádzajú analogické údaje pre testovaciu množinu.

Krok 4: Príprava dát

Výsledky predikcie sa môžu zlepšiť predprípravou dát. Najprv sa odstraňujú premenné, ktoré neprispievajú k tvorbe konečného modelu. Existujú dva prípady:

- V prípade, ak hodnoty premenných naprieč všetkými záznamami majú nulový rozptyl (môže sa považovať za konštantný). Funkcia `nearZeroVar` (`trainX`) vracia pole stĺpcov, ktorých hodnoty nevykazujú významný rozptyl.

```
> train.nzv <- nearZeroVar(trainX)
> if (length(train.nzv) > 0) {
+ trainX <- trainX[, -train.nzv]
+ testX <- testX[, -train.nzv] }
```

- Ak je premenná vo vzťahu k inej premennej a tým pádom neprináša novú informáciu. Funkcia `findCorrelation` prehľadáva vytvorenú maticu korelácií `trainX` a vracia pole stĺpcov, ktoré by sa mali odstrániť, aby sa znížila párová korelácia. Hranicu párovej korelácie stanovili na hodnotu 0,9.

```
> trainX.corr <- cor(trainX)
> trainX.highcorr <- findCorrelation(trainX.corr, 0.9)
> if (length(trainX.highcorr) > 0) {
+ trainX <- trainX[, -trainX.highcorr]
+ testX <- testX[, -trainX.highcorr] }
```

V nasledujúcom kroku sa zjednotí škála hodnôt použitých premenných, aby sa zamedzil vplyv veľkých hodnôt na predikčný model. Rozsah hodnôt sa upraví na interval $\langle 0,1 \rangle$.

Ďalšie zníženie počtu premenných sa môže vykonať pomocou metódy Analýzy hlavných komponentov (PCA – *principal component analysis*). Metóda určuje minimálny počet premenných, ktoré poskytujú maximálny rozptyl v dátach. Funkcia `preProcess` odhaduje potrebné parametre pre každú operáciu a funkcia `predict` ich aplikuje na konkrétny dataset.

```
> xTrans <- preProcess(trainX, method = c("center", "scale"))
> trainX <- predict(xTrans, trainX)
> testX <- predict(xTrans, testX)
```

Krok 5: Trénovanie modelov

Na tvorbu predikčných modelov sa použije sedem metód [43, 65]:

- metóda podporných vektorov – *support vector machine with radial kernel* (`svmRadial`),
- logistická regresia – *logistic regression* (`multinorm`),
- rekurzívne delenie – *recursive partitioning* (`rpart`),
- metóda k-najbližších susedov – *k-nearest neighbor* (`knn`),
- metóda tree bagging – tree bootstrap aggregating (`treebag`),
- náhodný les – *random forest* (`rf`),
- naivný Bayesov klasifikátor – *naive Bayesian classifier* (`nb`).

```
> models <- c("svmRadial","multinom","rpart","knn","treebag","rf","nb")
```

Každý model sa optimalizuje balíkom *caret*, ktorý trénuje jednotlivé typy modelov pomocou prislúchajúcich parametrov. Explicitne sa nastaví hodnota stupňa optimalizácie (`tuneLength`), ktorá sa vzťahuje na všetky modely.

```
> train.control <- trainControl(number=2)
> tuneLengthValue <- 5
```

Funkcia `train()` generuje predikčné modely (`fit`), ktoré sa uchovávajú v zozname modelov `modelList`, kvôli neskorším výpočtom presnosti, pokrytia a iných štatistík.

```
> modelsFit <- list()
+ for(model in models){
+ print(paste("training",model," ..."))
+ fit <- train(trainX, trainY, method = model,
+ tuneLength = tuneLengthValue, trControl = train.control,
+ metric = "Kappa")
+ modelsFit[[model]] <- fit
+ }
```

Krok 6: Výpočet predikcie

Natrénované modely pomocou funkcie `extractPrediction()` predikujú hodnoty na testovacej množine `testX`.

```
> pred.values <- extractPrediction(modelsFit, testX, testY)
> pred.values <- subset(pred.values, dataType == "Test")
> pred.values.split <- split(pred.values, pred.values$object)
```

Krok 7: Výpočet presnosti, pokrytia a F1-štatistiky

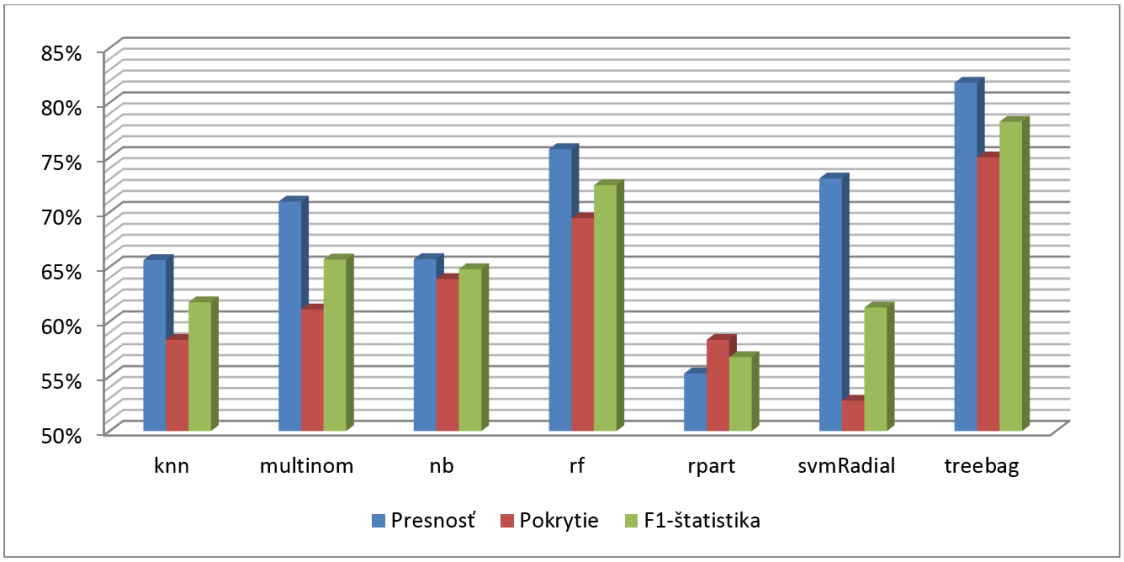
Záverečná časť skriptu počíta presnosť (*precision*), pokrytie (*recall*) a F1-štatistiku (*F-measure*) pre všetky modely a ukladá ich do premennej `results`.

```
> getPrecision <- function(x) as.numeric(unname(x$byClass[3]))
> getRecall <- function(x) as.numeric(unname(x$byClass[1]))
> getFmeasure <- function(x, y) 2 * ((x * y)/(x + y))
>
> n.row = length(pred.values.split)
> results <- NULL
> results <- dataFrame(
+ colClasses = c(Model = "character", Precision = "double",
```

```
+ Recall = "double", F.Measure = "double"), nrow = n.row)
>
> for (j in 1:length(pred.values.split)) {
+     conf.matrix <- confusionMatrix(pred.values.split[[j]]$pred,
+     pred.values.split[[j]]$obs, positive = "One")
+
+     precision <- getPrecision(conf.matrix)
+     if(is.na(precision)){ precision <- 0 }
+
+     recall <- getRecall(conf.matrix)
+     if(is.na(recall)){ recall <- 0 }
+
+     f.measure <- getFmeasure(precision, recall)
+     if(is.na(f.measure)){ f.measure <- 0 }
+
+     results[j, 1] <- names(pred.values.split)[j]
+     results[j, 2:4] <- c(precision, recall, f.measure)
+ }
```

Vypočítané výsledky presnosti, pokrytia a F1-štatistiky jednotlivých predikčných modelov možno zobraziť pomocou príkazu `print(results)`:

```
> print(results)
       Model  Precision     Recall  F.Measure
1        knn    0,65625  0,5833333  0,6176471
2   multinom  0,7096774  0,6111111  0,6567164
3         nb  0,6571429  0,6388889  0,6478873
4         rf  0,7575758  0,6944444  0,7246377
5      rpart  0,5526316  0,5833333  0,5675676
6  svmRadial  0,7307692  0,5277778  0,6129032
7    treebag  0,8181818       0,75  0,7826087
```



Obrázok 2.5 Porovnanie výsledkov použitých klasifikačných modelov.

Krok 8: Interpretácia výsledkov

Obrázok 2.5 znázorňuje graf výsledkov presnosti, pokrytia a F1-štatistiky modelov, ktoré sa vypočítali v kroku 7. Z výsledkov vyplýva, že *tree bag* model dosiahol najlepšie hodnoty. Presnosť

modelu je 82%, pokrytie 75% a F1-štatistika 78%. Z výsledku presnosti sa dá odvodiť, že *tree bag* model produkuje v priemere 18% falošne pozitívnych prípadov – teda klasifikuje triedu ako chybnú, hoci v skutočnosti neobsahuje chyby. Podobne z výsledku pokrytia sa dá odvodiť, že model produkuje 25% falošne negatívnych prípadov – klasifikuje triedy ako bezchybné, hoci obsahujú chyby.

2.6 Zhrnutie

Mýliť sa je ľudské, rovnako je ľudské aj poučenie sa z chýb. Dolovanie v repozitároch chýb ponúka možnosti na automatizáciu procesu strojového učenia a generovania užitočných odporúčaní, ktoré môžu pomôcť pri identifikácii chýb v zdrojovom kóde a predikcii výskyt budúcich chýb. Tieto prístupy umožňujú odhaliť miesta v zdrojovom kóde, ktoré sú náchylné na chyby a ktorým treba venovať zvýšenú pozornosť pri vývoji a testovaní. Tieto informácie môžu pomôcť zabrániť nepriaznivým situáciám, znižujú čas a náklady na testovanie, šetria ľudské zdroje, ktoré by boli potrebné pri neskoršom odhalení chýb.

Literatúra

[1] Čubranić, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: a project memory for software development. IEEE Trans. Software Eng. 31(6), 446–465 (2005). doi:10.1109/TSE.2005.71

[2] Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 23–32 (2003). doi:10.1109/ICSM.2003.1235403

[3] Herzig, K., Just, S., Zeller, A.: It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 392–401 (2013). doi:10.1109/ICSE.2013.6606585

[4] Kuhn, M.: caret: classification and regression training. Version 4.76, R package (2011). URL http://cran.r-project.org/web/packages/caret/caret.pdf. [citované 9.9. 2013]

[5] Murphy-Hill, E., Murphy, G.C.: Recommendation Delivery. In *Recommendation Systems in Software Engineering*. Springer, London, UK, (2014). ISBN 978-3-642-45134-8, pp. 223-242.

[6] Prifti, T., Banerjee, S., Cukic, B.: Detecting bug duplicate reports through local references. In: Proceedings of the International Conference on Predictor Models in Software Engineering, pp. 8:1–8:9 (2011). doi:10.1145/2020390.2020398

[7] Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 499–510 (2007). doi:10.1109/ICSE.2007.32

[8] Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 531–540 (2008). doi:10.1145/1368088.1368161

3 Sledovanie interakcií programátora

Tradičné spôsoby vyhodnocovania práce programátora a postupu softvérového projektu vychádzajú z dát o ich výsledkoch. Sledovať a vyhodnocovať však môžeme aj proces vykonaný k dosiahnutiu tohto výsledku. Pod spoločným názvom dáta interakcií rozumieme činnosti, artefakty, nástroje a kontext programátora, ktoré súviseli s vývojom softvéru. Prínos sledovania a využitia interakcií programátora je motiváciou aktuálneho výskumu v softvérovom inžinierstve, aj napriek viacerým ohraničeniam, ktoré vyplývajú zo sledovania programátorov pri ich činnosti.

3.1 Vyhodnotenie práce programátora

Sledovanie vývoja softvérového produktu (softvér) je dôležitou súčasťou každého softvérového projektu pre zabezpečenie kvalitného výsledku. Vyhodnotenie kvality softvérového produktu najčastejšie pozostáva zo sledovania práce programátorov a ich výsledku. Ako vyhodnotiť kvalitu softvérového produktu je otázkou, na ktorú odpovedajú viaceré štandardy, ako napr. ISO 9126. Kvalitu softvérového produktu určujú jeho vlastnosti, napr. udržovateľnosť, efektívnosť, alebo aj znovupoužiteľnosť [2]. Vlastnosti softvérového produktu určujeme z výsledkov metrík, ako napr. počet riadkov, index udržovateľnosti, alebo objektové metriky [4]. Tradičné metriky aj vlastnosti, ktorých história siaha až do 70. rokov 20. storočia, sa zameriavajú na hodnotenie softvéru ako výsledku práce programátorov.

O vývoji softvéru však môžeme sledovať viac ako len zdrojové kódy a zavedené zmeny, ktoré sú len výsledkom práce programátorov. Zaujíma nás aj proces, ktorým programátor dospel k výsledku. Ak by sme v odporúčacích systémoch vychádzali iba zo zdrojových súborov, ochudobnili by sme sa o množstvo dát a tým pádom aj o možnosti získania hodnotných informácií,

ktoré z výsledkov práce programátorov nedokážeme identifikovať. Jednoduchým príkladom je vyhodnotenie náročnosti úlohy programátora iba na základe počtu zmenených riadkov kódu[4]:

- Programátor A opravil chybu zmenou 14 riadkov kódu,
- Programátor B implementoval novú funkcionalitu 1965 riadkami kódu.

Našimi otázkami sú napr. ktorá úloha bola zložitejšia, ktorý programátor vykonal viac práce alebo či daní programátori boli pri svojej práci efektívni.

Ak odpovedáme na tieto otázky len na základe počtu zmien v zdrojovom kóde, vo väčšine prípadov bude naše konštatovanie subjektívne, keďže nezohľadňujeme proces vytvorenia príslušných zmien. Čiastočným zlepšením je zahrnutie času, za ktorý programátori priniesli výsledky, t.j. od začiatku práce na úlohe (napr. prevzatie zadania v systéme pre správu úloh) do odovzdania výsledku. Programátor však počas práce nemusel pracovať iba na jednej úlohe, prípadne akútne prerušil prácu na úlohe a venoval sa identifikovanej chybe, ktorej riešenie malo väčšiu prioritu. Potom je v časovej zložitosti vyhodnocovanej úlohy zahrnutá aj ďalšia úloha opravenia chyby.

Z uvedeného jednoduchého príkladu je zrejmé, že pre dostatočné a objektívne vyhodnotenie práce programátora potrebujeme pracovať s podrobnejším dátami. Riešenie úlohy programátora A mohlo pozostávať zo spustenia opravovanej aplikácie, krokovania behu programu, opakovaní spustenia a hľadania príčiny, študovania dokumentácie, študovania zdrojového kódu, opravenia a overenia správnosti opravy. Zachytenie, spracovanie a vyhodnotenie týchto interakcií je motiváciou pre detailnejší objektívny opis práce, či aj jej vyhodnotenie.

Sledovanie interakcií programátora nám poskytuje podrobnejšie dáta o vývoji softvéru a procese vedúcom k získaniu výsledkov. Pod interakciami programátora rozumieme vykonané *činnosti*, použité *artefakty* a *nástroje*, ale aj *kontext* pri riešení úlohy. V tejto kapitole uvedieme možnosti získavania dát interakcií, ich spracovanie, vyhodnotenie aj použitie. Zároveň uvedieme aj existujúce systémy v tejto oblasti, bližšie sa zameriame na Mylyn a PerConIK.

3.2 Zaznamenávanie práce programátora

Dáta interakcií získavame z viacerých zdrojov, s ktorými programátor interaguje pri svojej práci, t.j. v rámci ktorých ho dokážeme sledovať. Hlavným zdrojom interakcií sú pre nás už naznačené systémy správy zdrojových kódov a systémy pre správu úloh a chýb, ale ďalej aj vývojové prostredie, operačný systém a ostatné aplikácie [1].

Dáta interakcií zatrieďujeme do nasledujúcich štyroch typov:

Činnosti – činnosť vykonaná programátorom, napr. navigácia v zdrojovom kóde vo vývojovom prostredí, upravovanie kódu, pohyby myšou, odovzdanie zdrojových kódov.

Artefakty – súčasti softvérového projektu a jeho procesov, s ktorými programátor pracuje, najmä zdrojový kód, ale aj dokumentácia, záznamy chýb, alebo aj zadania úloh.

[4] Metrika *počet riadkov zdrojového kódu,* angl. Lines of Code (LOC).

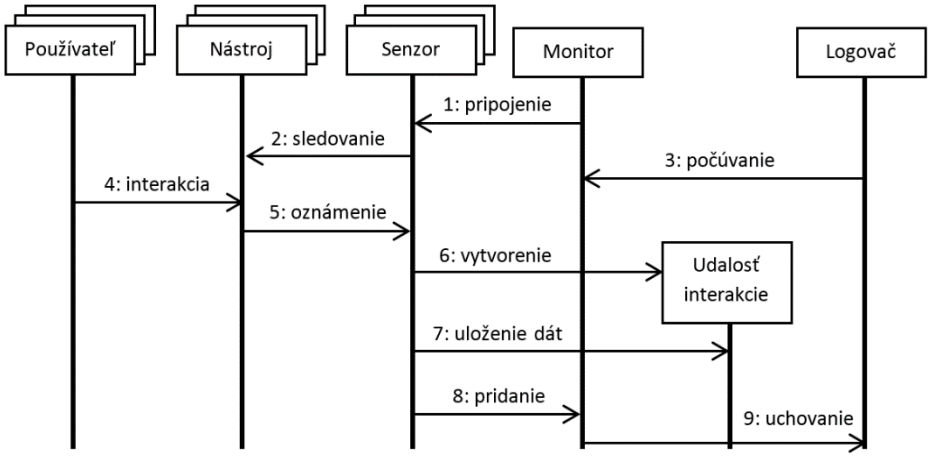
Nástroje – nástroje, ktoré programátor používa pri práci, napr. vývojové prostredie, webový prehliadač, systém pre správu úloh, systém pre správu zdrojových kódov.

Kontext – okolnosti ovplyvňujúce programátora a jeho prácu, napr. prostredie, v ktorom sa nachádza, jeho vedomosti, alebo aj artefakty súvisiace s úlohou, ktorú plní.

Uvedené typy dát interakcií nám umožňujú pozerať sa na zaznamenávanú prácu programátora z rôznych uhlov pohľadu. Uvažujme príklad, kedy programátor opravuje identifikovanú chybu v zdrojovom kóde, ktorú mu pridelili ako úlohu v systéme pre správu úloh. Potom:

- *Činnosťami* programátora sú čítanie popisu chyby, hľadanie dokumentácie k zdrojovému kódu, čítanie dokumentácie, aj práca so zdrojovým kódom. Opravenie chyby môže pozostávať zo študovania existujúcej implementácie, spustenia kódu, krokovania a sledovania jeho vykonávania. Následne po identifikovaní chyby ju programátor napraví, kontroluje správnosť, testuje, a nakoniec odovzdá riešenie.
- Pod *artefaktmi* v tomto príklade rozumieme opis chyby, študované dokumenty dokumentácie i súčiastky zdrojového kódu, s ktorými pracoval.
- Medzi *nástrojmi*, ktoré programátor pri riešení úlohy použil bol systém pre správu úloh (napr. Redmine[5], Jira[6], Bugzilla[7]), aplikácia pre prehliadanie dokumentácie (napr. webový prehliadač, Microsoft Word, Adobe Reader) a vývojové prostredie, v ktorom programátor pracoval so zdrojovým kódom (napr. Microsoft Visual Studio, Eclipse).
- *Kontextom* práce programátora bolo už samotné opravenie chyby v zdrojovom kóde, pretože to vplývalo na vykonávané činnosti. Kontextom bol aj čas, počasie, prostredie alebo jeho vedomosti ohľadom riešenia [9]. Programátorove vedomosti vplývali na potrebu študovania chyby, možnosti jej riešenia v zdrojovom kóde. Na druhej strane rušivé okolité prostredie a počasie mohlo negatívne vplývať na programátorovu produktivitu [3].

[5] Redmine, http://www.redmine.org/
[6] Jira, http://jira.atlassian.com/
[7] Bugzilla, http://www.bugzilla.org/



Obrázok 3.1 Komunikácia komponentov pre sledovanie činnosti programátora v nástrojoch. Číselné označenie šípok vyjadruje poradie ich komunikácie. Obrázok prevzatý z [10].

3.2.1 Zbieranie dát interakcií

Za jeden z typov dát interakcií považujeme nástroje používané programátorom pri svojej práci. Zároveň sú však tie pre nás aj zdrojom týchto dát, keďže v nich programátor vykonáva svoju činnosť a pracuje s artefaktmi. Obrázok 3.1 znázorňuje riešenie zbierania dát, ktoré pozostáva z pripojenia *senzora* k nástroju a zaznamenávania v *monitore* dát, do ktorého senzor odosiela zaznamenané dáta. Senzor počúva na udalosti činnosti programátora, najčastejšie cez aplikačné rozhranie nástroja (napr. kliknutie na tlačidlo, otvorenie súboru, spustenie kompilácie zdrojového kódu). Posledným komponentom v tomto modeli je *logovač*, ktorý uchováva dáta pre ich neskoršie spracovanie. Podľa umiestnenia logovača rozlišujeme medzi prístupmi:

- Lokálny prístup – zbierané dáta sa uchovajú na zariadení programátora
- Vzdialený prístup – zbierané dáta sa odosielajú na vzdialené centrálne úložisko

Motiváciou rozlíšenia umiestnenia logovača je problém ochrany súkromia programátora. Pri lokálnom prístupe má programátor väčšiu kontrolu nad zaznamenanými dátami a vie zabrániť nechcenému uverejneniu citlivých informácií (či už o svojej činnosti, alebo aj o samotných artefaktoch, s ktorými pracuje). Výhodou vzdialeného prístupu je však priamy prístup k dátam pre ich spracovanie a vyhodnotenie. Pre vyhodnotenie dát uložených lokálne je najprv potrebné ich zozbierať od jednotlivých programátorov a pracovať s nimi off-line. Priebežné odosielanie zaznamenaných dát na vzdialené centrálne úložisko umožňuje ich vyhodnocovanie v reálnom čase, aj v kombinácii naprieč rôznymi programátormi a priame získavanie informácií z dát.

Viaceré existujúce systémy pre získavanie dát interakcií kombinujú uvedené prístupy. Zaznamenané dáta sa uchovávajú lokálne na zariadení programátora, následne ich programátor môže odoslať do vzdialeného úložiska (systémy Mylyn, TeamVeawer, Blaze) [9].

3.2.2 Existujúce systémy

Zbieranie a spracovanie interakcií programátora je aktuálnou témou softvérového inžinierstva. Postupom času vzniklo viacero systémov, s ktorými sa v dnešnej dobe môžeme bežne stretnúť.

Jednotlivé systémy sa odlišujú úrovňou podrobnosti zbieraných dát, ich umiestnenia lokálne alebo vzdialene, alebo aj podporovanými zdrojmi dát. Úroveň podrobnosti dát vplýva na efektívnosť zbierania dát a možnosti ich použitia.

- Mylyn – verejne dostupné rozšírenie pre vývojové prostredie Eclipse[8]. Lokálny logovač uchováva agregované dáta o práci programátora so súbormi zdrojového kódu, ktoré je možné pridať k riešeniam úloh v systéme pre správu úloh v podobe kontextu úlohy. Kontext úlohy je výhodné použiť pri návrate k práci, alebo jej prevzatí po inom programátorovi.
- PerConIK – výskumný projekt zameraný na zbieranie dát, ich spracovanie a vyhodnotenie[9]. Oproti Mylyn sa vyznačuje zbieraním podrobnejších dát, a to ako z prostredia Eclipse, tak aj Microsoft Visual Studio do vzdialeného centrálneho úložiska. Spomedzi výskumných výstupov projektu môžeme spomenúť označovanie zdrojového kódu metadátami v podobe informačných značiek nesúcich informácie získanie práve z dát interakcií [11], odhadovanie expertízy programátora [8], alebo aj identifikáciu závislostí v zdrojovom kóde bez syntaktickej analýzy [6].

Hlavným problémom rozsiahleho rozšírenia a uplatniteľnosti systémov pre sledovanie programátora a je potreba vlastných rozšírení do existujúcich vývojárskych nástrojov a presvedčenie programátora ich používať. Podobné riešenia však už existujú pre zaznamenávanie činnosti používateľa na Webe. V kombinácii s narastajúcim trendom prechodu vývojových prostredí na Web[10] tak vidíme zjednodušenie rozšírenia prostredí o sledovanie činnosti programátora [5].

3.3 Reprezentácia a ukladanie dát interakcií

Hlavnou črtou dát interakcií je ich výskyt v čase. Pomocou interakcií sledujeme vývoj softvéru, ktorý je dlhodobá aktivita, a tak sa na dáta môžeme pozerať ako na prúd dát v čase. Tak, ako programátor pracuje s nástrojmi a svojou činnosťou, tak v čase vznikajú dáta interakcií, a tak ich aj v čase zaznamenávame a ukladáme logovačom. O každej zaznamenanej udalosti nás zaujímajú:

- informácie o programátorovi – pre odlíšenie viacerých programátorov,
- typ zaznamenanej udalosti – napr. otvorenie súboru, kompilovanie zdrojového kódu,
- špecifické údaje k udalosti – napr. názov súboru a projektu, alebo typ kompilácie,
- čas výskytu – kedy udalosť nastala,
- časové trvanie zaznamenanej udalosti,
- artefakt, ktorého sa udalosť týka – napr. súbor `ProductItem.java`, metóda `CreateOrder()`,
- nástroj, v ktorom bola udalosť zaznamenaná.

[8] Rozšírenie Mylyn pre Eclipse, http://eclipse.org/mylyn

[9] PerConIK – Personalized Conveying of Information and Knowledge, http://perconik.fiit.stuba.sk/

[10] Príkladmi vývojárskych nástrojov na Webe sú Microsoft Visual Studio Online, Ace, Codeanywhere, a ďalšie.

Rozmanitosťou zaznamenávaných typov dát sa vytvorenie efektívneho a normalizovaného relačného modelu stáva čoraz zložitejším. Každá zaznamenaná udalosť programátora môže byť opísaná špecifickými dátami, čo vedie k zložitej štruktúre dát. Vhodnejším spôsobom reprezentácie je opísať udalosti množinou RDF trojíc[11] „podmet-prísudok-predmet“. Ústredným objektom je výskyt udalosti, ktorá je pomocou predikátov prepojená s ostatnými objektami jej informácií. Ako príklad uvádzame udalosť upravenia metódy:

```
<event rdf:type interaction:SwitchToDocument>
  <event interaction:SolutionName TicTacToeGame />
  <event interaction:hasDuration 200 />
  <event interaction:concerns java?name=newGame />
  <java?name=myMethod rdf:type artefact:Method />
  <java?name=myMethod artefact:partOf java?name=TicTacToe.Board />
</event>
```

Spomedzi najčastejšie zaznamenávaných interakcií programátora vo vývojovom prostredí môžeme spomenúť nasledujúce, rozdelené podľa častí alebo použitia nástroja, kedy ich sledujeme:

- Editor zdrojového kódu:
  - pozícia textového kurzora v súbore zdrojového kódu,
  - aktuálne zvolený element zdrojového kódu (metóda, trieda, premenná),
  - aktuálne zobrazený rozsah riadkov súboru,
  - zmena znaku, riadku alebo elementu v súbore,
  - navigácia k definícii použitého elementu,
  - kopírovanie, prilepenie alebo vystrihnutie fragmentu zdrojového kódu.
- Strom štruktúry projektu:
  - založenie nového projektu,
  - otvorenie projektu alebo súboru zdrojového kódu,
  - pridanie nového súboru, zmazanie súboru,
  - prepnutie sa medzi súbormi zdrojového kódu.
- Vyhľadávanie:
  - vyhľadanie výrazu, získanie výsledkov,
  - prezeranie výsledkov a navigácia zo zvoleného výsledku do zdrojového kódu,
  - nahradenie výrazov novým výrazom.
- Ladenie zdrojového kódu:
  - odchytenie miesta zdrojového kódu (angl. breakpoint),
  - posunutie sa o krok ďalej,
  - založenie sledovania premennej,
  - zmena hodnoty premennej počas vykonávania.

[11] RDF – Resource Description Framework, http://www.w3.org/RDF/

3.4 Spracovanie dát interakcií

Ako sme uviedli v úvodnej motivácii, dáta interakcií sa vyznačujú vysokou úrovňou podrobnosti o práci programátora práve vďaka nízkej úrovni sledovaných udalostí. Tým sa však dostávame do rozporu, kedy síce zaznamenávame podrobné informácie, no dát interakcií je veľké množstvo a je zložité ich spracovať a vyhodnotiť. Príkladom je upravovanie obsahu súboru zdrojového kódu, ktoré môžeme sledovať až na úrovni stláčania kláves klávesnice. Preto dáta interakcií tradične predspracujeme pred ich vyhodnocovaním *filtrovaním*, *agregovaním*, či *identifikáciou sedení* [9].

Zo zaznamenaných interakcií sa najčastejšie snažíme *odfiltrovať* nepodstatnú činnosť. Z uvedeného príkladu zaznamenávania písania znakov zdrojového kódu nás nemusia zaujímať znaky, ktoré programátor napísal a následne zmazal.

Agregovaním dát interakcií sa posúvame na vyššie úrovne granularity programátorovej činnosti. Napr. písanie znakov agregujeme na úpravy riadkov zdrojového kódu, cez úpravy jednotlivých súčiastok v súboroch zdrojového kódu, až po úpravy samotných súborov. Podľa úlohy použitia môže postačovať poznať zmenené riadky v zdrojovom súbore postupne v čase, než presné časové pečiatky zadanie všetkých znakov. Rôzne úrovne granularity dosiahneme týmito prístupmi:

- *Sémantický prístup* – vopred určená hierarchia typov interakcií, napr. kopírovanie fragmentu kódu medzi súčiastkami zdrojového kódu pozostáva z akcie označenia, kopírovania a prilepenia.
- *Heuristický prístup* – rozdelenie interakcií do aktivít programátora podľa zadaných pravidiel, napr. študovanie (navigácia bez úprav obsahu kódu), opravovanie chyby (zmena kódu a jeho kontrolovanie), vylepšenie kódu (študovanie kódu a následné úpravy).
- *Pravdepodobnostný prístup* – identifikovanie vzorov v dátach interakcií použitím metód dolovania v dátach a strojového učenia. Tak môžeme podobne určovať aktivitu programátora, či študuje kód, pridáva novú funkcionalitu alebo opravuje chybu.

Poslednou časťou predspracovania je identifikácia *sedení*, kedy rozdeľujeme interakcie do súvislých úsekov práce programátora. Dáta interakcií predstavujú prúd zaznamenaných udalostí, ktorý sa bežne prerušuje iba neprítomnosťou programátora. V skutočnosti však programátor pracuje postupne na úlohách. Aby sme z dát interakcií získali cenné informácie, rozdeľujeme ich na jednotlivé sedenia k úlohám. Projekt Mylyn to umožňuje tým, že sám programátor určí svoju úlohu a prepína sa medzi nimi [7]. Sledovanie študovania kódu alebo aj úprav je špecifické pre konkrétnu úlohu. Identifikácia sedení však nie je vždy možná zo strany programátora, preto sa ich snažíme identifikovať aj pomocou týchto pozorovaní:

- Činnosť programátora so stanoveným cieľom typicky trvá od 30 do 90 minút [10].
- Práca na úlohe môže pokračovať medzi pracovnými dňami, preto je nutné spájať sedenia.
- Programátor sa počas spoločného sedenia môže sústrediť na viacero úloh. Vtedy je vhodné rozlišovať medzi sedeniami pomocou artefaktov, s ktorými pracuje.

- Významná udalosť (spustenie nástroja, otvorenie projektu) môže naznačovať prepnutie medzi úlohami.

3.5 Použitie dát interakcií

Pri zbieraní dát interakcií sa stretávame s predsudkami samotných programátorov voči ich sledovaniu. Dôvodom je obava z ich použitia, napr. pre zlé hodnotenie, alebo ovplyvnenie výsledkov použitých metód, napr. pre odvodenie závislostí zdrojového kódu si programátor dáva väčší pozor na správny výber súborov pri navigácii [6]. Na druhej strane však, ako sme uviedli túto kapitolu, s dátami o ich práci dokážeme objektívnejšie porovnať náročnosť úloh medzi programátormi, aj keď počet zmenených riadkov zdrojového kódu naznačuje inak. Z dát interakcií dokážeme vyčítať skutočnú činnosť programátora a odhadnúť náročnosť riešenia problému.

Dáta interakcií často používame pre zefektívnenie práce samotných programátorov:

- Zmenšenie priestoru informácií, s ktorými programátor pracuje, napr. výber artefaktov súvisiacich s riešenou úlohou.
- Odporučenie existujúcich riešení práve riešeného problému.
- Predpovedanie chybovosti kódu programátora, napr. ak bol vytvorený v rušivom prostredí, alebo v krátkom čase bez kontroly [3].

Dáta interakcií používame aj na určenie expertízy programátora s danou technológiou alebo problematikou, čo nám uľahčí výber riešiteľa úlohy alebo špecifickej chyby spomedzi členov tímu programátorov [8]. Pokiaľ expert nie je dostupný, môžeme identifikovať aj programátorov, ktorí daný kód poznajú z predchádzajúceho štúdovania.

Sledovanie a vyhodnotenie dát interakcií v čase uľahčí programátorom sledovať, čo navzájom riešia. Príkladom je situácia, keď jeden programátor študuje kód, ktorý druhý programátor v danej chvíli mení. Ak by sme vychádzali len z výsledkov práce programátora (po odovzdaní zmien), programátor študujúci kód by zakladal svoje riešenie na neaktuálnej verzii kódu.

3.6 Zhrnutie

Sledovaním a vyhodnocovaním vývoja softvéru len podľa výsledkov práce programátorov sa ochudobňujeme o mnohé informácie, ktoré vznikli počas ich práce. Tieto informácie môžeme využiť ako pre samotné vyhodnotenie softvérového produktu a projektu, tak aj počas jeho vykonávania. Programátori môžu využiť identifikované informácie o postupe vývoja, ako je napr. expertíza programátora pre riešenie úlohy, zmenšenie informačného priestoru, návrat k úlohe, a ďalšie.

Sledovanie činnosti programátora sa spája s viacerými problémami a výzvami, napr.:

- Presvedčenie softvérovej firmy o sledovaní svojich programátorov, najmä v prípade ak sledovanie zabezpečuje iná spoločnosť [9].
- Presvedčenie programátorov a obava zo zásahu do súkromia.
- Vytvorenie vhodnej infraštruktúry pre rozhodnutý spôsob sledovania a ukladania dát.

- Vyhodnotenie výsledkov a ich prezentácia programátorom.

Získavanie a vyhodnocovanie dát interakcií je aj napriek uvedeným problémom aktuálnou témou softvérového inžinierstva. Vyhodnotenie softvérového produktu a jeho programátorov z výsledkov práce (odovzdané zmeny v zdrojovom kóde) sa ukázali byť nedostačujúce pre odhalenie skrytých vzorov v správaní programátora, či prevenciu voči chybám. Dáta interakcií rozdeľujeme do 4 kategórií – činnosti, artefakty, nástroje a kontext. Rozličnosť existujúcich prístupov a metód vychádza z používanej granularity dát, možnosti ich sledovania, ukladania aj reprezentovania. Sledovanie a vyhodnotenie činnosti programátora pochopiteľne nie je jednoznačné a závisí od prostredia a podmienok, v ktorých sa nachádzame.

Literatúra

[1] Bieliková, M., Polášek, I., Barla, M., et al.: Platform Independent Software Development Monitoring: Design of an Architecture. In: Proc. of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014), Springer, (2014), pp. 126-137.

[2] Boehm, B., Brown, J.R., Lipow, M.: Quantitative Evaluation of Software Quality. In: Proc. of the 2nd International Conference on Software Engineering (ICSE ’76). IEEE CS Press, (1976), pp. 592-605.

[3] Eyolfson, J., Tan, L., Lam, P.: Do Time of Day and Developer Experience Affect Commit Bugginess?, In: Proc. of the 8th International Working Conference on Mining Software Repositories (MSR ’11). ACM, (2011), pp. 153-162.

[4] Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering, vol. 20, no. 6. IEEE Press, (1994), pp. 476-493.

[5] Kats, L.C.L., Vogelij, R.G., Kalleberg, K.T., et al.: Software Development Environments on the Web: A Research Agenda. In: Proc. of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, (2012), pp. 99-116.

[6] Konôpka, Bieliková, M.: Software Developer Activity as a Source for Identifying Hidden Source Code Dependencies. In: Proc. of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2015), Springer, (2015), pp. 449-462.

[7] Kersten, M., Murphy, G.C.: Using Task Context to Improve Programmer Productivity. In: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14). ACM, (2006), pp. 1-11.

[8] Kuric, E., Bieliková, M.: Estimation of Student’s Programming Expertise. In: Proc. of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM ’14). ACM, (2014), Article No. 35.

[9] Maalej, W., Fritz, T., Robbes, R.: Collecting and Processing Interaction Data for Recommendation Systems. In: Recommendation Systems in Software Engineering (Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T., Eds.), Chapter 7, Springer Berlin Heidelberg, (2014), pp. 173-197.

[10] Maalej,W., Happel, H.J.: From Work to Word: How Do Software Developers Describe Their Work? In: Proc. of the International Working Conference on Mining Software Repositories (MSR ’09). IEEE CS Press, (2009), pp. 121-130.

[11] Rástočný, K., Bieliková, M.: Enriching Source Code by Empirical Metadata. In: Proc. of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM ‘14). ACM, (2014), Article No. 67.

4 Modelovanie programátora v odporúčacích systémoch

Na personalizáciu akéhokoľvek obsahu potrebujeme poznať záujmy a preferencie používateľa, ktorému obsah prispôsobujeme. Preferencie používateľov treba zachytávať v nepretržitom procese, aby sme dokázali identifikovať charakteristické črty správania daného používateľa. Model obsahujúci zachytené informácie sa štandardne označuje ako model používateľa. V doméne odporúčania v softvérovom inžinierstve však rozlišujeme používateľov vyvíjaných systémov a používateľov odporúčacích systémov, ktorí sú zároveň vývojármi vyvíjaných systémov. Odporúčacie systémy v softvérovom inžinierstve slúžia na podporu práce druhej menovanej skupiny. Pre jednoznačnosť procesu budeme ďalej v kapitole hovoriť o modelovaní vývojára softvéru a o modeli vývojára.

Odporúčacie systémy v softvérom inžinierstve (OSSI) slúžia na pomoc pri vývoji a údržbe softvéru. Z tohto pohľadu vieme odporúčanie rozdeliť na personalizované a nepersonalizované v závislosti od typu úlohy, napr. pomoc pri hľadaní miesta v kóde, ktoré spôsobuje chybu, personalizáciu nevyžaduje, pretože výsledok je jednoznačný. V iných prípadoch je však vhodná pomoc závislá od miery znalosti konkrétneho vývojára. Novému vývojárovi odporučíme knižnicu, ktorú jeho skúsenému kolegovi pripomínať netreba, nakoľko ju pravidelne využíva sám. V takomto prípade je preto personalizácia opodstatnená.

4.1 Využitie modelu vývojára

V prípade pojmu personalizácia (zosobnenie) je vhodné spomenúť tiež, že rozlišujeme dva základné typy. Prvým je prispôsobovanie obsahu (angl. customization), ktorá sa často pokladá za ručnú personalizáciu. V takýchto systémoch si môžu vývojári sami explicitne budovať svoje

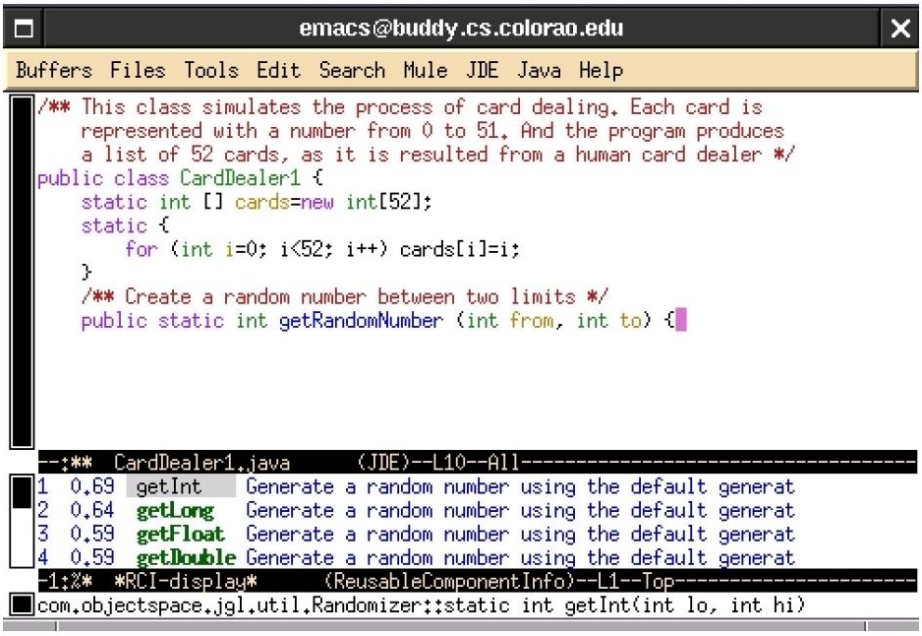
modely. Tieto systémy nazývame tiež *prispôsobiteľné*. Príkladom je MyYahoo!, kde si vývojári môžu nastaviť aký typ obsahu sa im bude zobrazovať na hlavnej stránke (novinové články, ceny akcií, počasie). V doméne vývoja softvéru je príkladom takéhoto prispôsobiteľného systému IBM Rational Application Developer. Zakladá sa na integrovanom vývojovom prostredí (angl. Integrated Development Environment – IDE) Eclipse a umožňuje vývojárom špecifikovať ich roly v procese vývoja systému (Java vývojár, Web vývojár, tester). Na základe rolí sa potom z prostredia odstránia nepotrebné komponenty [5]. Najjednoduchšou možnosťou je vytvorenie hašovacej mapy rola – vlastnosti prostredia. Techniky manuálneho prispôsobovania obsahu sú však dobre preskúmané.

Z tohto dôvodu sa v kapitole zameriavame skôr na druhý typ, teda na *adaptívne* systémy, ktoré slúžia na tvorbu automatických odporúčaní, založených na charakteristikách vývojára. Pri modelovaní je možné brať do úvahy informácie o vývoji systému, o interakcii s vývojovým prostredím ale tiež aj demografické údaje či záznamy z komunikačných sietí [1, 6, 8, 10].

V doméne softvérového inžinierstva a vývoja softvéru modely vlastností vývojára nemusia slúžiť len na personalizované odporúčanie zdrojového kódu či externých knižníc, ale tiež na odporúčanie informácií o iných vývojároch – expertoch schopných poradiť vývojárovi, ktorému odporúčame, s jeho úlohou. Príkladom takéhoto systému je ExpertiseRecommender [19], slúžiaci na odhaľovanie expertov na určitú vybranú časť kódu. Existujú teda dve hlavné oblasti, ktorými sa uberá modelovanie a následná personalizácia v OSSI – odporúčanie zdrojového kódu a odporúčanie expertov.

4.1.1 Odporúčanie zdrojového kódu

Príkladom prvého typu odporúčača je systém CodeBroker využívajúci model vývojára podporujúci adaptívne odporúčanie slúžiace na znovupoužitie existujúcich metód vhodných pre dokončenie aktuálnej úlohy vývojára [29]. Tomu sa v priebehu vývoja odporúčajú existujúce metódy s požadovanou funkcionalitou. Cieľom je okrem pomoci vývojárovi tiež zabránenie redundancií v podobe reimplementácie existujúcich metód. Systém slúži na odporúčanie Java metód v prostredí Emacs, kde sa vytváraný kód obohacuje o návrhy na dokončenie vyvíjanej metódy (Obrázok 4.1). Odporúčanie v tomto systéme vychádza zo signatúr vyvíjanej metódy, jej komentárov a ich porovnania s Java Develpment Kit API. Aby systém neodporúčal zbytočné návrhy, ktoré vývojár už pozná, CodeBroker si v modeli vývojára pamätá metódy, ktoré sa mu už v minulosti odporučili a pre ktoré teda existuje predpoklad, že ich vývojár už pozná. Pre nového vývojára model zachytáva zväčša len ním použité metódy. Model sa aktualizuje automaticky. Ukážka modelu obsahuje dve metódy – Obrázok 4.2. CodeBroker využíva vytvorený model na filtrovanie zoznamu API elementov, ktoré by sa potenciálne mohli odporučiť.



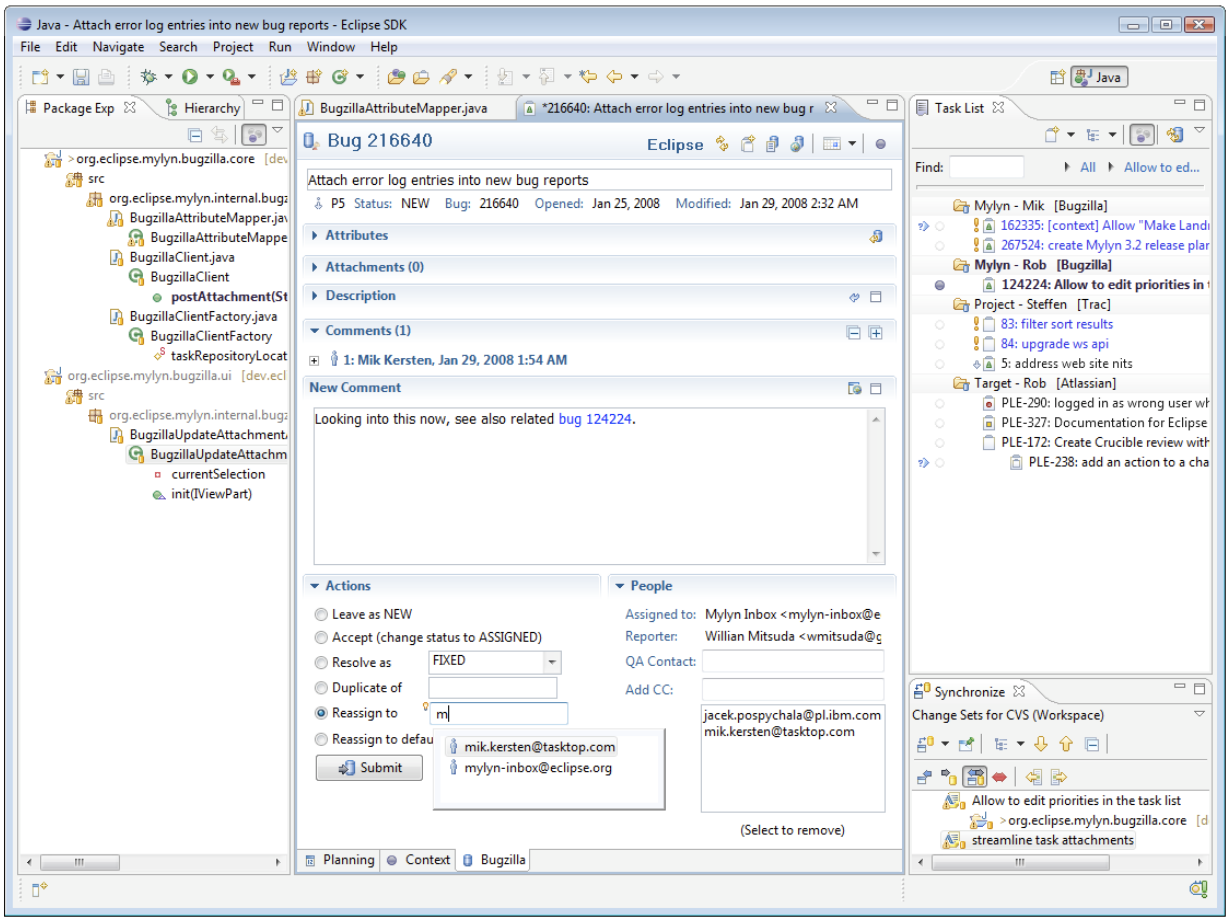
Obrázok 4.1. Ukážka odporúčania v systéme CodeBroker [27].

- java.applet
  - Applet
    - getParameterInfo (**added by Jeff at Thu 2 08:30:10 2000**)
- java.io
  - File
    - exists (Thu Nov 2 08:35:49 2000, Nov 2 08:15:10 2000, Nov 2 08:10:22 2000)
    - isAbsolute (Thu Nov 2 09:36:31 2000, Nov 2 09:19:15 2000)
  - CharArrayWriter
    - toCharArray (**added by Jeff at Thu 2 09:00:11 2000**)
- java.net (**added by Jeff at Thu 2 09:15:11 2000**)

Obrázok 4.2. Ukážka modelu vývojára používaného v systéme CodeBroker [28]. Odrážky prvej úrovne odkazujú na Java balíky, odrážky druhej úrovne odkazujú na konkrétne triedy v týchto balíkoch a odrážky tretej úrovne na metódy týchto tried.

Myšlienku filtrovania informácií posunul ešte ďalej systém Mylyn [14]. Ide o nástroj na adaptovanie vývojového prostredia Eclipse, ktorý vývojárovi zosobňuje prostredie tým, že mu zobrazí iba komponenty a informácie potrebné na aktuálnu úlohu (Obrázok 4.3). Pri adaptovaní však berie do úvahy vývojárovu históriu interakcií v prostredí Ecplise. Elementy navštevované častejšie alebo v nedávnej dobe majú väčšiu dôležitosť.

V modeloch systému Mylyn sa štruktúra ukladania relevantných komponentov a ich príslúchajúcej dôležitosti nazýva kontext úlohy. Jednotlivé dôležitosti má vývojár možnosť ručne meniť. Úlohou kontextu je zachytávať aktuálny záujem vývojára v konkrétnom čase. Kontexty úloh sa však môžu použiť na retrospektívnu obnovu komponentov viditeľných pre vývojára v konkrétnom bode v čase (napr. po nečakanom zlyhaní prostredia). Model vývojára použitý v Mylyn sa orientuje na úlohy a zachytáva výlučne informácie týkajúce sa aktuálnej úlohy.



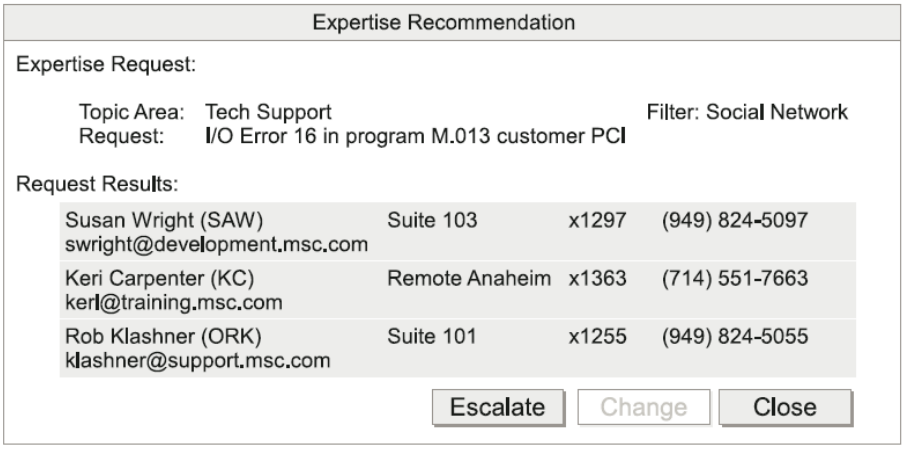
Obrázok 4.3. Ukážka odporúčacieho systému Mylyn v rámci nástroja Eclipse. Môžeme vidieť, že odporúčač redukuje komponenty IDE zobrazované vývojárovi v rámci aktuálne riešenej úlohy [22].

4.1.2 Odporúčanie expertov

OSSI môžu slúžiť okrem personalizovaného odporúčania zdrojového kódu tiež na odporúčanie expertov. Záujmom ich odporúčania v takomto prípade nie je žiadny komponent prípadne časť zdrojového kódu, ale iný vývojár, ktorý je expertom na úlohu alebo určitú časť kódu, ktorú aktuálne rieši vývojár, ktorému sa odporúčanie ponúka. Tento scenár nadobúda vysokú dôležitosť najmä v prípade veľkých a distribuovaných tímov [9].

Príkladom je systém Expertise Recommender [19], ktorý navrhli na pomoc pri technickej podpore so získaním kontaktu na pracovníkov, ktorí majú najlepšie predpoklady na vyriešenie určitej požiadavky zákazníka.

Obrázok 4.4 zobrazuje ukážku výsledku odporúčania v systéme Expertise Recommender.



Obrázok 4.4. Ukážka rozhrania zobrazujúceho vývojárovi zoznam expertov odporučených, na pomoc s aktuálne riešenou úlohou, odporúčacím systémom ExpertiseRecommender [19].

V tomto systéme sa odporúčania odvádzujú pomocou rozličných filtračných heuristík aplikovaných na spoločnú databázu modelov, v ktorej sa zdieľajú všetky zachytávané informácie o vývojároch v organizácii. V opisovanom systéme autori navrhli uchovávanie informácií z dvoch zdrojov – zo *systému na riadenie verzií* (angl. version control system, VCS) a zo *systému pre správu úloh* (angl. issue tracking system). Záznamy zo *systému pre správu úloh* slúžia najmä na hľadane technickej podpory, ktorá v minulosti úspešne dokázala vyriešiť požiadavku podobnú súčasnej. Požiadavka sa rozdelí na podproblémy a pre každý z nich sa vytvorí dopyt do databázy slúžiaci na vyhľadanie expertov, ktorí daný prodproblém v minulosti vyriešili. Výsledky jednotlivých dopytov sa potom posudzujú spoločne, pričom sa hľadá expert vyskytujúci sa v najviac výsledkoch.

Architektúra systému Expertise Recommender explicitne nezachytáva modely jednotlivých vývojárov ako samostatnú dátovú štruktúru. Tá exituje len pomocou dát z databázy, jej indexov a máp. Pritom však model vývojára zachytáva okrem základných informácií tiež zoznam modulov a vektor termov obsahujúci dôležité pojmy z problémov vyriešených vývojárom. Z toho vyplýva, že modely vývojárov sú konceptuálne entity, ktoré netreba explicitne reprezentovať ako jednotku implementácie v OSSI. Expertise Recommender je ukážkou systému využívajúceho modely vývojárov ako prvky znalostnej bázy a tiež na personalizáciu odporúčaní.

Expertise Recommender tiež poskytuje rozšírené odlišné módy vyhľadávania expertov, ktoré využívajú okrem príspevkov k zdrojovému kódu z *odovzdaní* (angl. commit) do VCS tiež informácie o zaradení pracovníkov v rámci štruktúry v ich organizácii (mód *Oddelenie*) a informácie zo sociálnych sietí (mód *Sociálna sieť*). Pri *Oddelení* sa odporučení vývojári zoraďujú s prihliadnutím na ich vzdialenosť z pohľadu pozícií v organizácii voči vývojárovi, ktorému sa odporúča. Teda čím bližšie sa nájdený expert nachádza (vlastné oddelenie), tým je priorita odporučenia vyššia. Mód *Sociálna sieť*, na druhej strane, uprednostňuje vývojárov podľa ich blízkosti k vývojárovi, ktorému odporúčame, v rámci ad-hoc sociálnej siete vytvorenej na základe vzťahov a predchádzajúcej komunikácie medzi vývojármi.

4.2 Znalosti o vývoji softvéru

V predchádzajúcej časti sme priblížili možnosti využitia modelu vývojára v OSSI. Pri samotnom odporúčaní, prípadne personalizácii však treba najskôr poznať vývojárove znalosti, preferencie a podobne. Preto sa v tejto kapitole zameriavame na objasnenie existujúcich trendov výberu informácií a spôsobu ich zachytenia. Znalosti o vývoji softvéru predstavujú znalosti, ktoré majú vývojári o systéme (systémoch), ktorý vyvíjajú a ich celkové skúsenosti s vývojom softvéru. Miera týchto znalostí je typicky získavaná skôr implicitne sledovaním akcií vykonaných vývojárom. Typickými predmetom sledovania sú 3 typy artefaktov:

- *záznamy o zmene* (angl. change logs) uložené vo VCS,
- *stopy interakcie* (angl. interaction traces) zbierané z IDE,
- *záznamy* (angl. records) uložené v *systémoch pre správu úloh.*

4.2.1 Záznamy o zmene kódu z VCS

Tieto dáta typicky odkazujú na odovzdania zdrojového kódu získané zo *systémov na kontrolu verzií zdrojového kódu* ako Git, SVN, CVS. Existuje viacero spôsobov využitia takéhoto typu dát, typicky ide o heuristiky.

Jedna z heuristík napr. predpokladá, že vývojár, ktorý mení určitú časť zdrojového kódu, jej aj rozumie. Táto heuristika teda zo záznamov o odovzdaniach zisťuje, kto konkrétne menil požadované riadky zdrojového kódu. Heuristika sa nazýva *Pravidlo riadku 10* (angl. Line 10 rule) a reálne ju na odporúčanie expertov jednotlivých modulov zdrojového kódu využíva vyššie opisovaný systém Expertise Recommender [19]. V tomto systéme model každého vývojára obsahuje informáciu o tom, ktorým modulom rozumie – teda ich menil ako posledný. Pri odporúčaní v systéme teda pre moduly súvisiace s aktuálne riešeným problémom identifikujeme expertov. Problémom takéhoto prístupu je určovanie expertízy API elementov a zdrojových kódov, pre ktoré nie je dostupná história zmien.

Inú heuristiku využíva systém CodeBroker, ktorý odporúčanie personalizuje pomocou vylučovania odporúčaní obsahujúcich API elementy, ktoré vývojár už pozná. Systém vychádza z predpokladu, že vývojár element respektíve knižnicu pozná, ak ju už použil. V takomto prípade mu systém z odporúčaní vylúči daný element a jeho podelementy, napr. metódy knižnice.

Pri identifikácii elementov, ktoré vývojár pozná, nemusí odporúčací systém pri modelovaní vychádzať len z priamo zmenených riadkov v kóde, ale môže v modeli vývojára uvažovať rovnako predchádzajúci a nasledujúci riadok [16]. Termy, ktoré sú zapísané do modelu, pochádzajú okrem konkrétnych elementov tiež z ich identifikátorov alebo komentárov.

Extrakcia elementov (metód) volaných v *odovzdaniach* je však výzva. Tu je totiž treba namapovať premenné objektov na ciele volaní metód. To zvyčajne robia kompilátory, avšak v prípade *odovzdaní* je proces ťažší, pretože v tomto prípade ide prevažne o malé časti kódu, ktoré boli zmenené. Pracovať pri každom *odovzdaní* s celým zdrojovým kódom je tiež veľmi neefektívne. Riešením výzvy zisku informácii z jednotlivých *odovzdaní* je napr. čiastočná analýza programu pomocou PPA [4].

4.2.2 Dáta o histórii interakcií

Druhý typ dát, ktoré sa využívajú na modelovanie záujmov vývojára v OSSI, je história jeho interakcií s jednotlivými elementami zdrojového kódu v projekte. Tento typ informácií sa odkazuje na sekvencie udalostí spustených akciami vykonanými vývojárom [17]. Príkladom systému využívajúceho tento typ dát je vyššie spomenutý odporúčač Mylyn [14]. Tento systém pri modelovaní adaptuje rozličné vlastnosti a komponenty IDE Eclipse a vývojárovi zobrazuje len tie, ktoré potrebuje pre aktuálnu úlohu. Z histórie jeho interakcií odporúčač vyberá softvérové artefakty vytvorené v rámci IDE, pomocou ktorých určuje relevanciu jednotlivých komponentov pre danú úlohu. Mylyn rozlišuje artefakty vytvorené priamou a nepriamou interakciou. Medzi udalosti priamej interakcie zaraďuje *označenia* (výber triedy a prezeranie jej elementov) a *úpravy*. Ostatné udalosti označuje ako nepriamu interakciu. Príkladom je *refaktorovanie* triedy – jej premenovanie sa označuje ako *príkaz*, premenovanie výskytov volaní v iných triedach sa označuje ako *propagácia*.

Okrem zmien elementov zdrojového kódu a navigácie medzi nimi však existujú aj ďalšie typy interakcie zachytávajúce veľkosť záujmu o daný element, napr. *strávený čas* (angl. linger time), ktorý sa vývojár elementu venoval , prípadne *rozsah posúvania* (scrolling), nepriamo odhaľujúci záujem vývojára [1]. Opisovaný systém Mylyn na určenie relevantnosti navštívených zdrojových súborov z pohľadu úlohy síce uvažuje, ako často a ako nedávno vývojár element navštívil, neberie však do úvahy *rozsah posúvania*. Dôvodom je, že autori štúdie [24] zistili, že rozsiahle *posúvanie* v skutočnosti predstavuje indikátor, že vývojár kódu nerozumie a stratil sa v ňom. Ďalším príkladom nepriamej interakciou je *počiatočné autorstvo* (angl. Initial authorship) v kódovom elemente, ktoré podľa [7] predstavuje silný faktor znalosti kódu.

4.2.3 Dáta zo systémov pre správu úloh

Tretí významný typ dát odhaľujúcich znalosť zdrojového kódu predstavujú dáta zo *systémov pre správu úloh*. Príkladom ich využitia sú „open source“ projekty, v ktorých je bežné, že sa chyby hlásia priamo vývojárom, ktorí danú časť projektu vytvorili a teda sú v tejto oblasti expertami. *Systémy pre správu úloh* sa však využívajú tiež komerčnými softvérovými spoločnosťami, kde slúžia na koordináciu vývojárov, testerov či manažérov. Scenár získavania dát z týchto systémov sa preto podobá prípadu „open source“ projektov. Podobné dáta sa získavajú zo systémov v centrách technickej podpory slúžiacich na zaznamenávanie hovorov.

Napr. odporúčací systém Expertise Recommender odporúča vývojárovi, ktorý dostal za úlohu vyriešiť zákazníkom nahlásenú chybu, tých kolegov (expertov), ktorí mu najlepšie dokážu pomôcť ju vyriešiť. Odporúčač pre každého vývojára, resp. pracovníka spoločnosti, vytvorí v databáze model, v ktorom zaznamenáva požiadavky, ktoré vývojár v minulosti vyriešil. Záznam sa skladá z troch typov informácií:

- Opis požiadavky – napr. vyriešenie chyby vyvíjaného systému
- Opis zákazníka, ktorý požiadavku inicioval
- Komponent systému, ktorý kvôli vyriešeniu požiadavky modifikovli

Pre každý typ informácie sa v modeli vývojára vytvorí vektor termov, ktorý sa postupne modeluje podľa vyriešených požiadaviek. Prvok vektora sa skladá z textového názvu a číselnej informácie o počte vyriešených požiadaviek daného typu. Vektor termov sa taktiež normalizuje celkovým počtom výskytu jednotlivých termov v celej databáze vyriešených požiadaviek.

Príklad systému kombinujúceho zber viacerých typov dát je [7] – jeho autori zbierajú implicitné dáta o histórii interakcií a o zmenách kódu vo VCS. Týmto spôsobom sa vytvára model vývojárovej znalosti elementov zdrojového kódu. Pri modelovaní autori pokladajú odovzdania zdrojového kódu do VCS za lepší indikátor znalosti v porovnaní s históriou interakcií.

N-rozmerný vektorový model vývojára zachytávajúci záznamy o chybách vo svojej práci použili tiež v práci [18]. Jednotlivé dimenzie modelu tu korešpondujú s indexovanými textovými termami, predstavujúcimi opis najnovšie vyriešených chýb.

4.2.4 Explicitný zber dát

V predchádzajúcich kapitolách sme opísali typy dát, ktoré sa pri modelovaní vývojára v OSSI využívajú najčastejšie. Spoločným aspektom týchto dát je spôsob ich zberu, ktorý je výlučne implicitný. Jeho výhodou je, že pri zbere netreba vývojára vyrušovať a teda môžeme zachytávať doslova každú jeho akciu. Nevýhodou naopak je, že na základe vykonaných akcií musíme vývojárove charakteristiky odhadovať pomocou rozličných heuristík. Modelované vlastnosti tak predstavujú iba odhad skutočnosti, ktorý nemusí byť vždy presný.

Omnoho priamejším riešením je explicitné získavanie informácií. Pre predstavu, v doméne „e-commerce“ sa explicitný zber typicky uskutočňuje formou hodnotenia položiek na definovanej n-árnej hodnotiacej škále. Jeho výhodou je, že informácie získavame priamo od zdroja, v tomto prípade vývojára, ktorý explicitne vyjadruje svoju znalosť prípadne preferenciu jednotlivých sledovaných elementov. Hodnotenie však vývojárov typicky obťažuje a jeho problémom je taktiež fakt, že vývojári často nedokážu dobre odhadnúť stupeň svojej expertízy.

Explicitná spätná väzba sa preto používa zväčša len ako doplnok k implicitnej alebo ju zbierajú skôr na veľmi vysokej úrovni abstrakcie, teda od vývojára sa požaduje vyjadrenie sa k rozsiahlym artefaktom, napr. celým knižniciam a nie ku konkrétnym metódam čí riadkom kódu. Príkladom odporúčacieho systému z domény OSSI je systém CodeBroker, ktorý dovoľuje vývojárovi ručne prispôsobovať svoj model. Vývojári tu tiež môžu označiť úroveň svojej expertízy pre rozsiahle časti zdrojového kódu na úrovni celých knižníc.

Pri porovnávaní pridanej hodnoty pochádzajúcej z explicitnej a implicitnej spätnej väzby sa v minulosti predpokladalo, že implicitná väzba poskytuje výsledky horšej kvality [12]. Nedávne štúdie jej však naopak prisúdili pozitívnejšiu perspektívu ako explicitne zbieraných dátam [26, 27].

4.2.5 Reprezentácia znalostí

Pri modelovaní vývojárových znalostí prípadne stupňa expertízy nestačí dáta len zbierať, nech už pochádzajú z akéhokoľvek zdroja. Ak má byť odporúčanie použiteľné v reálnom systéme, musí

byť kvalitné a čo najrýchlejšie. Je preto veľmi vhodné, aby model vývojára obsahoval dáta v predspracovanej forme, napr. už vo forme informácií.

Jedným z najjednoduchších spôsobov reprezentácie znalosti vývojára je zaznamenávať elementy programu, s ktorými interagoval. Pre tieto elementy predpokladáme, že ich vývojár pozná, pričom nás nezaujíma miera znalosti, ktorú by sme inak vedeli získať, napr. explicitnou spätnou väzbou. Takéto jednoduché modelovanie môžeme nájsť v systéme CodeBroker, kde sa model vývojára reprezentuje jednoducho ako zoznam signatúr pre všetky metódy. CodeBroker nemodeluje rozsah znalosti ale implicitne predpokladá, že vývojár má rovnakú znalosť každej metódy zachytenej v jeho modeli. Tento predpoklad však v množstve prípadov nepostačuje.

Vylepšenie predstavuje systém Expertise Browser, ktorý analogicky k Expertise Recommender, skladá model vývojára ako zoznam *atómov skúsenosti* (angl. experience atoms). *Atómy* v tomto prípade predstavujú miesta v zdrojovom kóde, skontrolované vývojárom [20]. Expertise Browser uchováva počet konkrétnych riadkov kódu, ktoré vývojár zmenil a na základe nich určuje expertízu vývojára alebo jeho organizácie. *Atómy* sa tiež môžu využiť na určenie najväčšieho experta pre konkrétnu časť zdrojového kódu.

Expertise Browser síce priniesol zlepšenie modelovania v podobe *atómov skúsenosti*, avšak jeho predpoklad je, že každý element zdrojového kódu rozširuje znalosti vývojára rovnako. Jediné váhovanie, ktoré systém v tomto smere prináša, je normalizácia váhy v závislosti od celkovej frekvencie výskytu modelovanej akcie u ostatných vývojárov rovnakého systému. Normalizácia funguje podobne ako algoritmus TF-IDF, s ktorým sa stretávame pri počítaní relevancie slov v textoch. To znamená, že použitie raritnejšej metódy predpokladá väčšiu znalosť ako použitie bežne volanej metódy a teda vývojárovi priradí vyššiu mieru znalosti. Aplikovateľným rozšírením prístupu by bolo zahrnutie času použitia metódy, teda akési zavedenie princípu zabúdania pre dávnejšie použité metódy.

Model vývojára sa však nemusí skladať len z elementov zdrojového kódu, s ktorými vývojár interaguje, ale môže tiež zachytávať *vzťahy* medzi jednotlivými elementami. Systém Mylyn pri zachytávaní nepriamych udalostí reprezentuje tiež *vzťahy* medzi elementami. Neukladá ich však do perzistentnej podoby priamo do modelu ale reprezentuje ich ako kontext úlohy. Výhodou tohto prístupu je, že kontext sa môže šíriť medzi ďalších vývojárov.

4.3 Informácie o organizácii a komunikačných sieťach

Pri odporúčaní v softvérovom inžinierstve je možné okrem informácií súvisiacich priamo s vyvíjaným systémom získať taktiež informácie o pozícii vývojára v rámci organizácie, v ktorej pôsobí alebo o jeho správach odoslaných prostredníctvom komunikačných sietí. Obidva zdroje môžu v OSSI pôsobiť v rozličných úlohách od pomôcky na filtrovanie odporúčaní vygenerovaných odlišným spôsobom, až po hlavný zdroj dát využívaného na generovanie odporúčaní.

Tento typ dát využíva napr. systém Expertise Browser, ktorý dovoľuje vývojárom preskúmať odporučených expertov prostredníctvom organizačnej štruktúry ich spoločnosti [20]. Naopak systém Expertise Recommender poskytuje možnosť filtrovať odporúčania tak, aby systém vývojárovi zobrazoval len expertov, ktorých má v kontaktoch na sociálnej sieti.

4.3.1 Zber dát

Informácie o organizačnej štruktúre v spoločnosti kde vývojár pracuje sa štandardne získavajú prostredníctvom *LDAP protokolu* (angl. Lightweight Directory Access Protocol) priamo zo záznamov spoločnosti. Bežne sa získavajú informácie o geografickej lokalite a pozícii zamestnanca v hierarchii. Vyhľadávacími dopytmi sa potom hľadajú zamestnanci z príslušnej lokality s rovnakou pracovnou rolou. Roly sa implicitne dolujú z platforiem na manažovanie softvérového vývoja, napr. z nástroja IBM Rational Team Concert [3] umožňujúceho vytváranie rolí a ich špecifikáciu. Podobnú funkcionalitu poskytuje jeho Java API.

Informácie o komunikácii vývojára sa môžu z komunikačných kanálov získavať z e-pošty či z rýchlej výmeny správ (angl. instant messaging, IM). Informácie sa potom dajú prezentovať ako grafy s hranami medzi odosielateľmi a prijímateľmi. Štandardne tu existuje tiež možnosť použiť *VCS systémy* a *systémy pre správu úloh*. Pri zbere dát z týchto zdrojov sa stretávame s viacerými výzvami. Jednou z nich je *namapovanie záznamov* z komunikácie na artefakty zdrojového kódu, ktorého sa týkajú, druhú výzvu predstavuje *dealiasing* rozličných emailových adries spojených s rovnakou osobou.

Odporúčací systém STEP_IN je príkladom OSSI, ktorý sa spolieha na informácie o organizácii a o komunikačných sieťach [30, 31]. Systém odporúča relevantných expertov a artefakty vychádzajúc z informácií z *VCS systémov, systémov pre správu úloh* a z *komunikačných sietí* vyžívaných prostredníctvom archívov emailov. Unikátna vlastnosť systému je, že okrem jednoduchého výsledku odporučenia uvádza tiež informáciu ako veľmi je odporučená osoba ochotná byť zapojená do komunikácie. Takto sa systém vyhýba situácii, kedy by niektorí vývojári neustále obťažovali iných, pričom by sami nepomáhali iným a taktiež situácii, kedy by boli určití experti preťažení množstvom úloh.

4.3.2 Reprezentácia

Komunikačná sieť je tvorená grafom, ktorý môžeme reprezentovať maticou *n*n*, kde *n* predstavuje počet uzlov grafu, teda počet vývojárov. Na základe hodnôt matice vieme hľadať a odporúčať vývojárov s najväčším možným prínosom pre určitú úlohu [2]. Vstupom odporúčania je okrem matice vývojárov tiež matica súvislostí medzi súbormi zdrojového kódu. Odporúčanie sa počíta pomocou požiadaviek na koordináciu násobením týchto matíc – Obrázok 4.5. Najprv sa vynásobí matica vývojárov s maticou súvislostí medzi súbormi. Takto dostaneme maticu súvislostí medzi vývojármi a súbormi, ktorá hovorí o predpokladanej znalosti súborov jednotlivými vývojármi – Obrázok 4.6.

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} f_1 & f_2 & f_3 & f_4 & f_5 \end{array} \\ \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{c} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \\ \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{bmatrix} . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \end{bmatrix} \end{array} = \begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{bmatrix} . & . & . \\ . & . & . \\ . & . & . \end{bmatrix} \end{array}$$

Obrázok 4.5. Rozmery vstupných matíc a výstupnej matice reprezentujúcej požiadavky na koordináciu.

$$\begin{array}{c} \\ d_1 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} f_1 & f_2 & f_3 & f_4 & f_5 \end{array} \\ \begin{bmatrix} . & . & . & . & . \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{c} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \\ \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{bmatrix} . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \end{bmatrix} \end{array} = d_1 \begin{array}{c} \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{bmatrix} . & . & . \end{bmatrix} \end{array}$$

Obrázok 4.6. Rozmery matice súvislostí medzi vývojármi a súbormi ukazujúca predpokladanú znalosť súborov jednotlivými vývojármi.

4.4 Údržba a uchovávanie modelu

Množstvo rozhodnutí vykonaných počas návrhu OSSI má vplyv na výslednú podobu modelu vývojára. Dve významné dimenzie v rámci návrhu sú údržba modelu a spôsob ukladania dát v ňom. Údržba má vplyv na to, ako bude model možné upravovať v čase. Táto vlastnosť sa nazýva *adaptívnosť*. Ukladanie modelu súvisí s komponentami OSSI, ktoré ho tvoria a ukladajú.

4.4.1 Adaptívnosť modelu vývojára

Najjednoduchší prístup k údržbe modelu je statické udržiavanie. Problémom tohto prístupu je, že v prípade, ak používame implicitne zbierané dáta a v modeli držíme všetky získané záznamy, tento typ modelov sa rýchlo stane zastaraným. Typicky sa používajú dve možnosti zlepšenia.

Prvá možnosť je dávkový modus, v prípade ktorého berieme do úvahy len aktuálne dáta definované pevne stanoveným časovým oknom, ktoré sa posúva v čase. Veľkosť okna si explicitne stanovuje vývojár uvedením časového obdobia, ktoré sa týka aktuálnej úlohy [20]. Takýto typ modelu sa nazýva adaptovateľný.

Druhá možnosť predstavuje dynamickejšie systémy, ktoré sa vývojára samy pýtajú, ako nastaviť parametre v jeho modeli. Príkladom je systém Mylyn [14], ktorý vývojára vyzve, aby stanovil rozsah aktuálnej úlohy. V tomto systéme sa model vývojára buduje z jednoduchého prúdu histórie interakcií, pričom dôležitosť starších dát sa automaticky znižuje. Dôvodom na explicitné identifikovanie aktuálnej úlohy je, že vývojár môže naraz riešiť viacero úloh. Opísaný proces preto radíme medzi poloautomatické. Príkladom je systém SpyWare, ktorý automaticky odhaduje ohraničenie úlohy. Identifikované ohraničenia vizuálne zobrazuje a identifikuje pracovné sedenia [23].

Automatická detekcia relevantných častí histórie interakcií, ktoré súvisia s aktuálnou úlohu, ostáva závažným problémom [28]. Jeho riešenie ale môže tkvieť vo vytvorení úplne adaptív-

nych modelov. V prípade takéhoto riešenia by bolo najpriamočiarejšie adaptovať model cez striktne definované časové okno. Namodelované informácie, ktoré sú natoľko staré, že prekročili hranicu takéhoto okna, však netreba vždy len mazať, aj keď to je pomerne používané riešenie. Alternatívou je ich naďalej uvažovať, ale so zníženou dôležitosťou.

4.4.2 Uchovávanie modelu vývojára

Pri vytváraní návrhu, ako sa budú v systéme modelovať vývojárove preferencie nás v neposlednom rade zaujíma tiež miesto fyzického uloženia modelu. Konkrétnejšie, budú sa dáta uchovávať na zdieľanom serveri alebo priamo lokálne u vývojárov?

Modely vývojárov založené na informáciách vydolovaných zo serverových repozitárov sa zvyknú ukladať na serveroch. Tieto repozitáre zahŕňajú systémy VCS a *systémy pre správu úloh*. Jednotné úložisko vyžadujú aj systémy využívajúce kolaboratívne informácie o viacerých vývojároch, či modely založené na informáciách o organizácii a informáciách z komunikačných sietí. Preto takéto systémy je vhodnejšie informácie držať na jednotnom mieste na serveri.

Najväčším protiargumentom proti použitiu serverových riešení je súkromie vývojárov. S ním sa existujúce odporúčacie systémy vysporadúvajú rozlične. Najjednoduchším príkladom je umožniť vývojárovi zakázať zber jeho aktivít systémom. Odporúčač Mylyn poskytuje „modus tichej aktivity“ [13], kedy sú vývojárove akcie pre systém neviditeľné. Avšak treba si uvedomiť, že po zákaze zberu aktivít je nástroj úplne nepoužiteľný a jeho používanie je zbytočné.

Pokiaľ odporúčací systém nefunguje na báze kolaboratívneho odporúčania, história interakcií sa typicky ukladá na strane klienta. Dôvodom nie je len ochrana súkromia vývojára, ale tiež objem dát pochádzajúcich od mnohých vývojárov, ktorý je vhodnejšie nechať distribuovaný a nespájať ho. V prípade ukladania dát na server je totižto často treba uvažovať kompresiu dát. Príkladom je spomenutý systém Mylyn, ktorý neukladá do modelov všetky dáta. Totožné udalosti zaoberajúce sa rovnakým programom sa agregujú do spoločných jednotných záznamov. V nich je viacero existujúcich akcií identifikovateľných na základe viacerých časových pečiatok. Takáto kompresia pre Mylyn nie je postačujúca a preto server odľahčuje ešte tým, že informácie o kontexte akcií, ktoré nepotrebuje pri kolaborácii, ukladá na strane klienta [14].

4.5 Zhrnutie

Adaptívne systémy využíva každý z nás dennodenne v rozličných doménach. Príkladom môžu byť spomenuté webové prehliadače či vyhľadávače. Adaptívnosť týchto systémov vychádza zo záujmov konkrétneho používateľa, ktoré sa zachytávajú a zbierajú vo forme modelu používateľa. V kontexte OSSI, modely vývojára umožňujú tvorbu personalizovaných odporúčaní rozličných artefaktov vývojárom ako aj odporúčaní iných vývojárov, ako expertov na požadované úlohy.

V kapitole sme sa pozreli na používané techniky modelovania znalostí a preferencií vývojárov v doméne OSSI. Tieto modely sa používajú v adaptívnych odporúčacích systémoch a vývojári na základe nich dostávajú od odporúčačov personalizované návrhy a odporúčania. Model vývojára v závislosti od požiadaviek konkrétneho odporúčacieho systému môže zachytá-

vať široký rozsah vývojárových charakteristík, znalostí, histórie jeho aktivít, informácií o jeho zaradení v organizácii kde pracuje a o jeho komunikácii na sociálnych sieťach.

V kapitole sme diskutovali množstvo problémov spojených s modelovaním vývojára pre potreby OSSI. Hlavnými zdrojmi dát pre modelovanie v tejto doméne sú *systémy na správu verzií VCS*, *história interakcií* či *systémy pre správu úloh*. Tieto dáta sa môžu používať na generovanie rozličných typov odporúčaní. Koncept personalizácie sa taktiež dôverne prepája s chybami použiteľnosti [1, 8, 10, 11, 21, 25]. Personalizácia v tejto oblasti súvisí najmä s pomocou vývojárom pri ich aktuálnej úlohe.

Literatúra

[1] Anand, S., Mobasher, B.: Intelligent techniques for web personalization. In: Revised Selected Papers of the IJCAI Workshop on Intelligent Techniques for Web Personalization. Lecture Notes in Computer Science, vol. 3169, pp. 1–36 (2005). doi:10.1007/11577935_1

[2] Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., Carley, K.M.: Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 353–362 (2006). doi:10.1145/1180875.1180929

[3] Cheng, L.T., de Souza, C.R.B., Hupfer, S., Patterson, J., Ross, S.: Building collaboration into IDEs. ACM Queue 1(9), 40–50 (2003). doi:10.1145/966789.966803

[4] Dagenais, B., Hendren, L.: Enabling static analysis for partial Java programs. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 313–328 (2008). doi:10.1145/1449955.1449790

[5] Findlater, L., McGrenere, J., Modjeska, D.: Evaluation of a role-based approach for customizing a complex development environment. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1267–1270 (2008). doi:10.1145/1357054.1357251

[6] Fischer, G.: User modeling in human–computer interaction. User Model. User-Adapt. Interact. **11**(1), 65–86 (2001). doi:10.1023/A:1011145532042

[7] Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E.: A degree-of-knowledge model to capture source code familiarity. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 385–394 (2010). doi:10.1145/1806799.1806856

[8] Gauch, S., Speretta, M., Chandramouli, A., Micarelli, A.: User profiles for personalized information access. In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds.) The Adaptive Web: Methods and Strategies ofWeb Personalization. Lecture Notes in Computer Science, vol. 4321, Chap. 2, pp. 54–89. Springer, Berlin (2007). doi:10.1007/978-3-540-72079-9_2

[9] Herbsleb, J.D.: Global software engineering: the future of socio-technical coordination. In: Proceedings of the Future of Software Engineering, pp. 188–198 (2007). doi:10.1109/FOSE.2007.5

[10] Jameson, A.: Adaptive interfaces and agents. In: Sears, A., Jacko, J.A. (eds.) The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications, 2nd edn., pp. 433–458. CRC Press, West Palm Beach (2008)

[11] Keenoy, K., Levene, M.: Personalisation of web search. In: Proceedings of the IJCAIWorkshop on Intelligent Techniques for Web Personalization. Lecture Notes in Computer Science, vol. 3169, pp. 201–228 (2005). doi:10.1007/11577935_11

[12] Kelly, D., Teevan, J.: Implicit feedback for inferring user preference: a bibliography. ACM SIGIR Forum **37**(2), 18–28 (2003). doi:10.1145/959258.959260

[13] Kersten, M., Murphy, G.C.: Mylar: A degree-of-interest model for IDEs. In: Proceedings of the International Conference on Aspect-Oriented Software Development, pp. 159–168 (2005). doi:10.1145/1052898.1052912

[14] Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11 (2006). doi:10.1145/1181775.1181777

[15] Lee, T., Nam, J., Han, D., Kim, S., In, H.P.: Micro interaction metrics for defect prediction. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 311–321 (2011). doi:10.1145/2025113.2025156

[16] Ma, D., Schuler, D., Zimmermann, T., Sillito, J.: Expert recommendation with usage expertise. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 535–538 (2009). doi:10.1109/ICSM.2009.5306386

[17] Maalej, W., Fritz, T., Robbes, R.: Collecting and processing interaction data for recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering. Springer, Berlin (2014)

[18] Matter, D., Kuhn, A., Nierstrasz, O.: Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 131–140 (2009). doi:10.1109/MSR.2009.5069491

[19] McDonald, D.W., Ackerman, M.S.: Expertise recommender: a flexible recommendation system and architecture. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 231–240 (2000). doi:10.1145/358916.358994

[20] Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 503–512 (2002). doi:10.1145/581339.581401

[21] Montaner, M., López, B., De La Rosa, J.L.: A taxonomy of recommender agents on the internet. Artif. Intell. Rev. 19(4), 285–330 (2003). doi:10.1023/A:1022850703159

[22] Mylyn. [cit 2015-2-4], dostupné na internete <http://eclipse.org/mylyn/images/mylyn-3.1-screenshot.png>

[23] Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of the IEEE International Conference on Program Comprehension, pp. 155–166 (2007). doi:10.1109/ICPC.2007.12

[24] Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: an exploratory study. IEEE Trans. Software Eng. 30(12), 889–903 (2004). doi:10.1109/TSE.2004.101

[25] Steichen, B., Ashman, H., Wade, V.: A comparative survey of personalised information retrieval and adaptive hypermedia techniques. Inf. Process. Manag. 48(4), 698–724 (2012). doi:10.1016/j.ipm.2011.12.004

[26] Teevan, J., Dumais, S.T., Horvitz, E.: Personalizing search via automated analysis of interests and activities. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 449–456 (2005). doi:10.1145/1076034.1076111

[27] White, R.W., Ruthven, I., Jose, J.M.: Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 57–64 (2002). doi:10.1145/564376.564389

[28] Ye, Y.: Supporting component-based software development with active component repository systems. Ph.D. thesis, Department of Computer Science, University of Colorado, Boulder (2001)

[29] Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 513–523 (2002). doi:10.1145/581339.581402

[30] Ye, Y., Yamamoto, Y., Nakakoji, K.: A socio-technical framework for supporting programmers. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 351–360 (2007). doi:10.1145/1287624.1287674

[31] Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y., Asada, M.: Searching the library and asking the peers: learning to use Java APIs on demand. In: Proceedings of the International Symposium on Principles and Practice of Programming in Java, pp. 41–50 (2007). doi:10.1145/1294325.1294332

[32] Ying, A.T.T., Robillard, M.P.: The influence of the task on programmer behaviour. In: Proceedings of the IEEE International Conference on Program Comprehension, pp. 31–40 (2011). doi:10.1109/ICPC.2011.35

5 Poskytovanie odporúčania

Vytvorenie užitočného odporučenia je iba prvý krok pri tvorbe odporúčacieho systému. Odporúčania sa musia poskytovať prostredníctvom používateľského grafického rozhrania takým spôsobom, ktorý umožňuje používateľovi oboznámiť sa s dostupnými odporúčaniami, rozhodnúť sa, či niektoré z nich má najväčší prínos a následne konať podľa vybraného odporúčania. Tvorcovia OSSI majú neľahkú úlohu pri návrhu vhodnej formy prezentovania odporúčaní. Odporúčania by sa mali poskytovať v správnom čase a formou, ktorá je pre používateľa dostatočne zrozumiteľná a dôveryhodná. Výber najvhodnejších grafických komponentov závisí od spôsobu prvotného kontaktu s používateľom, frekvencie odporúčania, formy poskytovania dodatočných informácii, stratégie hodnotenia rozhrania a získavania spätnej väzby od používateľa.

Odporúčací systém v softvérovom inžinierstve sa môže rozdeliť na dve časti: systémovú vrstvu, ktorá rozhoduje čo sa odporučí a vrstvu používateľského rozhrania, ktorá poskytuje vybrané odporúčanie. Používateľské rozhranie odporúčacieho systému musí prezentovať odporúčania spôsobom, ktorý umožní používateľom zvážiť a konať podľaodporúčania. Prezentácia odporúčaní väčšinou vyžaduje existenciu používateľského rozhrania. Existujú však prístupy, ktoré dokážu zobrazovať odporúčania bez použitia vlastné grafického rozhrania alebo rozhrania vývojového nástroja. Tvorcovia OSSI musia urobiť veľa rozhodnutí pri návrhu používateľského rozhrania odporúčacieho systému. Prijatie odporúčania používateľom závisí od rôznych faktorov. Vo všeobecnosti existuje päť faktorov, na ktoré by sa mali tvorcovia nástrojov zamerať: pochopiteľnosť, transparentnosť, vyhodnotiteľnosť, dôveru a načasovanie [9].

Pochopiteľnosť

Pochopiteľnosť vyjadruje schopnosť používateľa určiť, *čo* mu odporúčací systém navrhuje. Pochopiteľnosť má dva samostatné rozmery: *samozrejmosť* a *kognitívne úsilie*. Rozmer samozrejmosti opisuje, aké ľahké alebo ťažké je pre používateľa určiť druh poskytovaného odporúčania, napr. odporúčania duplicitných chýb v systéme na sledovanie chýb počas vytvárania záznamu o chybe. Systém odporúča používateľovi existujúce záznamy o chybách vo forme správ o chybách. Túto formu používateľ okamžite spoznáva a vie, že ide o odporúčanie. Keď je odporúčanie samozrejmé, tak treba len malé alebo žiadne školenie používateľa. Za menej samozrejmé sa považuje napr. odporučenie určitej vlastnosti softvérového artefaktu počas vývoja softvéru. Druhý rozmer opisuje koľko úsilia treba na pochopenie významu odporúčania, napr. pochopenie prečo odporúčací systém odporúča určitú metódu zdrojového kódu, ktorá by mohla byť relevantná pre práve upravovaný element, môže byť spojené s vysokým kognitívnym úsilím.

Ako môže tvorca nástrojov zlepšiť pochopiteľnosť systému? Nielson poskytuje vo svojej práci [11] niekoľko heuristických prístupov určených pre návrh používateľského rozhrania, z ktorých tri sú aplikovateľné na pochopiteľnosť v odporúčacích systémoch. Prvou heuristikou je "zhoda medzi systémom a reálnym svetom", v ktorej navrhuje, aby sa v systéme používali slová, frázy a koncepty, s ktorými sa používateľ už pravdepodobne stretol. Druhou heuristikou je "konzistencia a štandardy", v ktorej navrhuje, aby systém nasledoval konvencie a nenútil používateľa rozmýšľať, prečo dve rôzne slová alebo koncepty znamenajú to isté. Treťou heuristikou je "nápoveď a dokumentácia", ktorá navrhuje, aby bol systém použiteľný bez nápovede, ale vedel poskytnúť pomoc a informácie, keď je to treba.

Transparentnosť

Ďalším faktorom je transparentnosť. Okrem pochopenia toho, čo je odporúčanie, používateľ musí byť schopný zistiť, prečo sa odporúčanie poskytuje. Transparentnosť sa týka zdôvodňovania. Ak je jasné, prečo sa odporúčanie poskytlo, tak je transparentnosť vysoká. Ako príklad môžeme použiť spomenuté odporúčanie existujúcich záznamoch o chybách, kde je pomerne jednoduché poskytnúť transparentnosť, ak sa odporúčanie zakladá na textovej podobnosti, ktorá sa vyjadruje percentuálne alebo indikáciou hodnôt, ktoré sa zhodujú. Ak sa odporúčanie nezakladá na jednoduchých metrikách, tak je zdôvodnenie zložitejšie. Napr., ak odporúčací systém počas vývoja softvéru navrhuje použiť pravdepodobne efektívnejší príkaz, treba poskytnúť podstatné textové informácie alebo obrazové diagramy, ktoré vysvetlia ako nový príkaz nahradí pôvodný.

Ako môže tvorca OSSI zvýšiť transparentnosť? Vo všeobecnosti, čím viac informácií môže systém poskytnúť ohľadom zdôvodnenia odporúčania, tým lepšie. Tieto informácie sú dostupné pre odporúčací algoritmus a transparentnosť je iba záležitosťou ich poskytnutia používateľovi. Výzvou je urobiť to spôsobom, ktorý bude stále dostatočne zrozumiteľný. V odbore všeobecných odporúčacích systémov Ozok a kol.[12] radí používať konkrétne vysvetlenia, napr. „*Ľudia, ktorí používajú X používajú aj Y.*" namiesto „*Môže sa vám páčiť aj Y.*"

Vyhodnotiteľnosť

Keď používateľ chápe, čo je odporúčanie a prečo je práve k dispozícii, je stále treba posúdiť, či je odporúčanie relevantné a či ho chce používateľ vykonať. Odporúčania vo vývoji softvéru vyžadujú zvyčajne väčšie zhodnotenie, ako odporúčania v oblastiach zameraných na zákazníkov, napr. odporúčanie súvisiacich novinových článkov [8] sa môže vyhodnotiť v zlomku sekundy, odpovedaním na jednoduchú otázku: *„Je názov článku dostatočne zaujímavý?“* Naopak, odporúčanie potenciálnej duplicitnej chyby vyžaduje pochopenie odporúčaného záznamu o chybe a jeho porovnanie s vytváraným záznamom.

Vyhodnotiteľnosť súvisí s pochopiteľnosťou a transparentnosťou. Ľahko pochopiteľné a transparentné odporúčanie sa bude dať tiež pravdepodobne ľahko vyhodnotiť. Avšak, môže nastať situácia, kedy je odporúčanie pochopiteľne a transparentné, no napriek tomu je ťažko vyhodnotiteľné. Príkladom takéhoto prípadu je vyššie uvedená duplicita chýb.

Ako môže tvorca nástrojov zvýšiť vyhodnotiteľnosť v systéme? Ak je odporučenie alternatívou k tomu, čo už používateľ vykonal, tak by mal systém uľahčiť porovnanie existujúcej a navrhnutej činnosti alebo položky. V príklade s reportom o chybe môže systém zvýrazniť hlavné rozdiely v záznamoch o chybe. Ak treba vykonať viacero porovnaní, tak je vhodné použiť sumarizáciu rozdielov. Vo všeobecnosti platí, že systém môže uľahčiť používateľovi posúdenie hodnoty odporúčania porovnaním daného odporúčania s alternatívami, ktoré používateľ pôvodne zvolil.

Dôvera

Aj v prípade, keď je odporúčanie pochopiteľné, transparentné a hodnotiteľné, musí používateľ dôverovať poskytnutému odporúčaniu. Dôvera sa mení v závislosti od záväzku, ktorý musí používateľ vytvoriť, aby mohol využívať poskytnuté odporúčania. Príkladom je systém, ktorý odporúča zmenu v zdrojom kóde, a aj ju automaticky vykoná, vyžaduje vysoký stupeň dôvery. Ak by totiž automatická zmena vniesla chybu do zdrojového kódu, ktorú si vývojár nevšimne hneď, ale až niekedy v budúcnosti, môže byť odstránenie danej chyby spojené s veľkým úsilím a značným poklesom dôvery v následné automatické odporúčania.

Ďalším príkladom odporúčania v softvérovom inžinierstve sú nástroje na automatické dopĺňanie názvov metód v zdrojovom kóde. Tieto odporúčania nevyžadujú potrebu vysokej dôvery ako v predchádzajúcom príklade. Ak sa vývojárovi nepáči navrhnutá metóda, tak ju hneď vymaže a zabráni tak vzniku chyby.

Pre zvýšenie dôvery používateľa v odporúčanie sa môžeme uplatniť tieto tipy v OSSI:

- *Vybudovať ju* – začať s malými, skromnými odporúčaniami a postupne poskytovať väčšie a podstatnejšie odporúčania. Tento proces nemusí byť vždy optimálny, pretože používateľské rozhranie vývojových nástrojov neposkytuje vývojárom možnosť spätnej väzby, ktorá by odzrkadľovala aktuálny stupeň dôvery v odporúčania systému [10].

- *Požičať si ju* – požičať si dôveru od niekoho alebo niečoho, čo ju už má, napr. kolega v práci. Je lepšie poskytnúť odporučenie v zmysle *"Pri vykonaní podobných zmien sa váš kolega X často pozrel na triedu Y.*" namiesto všeobecného odporúčania "*Pri vykonaní tejto zmeny by ste sa mohli pozrieť aj na triedu Y.*"
- *Predstierať ju* – ľudia sú sociálne bytosti, ktoré sú ovplyvniteľné sociálnymi podnetmi, ktoré by mohli byť využité na zlepšenie prijatia odporúčania. Cialdini [4] uvádza vo svojom výskume 6 princípov presviedčania, ktoré by sa mohli použiť pri odporúčaní v softvérovom inžinierstve: reciprocita, záväznosť a konzistencia, sociálny dôkaz, autorita, sklon/náklonnosť a núdza. Tvorca OSSI môže napr. použiť princíp autority, ktorý sa odvoláva na fakt, že odporúčanie bolo odvodené z PhD. práce, ktorá analyzovala milióny riadkov zdrojového kódu.

Načasovanie

Dôležitá je otázka načasovania. Mnoho odporúčacích systémov v softvérovom inžinierstve poskytuje odporúčania, keď o ne používateľ požiada. V tomto prípade je odpoveď jasná – poskytni odporúčanie, keď sa oň žiada. V systémoch, ktoré sa musia samostatne rozhodnúť, už nemusí byť otázka načasovania úplne jasná. Niektoré systémy poskytujú odporúčania uprostred práce používateľa, čo môže byť kritické, napr. nástroj BeneFactor[12] môže vývojárovi navrhnúť, aby ručne refaktoroval vytváraný zdrojový kód [6]. V tomto prípade, čím dlhšie bude systém čakať s odporúčaním, tým menej času vývojárovi ušetrí, ak sa bude riadiť jeho odporúčaniami. Skoré a potenciálne časté odporúčanie rovnako ako odporúčanie uprostred práce môže byť rušivé. Náklady spôsobené prerušením môžu prevážiť výhody, ktoré prináša odporúčanie (za predpokladu, že si vývojár uvedomuje jeho prínos).

Ako môže tvorca OSSI znížiť rušenie spôsobené odporúčaniami? V rámci používateľského rozhrania navrhli niekoľko techník, ktorých úlohou je vytvárať rovnováhu medzi potrebou včasného odporúčania a potrebou vyhnúť sa rušeniu. Dobrým príkladom sú anotácie, ktoré informujú o dostupných odporúčaniach, no nenútia okamžite konať. Vývojár ich môže jednoducho odložiť alebo úplne ignorovať. Ďalšou technikou je využívanie "citlivého" upozorňovania [7], ktoré sa snaží odvodiť čas, kedy nie je vývojár uprostred dôležitej úlohy. Carter a Dewan [3] vytvorili systém, ktorý poskytuje pomoc vývojárom, keď deteguje, že sa zasekli a nevedia pokračovať v práci.

5.1 Používané stratégie tvorby používateľského rozhrania

Existuje niekoľko spôsobov prezentácie odporúčaní. V závislosti od účelu [9] ich môžeme rozdeliť na skupinu získania pozornosti používateľa a skupinu poskytovania ďalších informácií.

[12] Refaktorovací zásuvný modul do vývojového prostredia Eclipse.

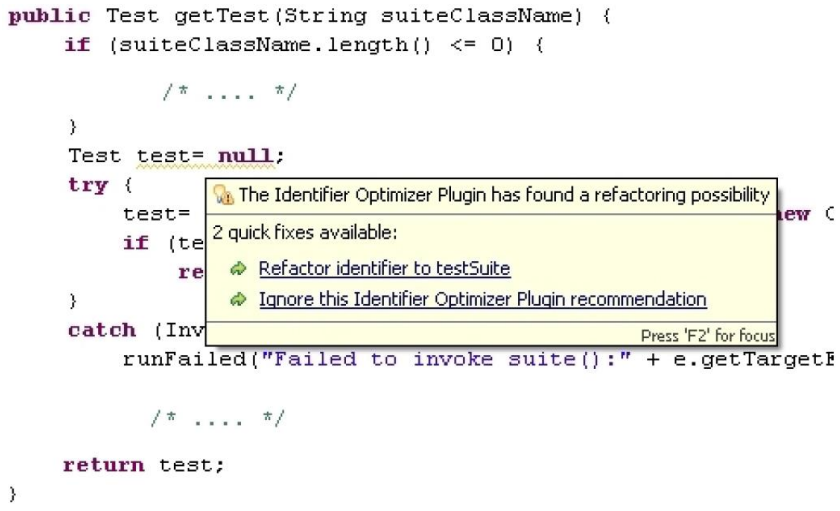
5.1.1 Získanie pozornosti používateľa

Ako sme naznačili v predchádzajúcich častiach dokumentu, forma prvotného kontaktu medzi používateľom a OSSI je jedným z kľúčových rozhodnutí, ktoré musia tvorcovia OSSI navrhnúť. Existujú dva hlavné prístupy: reaktívny a proaktívny[13]. Reaktívny prístup poskytuje používateľovi odporúčania vtedy, keď o ne požiada. Proaktívny prístup nevyžaduje požiadavku používateľa. Nástroj poskytuje odporúčania v závislosti od toho, ako bol naprogramovaný, napr. v určený čas alebo po zaznamenaní konkrétnej udalosti.

Proaktívny prístup je vhodný v prípadoch, kedy môžu včasné odporúčania výrazným spôsobom zlepšiť práve vykonávanú úlohu. Reaktívny prístup je vhodný v situáciách, kedy sa poskytnutie odporúčania nevzťahuje na časovo citlivé úlohy. Reaktívne odporúčania majú jednoduchú implementáciu. Tvorca OSSI poskytuje používateľom tlačidlá a klávesové skratky na vyvolanie odporúčania. Nájdenie takéhoto tlačidla alebo klávesovej skratky môže byť niekedy výzvou pre používateľa. Ešte väčšou výzvou je návrh a implementácia proaktívneho odporúčania. V nasledujúcej časti uvádzame existujúce používateľské rozhrania na uľahčenie proaktívneho odporúčania.

Anotácie

Anotácia je značkovanie aplikované na text. Spája osobitné odporúčanie s konkrétnym segmentom textu. Anotácie majú najčastejšie podobu podčiarknutia alebo zvýraznenia textu (Obrázok 5.1). Výhodou anotácií je ich rozšírenosť vo vývojových nástrojoch a ich jednoduchá implementácia. Anotácie sa objavujú vždy na vhodnom mieste, ktoré sa spája s konkrétnym miestom v kóde, ktorému by mal vývojár venovať pozornosť. Avšak anotácie nie sú vhodné pre všetky situácie, napr., ak sa anotácie objavujú v zdrojovom kóde až príliš často, alebo keď sú nedôležité, nepresné alebo ak sa prekrývajú. Pod pojmom časté anotácie si možno predstaviť také anotácie, ktoré sú roztrúsené po celom kóde a zaťažujú vnímanie používateľa do takej miery, že ich začne ignorovať alebo ich vypne.



Obrázok 5.1 Ukážka anotácie a poskytnutého odporúčania vo vývojovom nástroji Eclipse.

Ikony

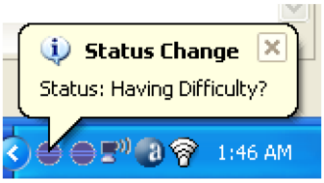
Ikony sú malé grafické obrázky, ktoré sa objavujú vo vývojovom prostredí. Typicky sa zobrazujú v okrajovej časti pracovného prostredia (Obrázok 5.2). Ikony zdieľajú mnoho výhod a nevýhod anotácií. Medzi výhody sa zaraduje vyššia informačná hodnota. Ikony môžu obsahovať obrázky, krátky text a symboly rôznych tvarov a farieb, ktoré sprostredkúvajú viac informácii v porovnaní s anotáciami. Tie poskytujú iba podčiarknutie textu pomocou čiar rôznych farieb a typu (rovná, vlnovková, čiarkovaná atď.). Nevýhodou ikon môže byť ich zobrazenie. Ikony sa nezobrazujú na rovnakom mieste ako anotácie, preto môžu byť menej nápadné a to najmä v prípade, ak je vývojár zaujatý inou činnosťou, kvôli ktorej ich môže prehliadnuť.

```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println("peers="  + peers.length);
System.out.println();
```

Obrázok 5.2 Ukážka ikony nástroja BeneFactor, ktorý odporúča refaktorovanie vyznačených riadkov zdrojového kódu.

Vyskakovacie okná

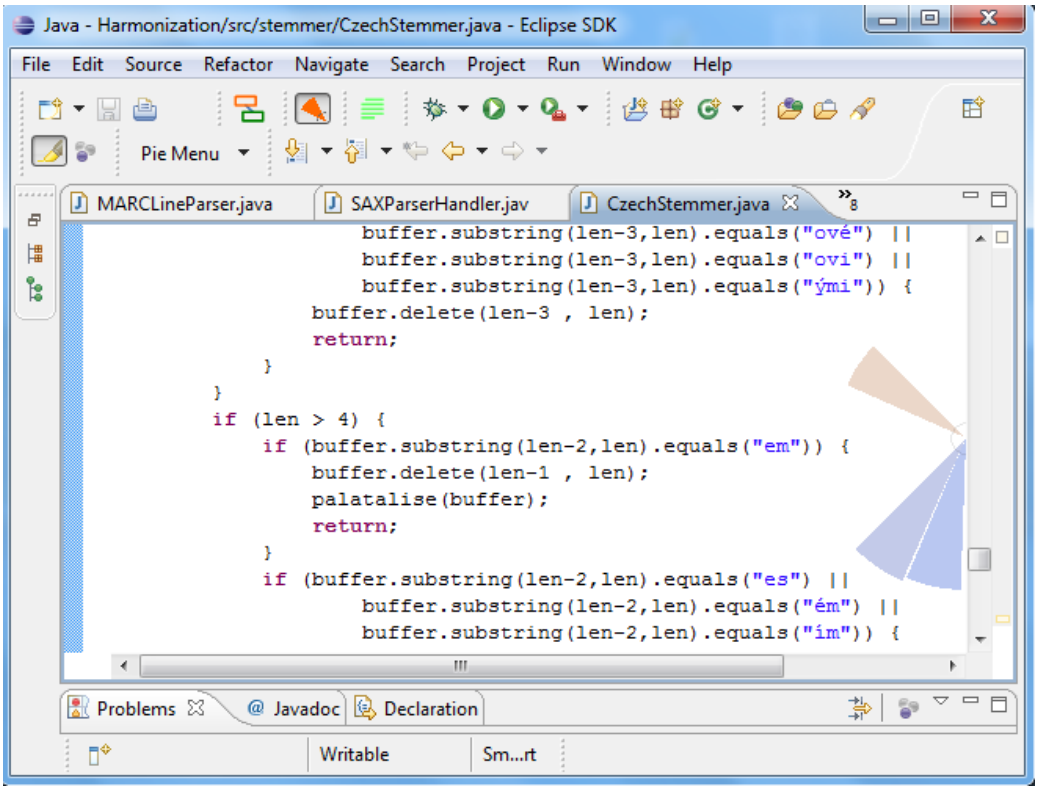
Odporúčania založené na forme vyskakovacích okien sú také, ktoré sa zobrazujú v novej vrstve nad existujúcim používateľským rozhraním, kedykoľvek sú dostupné nové odporúčania (Obrázok 5.3). Niektoré vyskakovacie okná môžu nútiť používateľov, aby ich potvrdili, iné zmiznú po uplynutí určitého času. Vyskakovacie okná majú rôzne stupne prchavosti, od úplného zmiznutia až po zanechanie ikony, na ktorú môže používateľ kliknúť a vyvolať zobrazenie odporúčania potom, čo vyskakovacie okno úplne zmizlo. Vyskakovacie okná sú vhodné v situáciách, kedy je veľmi dôležité získať pozornosť používateľa a usmerniť ju na práve poskytované odporúčanie. Vyskakovacie okná nie sú vhodným riešením v situáciách, keď je málo pravdepodobné, že sa používateľ bude riadiť odporúčaním alebo keď sa odporúčania vyskytujú veľmi často.



Obrázok 5.3 Ukážka vyskakovacieho okna s pomocou vývojárovi pri detekcii určitých problémov alebo dlhej neaktivite.

Indikačný panel

Indikačný panel predstavuje grafické rozhranie, ktoré má známe a nemenné umiestnenie na obrazovke, zvyčajne v jej okrajovej časti, aby nerušilo používateľa pri práci, no súčasne poskytovalo dostupné odporúčania a iné informácie bez explicitného vyžiadania (Obrázok 5.4). Odporúčania majú prevažne grafickú podobu, aby ich mohli rýchlo a jednoducho pochopiť. Indikačný panel v OSSI zvyčajne integruje niekoľko typov odporúčaní z jedného alebo viacerých zdrojov. Indikačné panely fungujú dobre v situáciách, keď sú odporúčania spojité a všadeprítomné. Nie sú vhodné v situáciách, keď používateľ nemá dostatočne veľkú obrazovku na ich zobrazenie.



Obrázok 5.4 Stench Blossom je modul do vývojového nástroja Eclipse. Každý "lupeň" predstavuje rozsah určitého zápachu v kóde, napr. najspodnejší lupeň indikuje pach s názvom veľká trieda.

Emailové notifikácie

Emailové notifikácie sa používateľovi poskytujú prostredníctvom emailu a nie priamo v integrovanom vývojovom prostredí. Emailové notifikácie sa používajú v situácii, kedy by mal všetky odporúčania zvážiť používateľ, ako aj v situácii kedy sa vyžaduje spolupráca viacerých strán, napr. spolupráca medzi projektovým manažérom a vývojárom. Hlavnou výhodou toho prístupu je fakt, že používateľov možno informovať o novom emaile podľa nastavení svojho poštového klienta, čo im poskytuje väčšiu mieru prispôsobovania. Nevýhodou prístupu je asynchrónna povaha emailov, ktorá nie je vhodná v prípadoch, keď treba okamžite reagovať na odporúčanie.

5.1.2 Poskytovanie ďalších informácií

V niektorých prípadoch chce odporúčací systém poskytnúť používateľovi doplňujúce informácie. Poskytnutie veľkého množstva informácií v úvodnom odporúčaní by mohlo demotivovať používateľa, aby aplikoval navrhnuté odporúčanie. Preto musia tvorcovia OSSI využívať vhodné formy poskytovania doplňujúcich informácií vzhľadom na situáciu a zámer.

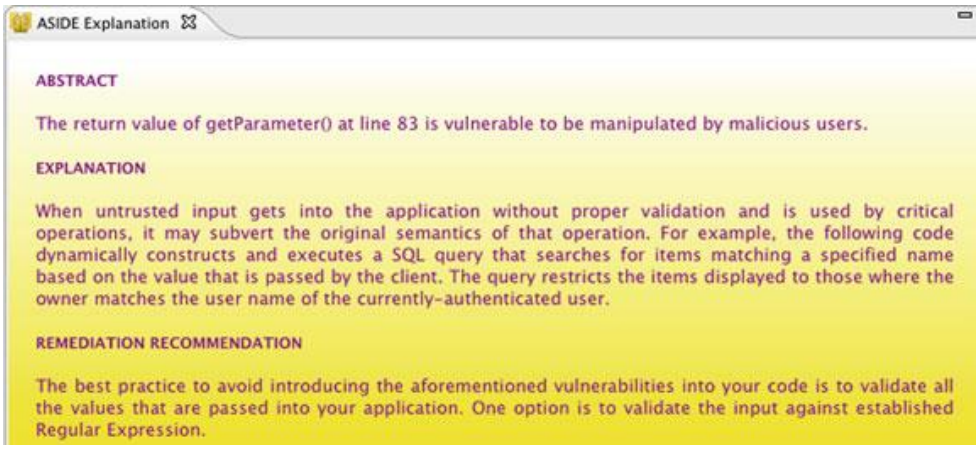
Informácie sa môžu poskytovať v tvare textu, transformácie alebo vizualizácie. Uvedené tvary sú vhodné pre proaktívne ako aj reaktívne odporúčacie systémy.

Textový tvar

Odporúčania majú textovú podobu. Textové opisy sa môžu byť zvýrazniť anotáciami, tabuľkami a rôznym formátovaním, ktoré uľahčujú pochopenie odporúčania.

Príkladom nástroja, ktorý poskytuje doplňujúce informácie textovou formou je ASIDE [14]. Obrázok 5.5 uvádza príklad, kedy nástroj vysvetľuje možnú zraniteľnosť vstupov a navrhuje riešenie, napr. SQL injekcia. Ďalším nástrojom je Seahawk, ktorý odporúča vývojárom odpovede zo stránok Q&A [1].

Textové opisy sú vhodné v prípadoch, kedy sa vyžaduje zdôvodnenie súvislostí a presné pochopenie navrhnutého odporúčania. Forma nie je vhodná v situácii, keď majú používatelia málo času na prečítanie textu.



Obrázok 5.5 Program ASIDE poskytuje odporúčania pre vývojárov týkajúce sa bezpečnosti vyvíjaného systému [14].

Transformatívny tvar

Transformačné odporúčania možno vývojárovi prezentovať formou vplyvu prijatia odporúčania, napr. zmeny v zdrojovom kóde. Konvenčnejšie implementácie transformačného odporúčania zobrazujú náhľad zmeny v samostatnom okne. Obrázok 5.6 zobrazuje nástroj WitchDoctor [5] navrhujúci zmeny v kóde.

Transformačné odporúčania sú vhodné v situáciách, kedy je známy dôsledok odporúčania. Naopak, nemusia pracovať dobre v situáciách, kedy pochopenie dôsledku odporúčania zaberie vývojárovi množstvo času alebo keď vývojár musí spätne analyzovať transformáciu aj riešený problém.

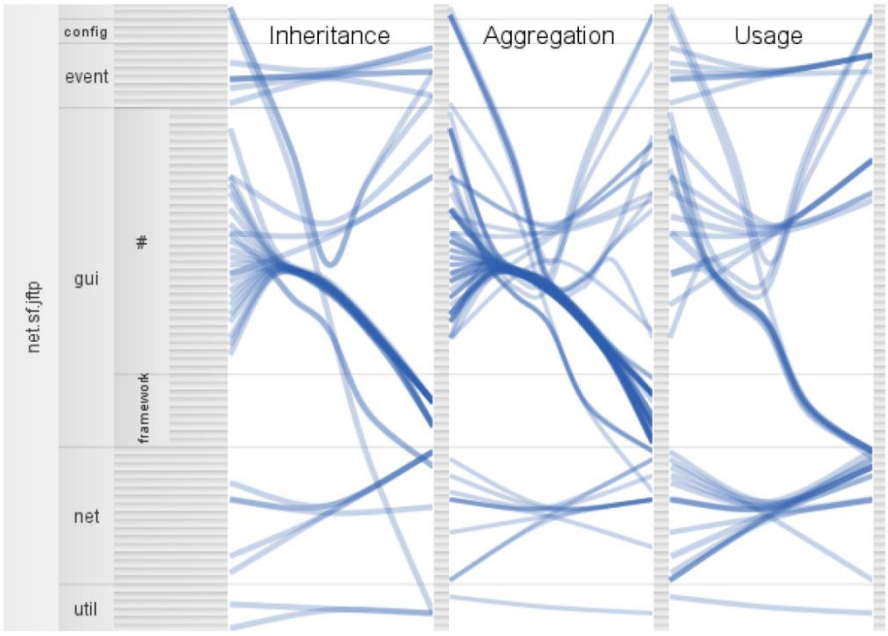
Vizualizačný tvar

Vizualizácie sprostredkúvajú odporúčania grafickým spôsobom. Vizualizácie sú vhodné v situáciách, keď sú odporúčania nepriame a vyžadujú úsudok vývojára. Vizualizácia zhromažďuje a zobrazuje informácie s nádejou, že vývojár bude konať na základe týchto informácií. Toto je v rozpore s mnohými textovými odporúčaniami, ktoré presne hovoria, čo má vývojár urobiť. Obrázok 5.7 uvádza príklad zobrazenia previazaní medzi triedami v projekte JFtp [15].

```
public static int fullSize()
{
    int size = 0;

    for(String string : list)
    {
        findSize();
    }

    return size;
}
```

```
public static int fullSize()
{
    int size = 0;

    for(String string : list)
    {
        size = findSize(size, string);
    }

    return size;
}
private static int findSize(int size,String string){
    int string_size=string.length();
    size+=string_size;
    return size;
}
```

Obrázok 5.6 Ukážka odporúčania nástroja WitchDoctor. Vľavo je pôvodný kód. Vpravo sa nachádza refaktorovaný kód s odporúčanou vygenerovanou metódou (text sivej farby).



Obrázok 5.7 Vizualizácia vzťahov tried v projekte JFtp, ktorá má pomôcť vývojárovi pri úpravách a refaktorovaní [15].

5.2 Stratégie návrhu a vyhodnotenia

Tvorcovia OSSI majú zložitú úlohu pri implementácií rozhrania, ktoré má poskytovať odporúčania efektívnym spôsobom. Prvým spôsobom ako to docieliť je určenie miery angažovanosti (úsilia a času) na vytvorenie správneho používateľského rozhrania. Na jednej strane stoja nástroje, ktoré majú len minimálne alebo žiadne rozhranie. Príkladom je nástroj DoctorWitch, ktorý navrhuje zmenu zdrojového kódu priamo vo vývojom prostredí, neobsahuje vlastné rozhranie. Na druhej strane stoja nástroje, ktoré chcú ich tvorcovia rozšíriť v širšej komunite používateľov, preto musia vynaložiť väčšiu angažovanosť pri tvorbe ich používateľského rozhrania.

Druhým spôsobom je zvolenie stratégie na vytvorenie dobrého používateľského rozhrania, ktorá je zhodná s úrovňou angažovanosti. Typicky daná stratégia obsahuje dve symbiotické časti: návrh (vytvorenie používateľského rozhrania) a vyhodnotenie (určenie vhodnosti rozhrania). Pre nízku úroveň angažovanosti existuje niekoľko vhodných stratégií návrhu:

- *Inšpirovanie sa iným rozhraním* – ak je používateľské rozhranie navrhnuté riešiť problém určitým spôsobom, potom aj nástroj poskytujúci odporúčania, by mal byť navrhnutý s podobným rozhraním.
- *Maketa (mock-up)* – umožňuje návrhárom OSSI vytvoriť počiatočnú ideu používateľského rozhrania. Návrh má grafickú podobu, ktorá uľahčuje komunikáciu medzi vývojármi a používateľmi. Makety môžu byť vytvorené pomocou aplikácií alebo jednoducho na papieri.
- *Kognitívne prechádzky* – začínajú návrhom používateľského rozhrania (napr. maketa), tvorcovia OSSI potom skúmajú navrhnuté rozhranie tým, že prechádzajú množinou zadaných úloh, hodnotia zrozumiteľnosť rozhrania a ako ľahké je naučiť sa pracovať s ním.
- *Experimenty čarodejníka z krajiny Oz* – v tejto stratégii tvorcovia hodnotia rozhranie OSSI bez úplnej implementácie OSSI, namiesto toho sa používateľom poskytnú falošné (ale užitočné) odporúčania prostredníctvom používateľského rozhrania. Odporúčania poskytujú tvorcovia OSSI.
- *Heuristická evaluácia* – princíp tkvie v tom, že skupina expertov nezávisle hodnotí vytvorené rozhranie a porovnáva jeho prvky so zoznamom známych heuristík použiteľnosti.

Pri vysokej angažovanosti zahŕňajú stratégie návrhu rozhrania zber používateľských požiadaviek na rozhranie. Tvorcovia môžu považovať OSSI za časť softvéru a navrhovať jeho rozhranie pomocou metód, napr. participatívny návrh (*Participatory design*) alebo technika JAD (*Joint Application Design*) [2]. Stratégie vyhodnocovania pri vyššej angažovanosti:

- *testovanie* – tento typ hodnotenia využíva dve skupiny používateľov. Každej skupine sa poskytne odlišné používateľské rozhranie. Porovnaním správania používateľov sa zisťuje miera ich záujmu a správnosť navrhnutého rozhrania, napr. medzi hodnotiace parametre rozhrania patrí počet prijatých odporúčaní.
- *Kontrolované experimenty* – v kontrolovaných experimentoch sa rôznym skupinám používateľov poskytnú rozličné rozhrania OSSI. Experimenty sa uskutočňujú v kontrolovanom prostredí, kde sa sleduje správanie používateľov a zaznamenávajú sa činnosti spojené s používaním rozhrania. Rovnako ako v A/B testoch, aj tu sa zisťuje miera záujmu a iné výstupy, ktoré sa medzi skupinami navzájom porovnávajú.
- *Prípadové štúdie* – sú ako kontrolované experimenty s tým rozdielom, že sa nevykonávajú v kontrolovaných podmienkach ale priamo na pracovisku používateľa.

5.3 Zhrnutie

V kapitole sme preskúmali a opísali niektoré vlastnosti OSSI a rôzne používateľské rozhrania, ktoré pomáhajú implementovať dané vlastnosti. Návrh, implementácia a overenie správneho a použiteľného rozhrania zaberá veľa času a úsilia. Je to však nevyhnutný krok pri vytváraní úspešného odporúčacieho systému pre softvérové inžinierstvo. Tvorcovia OSSI musia poznať oblasť a problematiku, v ktorej chcú poskytovať odporúčania (napr. detegovanie a zobrazenie duplicitných záznamov o chybe, pomoc pri refaktorovaní zdrojového kódu, detekcia pachov kóde,

pomoc pri návrhu a pod.). Je nevyhnutné zvoliť primeranú formu prvotného kontaktu medzi používateľom a OSSI ako aj formu poskytovania ďalších doplňujúcich informácii a odporúčaní, aby používateľ dostal potrebné, zrozumiteľné a dôveryhodné odporúčania v správny čas.

Literatúra

[1] Bacchelli, A., Ponzanelli, L., Lanza, M.: Harnessing Stack Overflow for the IDE. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering, pp. 26–30 (2012).

[2] Carmel, E., Whitaker, R.D., George, J.F.: PD and joint application design: A transatlantic comparison. Commun. ACM **36**(6), 40–48 (1993). doi: 10.1145/153571.163265

[3] Carter, J., Dewan, P.: Design, implementation, and evaluation of an approach for determining when programmers are having difficulty. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 215–224 (2010). doi: 10.1145/1880071.1880109

[4] Cialdini, R.B.: Influence: Science and Practice, 4th edn. Allyn and Bacon, London (2000)

[5] Foster, S.R., Griswold, W.G., Lerner, S.: WitchDoctor: IDE support for real-time autocompletion of refactorings. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 222–232 (2012). doi: 10.1109/ICSE.2012.6227191

[6] Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 211–221 (2012).

[7] Horvitz, E., Jacobs, A., Hovel, D.: Attention-sensitive alerting. In: Proceedings of the Conference on Uncertainty in Artificial Intelligence, pp. 305–313 (1999)

[8] Konstan, J.A., Miller, B.N., Maltz, D., Herlocker, J.L., Gordon, L.R., Riedl, J.: GroupLens: Applying collaborative filtering to Usenet news. Commun. ACM 40(3), 77–87 (1997). doi: 10.1145/245108.245126

[9] Murphy-Hill, E., Murphy, G.C.: Recommendation Delivery. In *Recommendation Systems in Software Engineering*. Springer, London, UK, (2014). ISBN 978-3-642-45134-8, pp. 223-242.

[10] Nass, C.I., Yen, C.: The Man Who Lied to his Laptop: What Machines Teach Us about Human Relationships. Current Hardcover, New York (2010)

[11] Nielsen, J.: Ten Usability Heuristics. Alertbox (2005). URL. http://www.useit.com/alertbox/20030825.html. [last accessed on 11/14/05]

[12] Ozok, A.A., Fan, Q., Norcio, A.F.: Design guidelines for effective recommender system interfaces based on a usability criteria conceptual model: Results from a college student population. Behav. Inform. Tech. 29(1), 57–83 (2010). doi: 10.1080/01449290903004012

[13] Xiao, J., Catrambone, R., Stasko, J.: Be quiet?: Evaluating proactive and reactive user interface assistants. In: Proceedings of the IFIP TC13 International Conference on Human–Computer Interactaction, vol. 3, pp. 383–390 (2003)

[14] Xie, J., Lipford, H., Chu, B.T.: Evaluating interactive support for secure programming. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 2707–2716 (2012).

[15] Beck, F., Petkov, R., Diehl, S.: Visually Exploring Multi-Dimensional Code Couplings. In 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), Williamsburg, VA, USA. (2011)doi: 10.1109/VISSOF.2011.6069455

6 Charakteristiky a metriky na vyhodnocovanie odporúčačov

Dnešnú dobu charakterizuje snaha o neustále zlepšovanie. Tento trend môžeme sledovať aj v oblasti odporúčania. Aby sme vedeli povedať, ktorý odporúčač je najlepší, potrebujeme určiť charakteristiky a metriky, podľa ktorých budeme odporúčače vyhodnocovať. Podľa týchto charakteristík môžeme odporúčače nielen hodnotiť, ale aj vzájomne porovnávať. V tejto kapitole sa pokúsime zhrnúť základné charakteristické vlastnosti odporúčačov a najbežnejšie spôsoby merania týchto charakteristík s cieľom posúdenia, ktorý odporúčač je vhodnejší na použitie.

Poznáme množstvo charakteristík, podľa ktorých je možné vyhodnocovať odporúčače. Na základe charakteristík odporúčačov dokážeme lepšie určiť vhodnosť odporúčača pre konkrétny problém alebo odporúčače vzájomne porovnať. Charakteristiky odporúčačov môžu byť dvoch typov: kvantitatívne a kvalitatívne.

Kvantitatívne charakteristiky sa dajú vyjadriť číslom/množstvom, napr. počet správnych odporúčaní alebo percentuálne pokrytie systému, stabilita a podobne. Hodnoty týchto charakteristík možno vyjadriť číslom a vzájomne porovnávať.

Kvalitatívne charakteristiky sa nedajú merať. Zvyčajne len hovoria o prítomnosti alebo neprítomnosti určitej vlastnosti. Príkladom môže byť charakteristika dôveryhodnosť, ktorá je buď prítomná (odporúčač je považovaný za dôveryhodný) alebo nie je.

6.1 Charakteristiky a metriky

Nasledujúce časti venujeme opisu základných kvantitatívnych a kvalitatívnych charakteristík odporúčačov, ktoré identifikovali autori publikácie [6].

6.1.1 Správnosť

Typ charakteristiky: kvantitatívna

Jednou z najdôležitejších vlastností odporúčania je odporúčanie správnych položiek. Medzi výsledkami odporúčača by sa mali nachádzať tie položky, ktoré sú relevantné a zároveň by sa nemali zobrazovať tie, ktoré relevantné nie sú. Správnosť vyjadruje vzťah medzi množinou odporúčaní, ktoré navrhne odporúčač a množinou relevantných odporúčaní. Množina relevantných odporúčaní sa zvykne označovať pojmom zlatý štandard. Správnosť môžeme merať prostredníctvom odchýlky od zlatého štandardu alebo vzťahmi na výpočet relevantnosti.

Meranie odchýlky zobrazovaných položiek

Pri meraní odchýlky počítame rozdiel medzi odhadovanou hodnotou, ktorú určí odporúčač a skutočnou hodnotou položky. Táto hodnota vyjadruje jej poradie umiestnenia medzi navrhovanými položkami. Najpoužívanejšie vzťahy na výpočet odchýlky sú stredná kvadratická chyba (angl. Root Mean Square Error, RMSE) a absolútna odchýlka (angl. Mean Absolute Error, MAE). Vzťah na výpočet strednej odchýlky je:

$$MAE = \frac{\sum |\hat{r}_{ui} - r_{ui}|}{N} \tag{6.1}$$

kde r_{ui} vyjadruje skutočnú hodnotu položky i pre používateľa u, $\hat{r}_{ui}$ je odhadovaná číselná hodnota tejto položky a N je celkový počet položiek. Vzťah na výpočet strednej kvadratickej odchýlky je podobný, namiesto absolútnej hodnoty sa však používa druhá mocnina a výsledok sa odmocní.

$$RMSE = \sqrt{\frac{\sum (\hat{r}_{ui} - r_{ui})^2}{N}} \tag{6.2}$$

Aplikovanie tejto operácie slúži na to, aby sa potlačil vplyv malých chýb (hodnoty menšie ako 1). Obidve metódy na výpočet odchýlky môžeme normalizovať na interval <0,1>. Premenné r_{max} a r_{min} vyjadrujú maximálnu a minimálnu hodnotu položky z celého súboru.

$$RMSE_{norm} = \frac{RMSE}{r_{max} - r_{min}} \tag{6.3}$$

$$MAE_{norm} = \frac{MAE}{r_{max} - r_{min}} \tag{6.4}$$

Relevantnosť súboru položiek

Druhou metrikou, ktorá sa využíva pri hodnotení odporúčania, je relevantnosť položiek, napr. pri meraní kvality vyhľadávačov [3]. Každú položku môžeme zaradiť do jednej zo štyroch skupín, ktoré vznikli kombináciou relevantnosti a reálneho výskytu medzi odporúčanými položkami:

Tabuľka 6.1: matica relevantnosti položiek. Prvé písmeno T/F (z angl. true/false) označuje, či sú položky správne zaradené, druhé písmeno (z angl. positive/negative) označuje, či sa položky majú zobrazovať.

Položky	**Odporúčané**	**Neodporúčané**
Relevantné	TP	FN
Nerelevantné	FP	TN

Na meranie relevantnosti slúžia viaceré metriky založené na pomeroch:

- *presnosť* – odporúčané a relevantné položky ku všetkým odporúčaným (vzťah 6.5),
- *úplnosť* – odporúčané a relevantné položky ku všetkým relevantným (vzťah 6.6),
- *akurátnosť* – správne zaradené položky ku všetkým položkám (vzťah 6.7),
- *špecifickosť* – nerelevantné neodporúčané položky ku všetkým nerelevantným (vzťah 6.8).

$$\text{presnosť} = \frac{TP}{TP + FP} \tag{6.5}$$

$$\text{úplnosť} = \frac{TP}{TP + FN} \tag{6.6}$$

$$\text{akurátnosť} = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.7}$$

$$\text{špecifickosť} = \frac{TN}{FP + TN} \tag{6.8}$$

Z hľadiska hodnotenia systému sú najdôležitejšie metriky presnosť a úplnosť, nakoľko na hodnotenie vplýva počet správne odporúčaných položiek. V prípade zvyšných metrík (akurátnosť, špecifickosť) sa na pozitívnom hodnotení relevantnosti prejavuje aj nezaradenie nerelevantných položiek. Problémom je, že nerelevantných položiek je väčšinou oveľa viac ako relevantných, a preto je takýto výsledok skreslený. Presnosť a úplnosť sú preto kľúčové vlastnosti, ale zvyšovanie jednej z nich spôsobuje znižovanie druhej. Keďže ani jedna z nich nie je samostatne použiteľná, často sa využíva metrika F-skóre (označenie F), ktorá tieto hodnoty kombinuje (vzťah 6.9). Ide o kompromis medzi presnosťou a úplnosťou.

$$F = 2 \times \frac{\text{presnosť} \times \text{úplnosť}}{\text{presnosť} + \text{úplnosť}} \tag{6.9}$$

6.1.2 Pokrytie

Typ charakteristiky: kvantitatívna

Charakteristika pokrytia vyjadruje, aká časť systému je schopná využívať odporúčanie. Rozlišujeme dva pohľady na pokrytie [1]. Prvým je pokrytie katalógu vyjadrujúce množinu dostupných položiek, ktoré je schopný odporúčač navrhovať. Druhou je pokrytie predikcie, čo predstavuje množinu používateľských akcií, ku ktorým je systém schopný poskytnúť nejaké odporúčanie. Vzťah na výpočet pokrytia katalógu je:

$$\text{Pokrytie katalógu} = \frac{|S_r|}{|S_a|} \tag{6.10}$$

kde S_r je množina položiek, ktoré sú dostupné na odporúčanie a S_a je množina všetkých dostupných položiek. Keďže množina dostupných položiek obsahuje aj tie položky, ktoré nemusia patriť do oblasti záujmu používateľa, častejšie sa využíva váhované pokrytie katalógu [1]:

$$\text{Váhované pokrytie katalógu} = \frac{|S_r \cap S_s|}{|S_s|} \tag{6.11}$$

kde S_s je množina položiek, ktoré sú pre používateľa zaujímavé. Podobne sa určuje aj miera pokrytia predikcie, ktorá vyjadruje pomer používateľských akcií, ku ktorým je systém schopný vygenerovať odporúčanie ku všetkým vykonateľným akciám používateľským.

6.1.3 Rozličnosť

Typ charakteristiky: kvantitatívna

V niektorých prípadoch nie je vhodné, aby boli medzi odporúčanými výsledkami veľmi podobné položky. Pre používateľa môže byť náročné vybrať jednu položku z veľkého počtu veľmi podobných položiek. Smyth a McClave [7] definovali rozličnosť v množine položiek ako priemernú rozličnosť medzi všetkými dvojicami položiek. Rozličnosť možno vyjadriť viacerými metrikami, napr. cez euklidovskú alebo manhattanskú vzdialenosť.

6.1.4 Dôveryhodnosť

Typ charakteristiky: kvalitatívna

Niektorí používatelia odmietajú využívať odporúčania systému dovtedy, kým sa sami nepresvedčia, že je to pre nich užitočné. Zo začiatku ich považujú za obťažujúce. Odporúčač si preto musí vybudovať dôveru. Keď bude navrhovať odporúčania, ktoré nebudú použiteľné, používateľ ich začne ignorovať alebo odporúčanie úplne vypne. Najbežnejším spôsobom na zistenie dôvery je umožniť používateľovi vyjadriť spokojnosť s navrhovanými výsledkami – získať spätnú väzbu od používateľa. Spätná väzba môže byť implicitná (na základe správania používateľa) alebo explicitná (na základe vyjadrenia používateľa). Ak nechceme používateľa často obťažovať, môže-

me využívať aspoň implicitnú spätnú väzbu, ktorá spočíva v sledovaní používateľských akcií (napr. ako často používateľ využíva alebo odmieta jednotlivé odporúčania).

6.1.5 Spoľahlivosť

Typ charakteristiky: kvalitatívna, kvantitatívna

Spoľahlivosť vyjadruje, či je systém schopný ponúknuť aspoň nejaké odporúčanie, teda do akej miery sa môže používateľ spoľahnúť na to, že systém niečo odporučí. Aby bol odporúčač schopný poskytnúť odporúčanie pri rôznych aktivitách používateľa, musí najskôr sledovať správanie používateľa. Takto funguje aj odporúčanie webových stránok, keď webový prehliadač navrhuje stránky na základe histórie navštívených odkazov, prípadne histórie jemu podobných ľudí. Ak systém nemá žiadne informácie o používateľovi, musí si poradiť na základe iných dostupných atribútov, napr. informácii o prostredí, kde používateľ pracuje (lokalita, jazykové nastavenia systému, verzia operačného systému a pod.).

6.1.6 Novosť

Typ charakteristiky: kvalitatívna, kvantitatívna

Novosť predstavuje schopnosť odporúčača vysporiadať sa s novými vecami. Najčastejšie ide o prípady, kedy nemá ešte systém používateľa zmapovaného, alebo keď používateľ odrazu zmení svoj model správania. Reakcia na prudké zmeny správania používateľa je náročná. Jedno z možných riešení je odstrániť súčasnú bázu dát a vytvoriť si nový obraz o používateľovi. Trvalé odstránenie však nemusí byť vyhovujúce, lebo sa môže stať, že zmena správania je len dočasná, napr. keď je používateľ unavený alebo systém používa dočasne iný používateľ. V takom prípade je výhodnejšie vytvoriť nový model používateľa a priebežne porovnávať aktivity s aktivitami v predchádzajúcom používateľskom modeli.

6.1.7 Priaznivosť

Typ charakteristiky: kvalitatívna, kvantitatívna

Priaznivosť hovorí o tom, do akej miery je systém schopný poskytnúť neočakávané a zároveň správne odporúčania. Ide o situácie, kedy používateľ netuší, čo má robiť a odporúčač mu pomôže pokračovať. Na rozdiel od novosti, kde sa systém snaží zmapovať používateľské preferencie, je pri tomto parametri dôležitá správnosť odporúčania (v prípade novosti je skôr dôležité vytváranie modelu používateľa, pri priaznivosti to môžeme nazvať náhoda). Meranie sa uskutočňuje podľa počtu priaznivých odporúčaní počas určitej doby používania systému.

6.1.8 Užitočnosť

Typ charakteristiky: kvalitatívna, kvantitatívna

Užitočnosť predstavuje pridanú hodnotu, ktorú získa používateľ využitím odporúčania. Meranie tohto atribútu vychádza najčastejšie z toho, koľko práce alebo času ušetrí používateľovi využitie odporúčania. Je potrebné rozlišovať, či ide len o správne, alebo aj o užitočné odporúčanie. Správne odporúčanie nemusí byť vždy užitočné. Príkladom je doplnenie textu, kedy už používateľ vie, čo chce napísať, a rýchlejšie by to napísal bez použitia odporúčača. Listovanie v dlhom zozname odporúčaných položiek kvôli doplneniu jedného písmena je neefektívne. Príkladom je používanie technológie T9 (angl. Text on 9 keys) na starších telefónoch s deviatimi tlačidlami.

Meranie užitočnosti je možné dvomi spôsobmi. Prvým je subjektívne ohodnotenie užitočnosti používateľom. Druhý spôsob je taký, že výpočet realizuje odporúčač na základe toho, ako veľmi odporúčanie pomohlo. V prípade generovania zdrojového kódu sa dá vyjadriť počtom nových riadkov zdrojového kódu, ktoré nemusí písať používateľ manuálne.

6.1.9 Rizikovosť

Typ charakteristiky: kvalitatívna

Pri využívaní odporúčača je potrebné brať do úvahy aj riziká a potenciálne problémy, ktoré nastanú aplikovaním odporúčania. Predstavme si výber nového komponentu s určitou funkcionalitou. Každý používateľ má svoje preferencie, čo nesie väčšie riziko. Pre konzervatívneho vývojára je výhodnejšie použiť starší, ale overený komponent, ktorý sa už dlho používa a poskytuje potrebnú funkcionalitu. Pre iného vývojára je zase vhodnejšie nasadiť nový komponent, ktorý má síce menej funkcií, ale intenzívne na ňom pracujú vývojári a je predpoklad, že bude v budúcnosti aktualizovaný. Rizikovosť tkvie v nesprávnom výbere vhodného komponentu, ktorý vybrali na základe zlého posúdenia odporúčača.

6.1.10 Robustnosť

Typ charakteristiky: kvantitatívna

Robustnosť je schopnosť odporúčača tolerovať a potláčať falošné informácie. Ich zdrojom môžu byť neskúsení používatelia, ktorí sa dopúšťajú chýb, alebo úmyselná snaha používateľov, ktorí sa snažia vedome systém poškodiť. Na vyhodnotenie robustnosti systému sa využíva metóda porovnávania množiny odporúčaných položiek pred a po vložení falošných informácií. Sleduje sa miera zmeny odporúčania systému [5]. Mieru zmeny pre konkrétnu položku vyjadrujeme vzťahom:

$$\Delta i = \sum_{u \in U} \frac{\hat{\hat{r}}_{ui} - \hat{r}_{ui}}{|U|} \tag{6.12}$$

kde *i* je navrhovaná položka a Δi je zmena hodnotenia položky *i*. Premenné $\hat{\hat{r}}$ a $\hat{r}$ sú hodnotenia položky pred, resp. po vložení falošných informácií a U je množina používateľov. Väčšinou nie je podstatné posunutie číselnej hodnoty položky, ale či dôjde k zmene navrhntuej pozície.

6.1.11 Rýchlosť učenia

Typ charakteristiky: kvantitatívna

Rýchlosť učenia vyjadruje schopnosť odporúčača prispôsobovať sa zmenám a novému správaniu sa používateľov. Ide o opačnú vlastnosť k vlastnosti robustnosť. Čím rýchlejšie sa systém prispôsobuje, tým je rýchlosť učenia rýchlejšia, ale zároveň slabšia odolnosť systému voči falošným informáciám. Vyhodnocovanie rýchlosti učenia môžeme realizovať niekoľkými metrikami:

- čas, kedy systém opätovne odporúča správne výsledky pri zmene záujmu používateľa,
- čas, kedy systém dosiahne určitú úroveň správnosti pri nových používateľoch,
- správnosť odporúčaných položiek, ktoré systém dosiahne za určený čas.

6.1.12 Použiteľnosť

Typ charakteristiky: kvalitatívna, kvantitatívna

Aby sa dali odporúčania systému považovať za užitočné, musia ich byť schopní používatelia ľahko a efektívne použiť. Použiteľnosť odporúčača do veľkej miery závisí od rozhrania: akým spôsobom sa zobrazujú odporúčané položky, akým spôsobom sa dajú vyvolať či skryť odporúčané položky, ako položku aplikovať, veľkosť textu, farbu textu, priehľadnosť textu a ďalšie. Ide často kľúčový faktor pri získavaní dôvery používateľa.

Príkladom aplikovania položky je vygenerovanie kostry zdrojového kódu pre cyklus *for*. Tu nestačí len vygenerovať kus kódu. Znakom dobrej použiteľnosti je aj správne umiestnenie kurzora do vytvoreného kódu, čo umožní používateľovi jednoducho modifikovať parametre, napr. názov premennej, cez ktorú sa iteruje a vyplniť telo cyklu. Nemenej dôležitou funkciou je prehľadné a jednoduché zobrazenie nápovede, keď používateľ nemá dostatočné skúsenosti alebo vedomosti o použití odporúčania.

6.1.13 Škálovateľnosť

Typ charakteristiky: kvantitatívna

Škálovateľnosť vyjadruje schopnosť systému pracovať rovnako dobre bez ohľadu na počet používateľov alebo veľkosti spracovávaných dát. To sa vyžaduje najmä pri online aplikáciách, ktoré sú čoraz populárnejšie a využíva ich veľké množstvo používateľov. Odozva odporúčača by mala okamžitá bez ohľadu na to, s koľkými položkami a používateľmi odporúčač v danú chvíľu pracuje. Pri škálovaní sa riešia dva kritické problémy:

- čas potrebný na natrénovanie odporúčača,
- výkonnosť odporúčača so zvyšujúcim sa počtom používateľov a objemom dát v databáze.

Meranie škálovateľnosti systému sa realizuje sledovaním počtu odporúčaní, ktoré je systém schopný navrhnúť za jednotku času (zvyčajne sekundu).

6.1.14 Stabilita

Typ charakteristiky: kvantitatívna

Stabilita predstavuje konzistenciu odporúčaní. Čím častejšie mení odporúčač navrhované položky na podobné dopyty, tým sa považuje za menej stabilný. Pri častej zmene odporúčaní môže dochádzať k zníženiu dôveryhodnosti zo strany používateľov, lebo budú zmätení z toho, že im systém odporúča stále iné položky. Meranie stability sa realizuje porovnávaním položiek, ktoré systém navrhuje na ten istý dopyt počas určitého časového obdobia. Ide o percentuálne vyjadrenie zhody položiek medzi jednotlivými meraniami. Stabilita je opakom robustnosti.

6.1.15 Súkromie

Typ charakteristiky: kvantitatívna, kvalitatívna

Aby sa odporúčania personalizovali, systém si vytvára model používateľa v tvare záznamov (aké akcie používateľ vykonal, ktoré odporúčania akceptoval, ako dlho systém používal a pod.). Zaznamenávanie týchto aktivít mnohým používateľom nevyhovuje. Preto by sa mali údaje uchovávať v takom formáte, aby sa nedali použiť na iné účely. Najbezpečnejšou možnosťou je šifrovanie citlivých dát, čo však spôsobuje výrazné spomalenie výpočtov. K takýmto záznamom napr. patrí zoznam navštívených webových stránok. Webový prehliadač zvyčajne uprednostňuje stránky, ktoré používateľ navštívil v minulosti.

Problém zverejňovaných údajov o používateľoch, ktoré dnešné systémy uchovávajú, je kontroverzná téma [4, 8]. Špeciálne sa tejto problematike venuje Patrick Tucker v publikácii s výstižným názvom Odhalená budúcnosť (angl. Naked Future [8]), kde uvádza množstvo príkladov, ako sa dajú zozbierané dáta využiť bez vedomia používateľov. Dokonca sa podarilo úspešne predpovedať lokalitu výskytu osoby, ktorá o sebe nič nezverejňovala, len na základe zozbieraných aktivít jeho priateľov.

6.1.16 Používateľské preferencie

Typ charakteristiky: kvantitatívna, kvalitatívna

Pri hodnotení odporúčačov je treba brať do úvahy aj preferencie samotných používateľov. Systém by mal umožniť používateľom upraviť nastavenia a rozhranie, napr. spôsob zobrazovania odporúčaní, maximálny počet návrhov či frekvenciu generovania odporúčaní. Každý používateľ je špecifický a má svoje preferencie. Veľký rozdiel je najmä medzi novými používateľmi a skúsenejšími. Skúsení používatelia odmietajú triviálne rady, resp. ich považujú za obťažujúce. Na druhej strane by sa noví používatelia bez nich často nezaobišli. Treba však zvážiť aj samotnú úroveň nastavení, lebo nie všetci sú ochotní pred prvým použitím systému prechádzať rozsiahly formulár s nastaveniami.

6.2 Vzťahy medzi charakteristikami

Niektoré z vymenovaných charakteristík navzájom súvisia. Vzťahy medzi charakteristikami vyjadrujú, ako zmena jednej charakteristiky ovplyvňuje iné charakteristiky. Rozlišujeme tri druhy vzťahov vzájomného ovplyvňovania:

- *kladné (priame) ovplyvnenie* – zlepšením jednej charakteristiky sa zlepší aj druhá (napr. správnosť a dôveryhodnosť),
- *záporné (opačné) ovplyvnenie* – zvýšením jednej charakteristiky sa zníži iná charakteristika (napr. robustnosť a rýchlosť učenia),
- *bez ovplyvnenia* – dvojica charakteristík sa navzájom neovplyvňujú.

Obrázok 6.1 uvádza vzájomné vzťahy charakteristík.

matica	Používateľské preferencie	Správnosť	Pokrytie	Spoľahlivosť	Dôveryhodnosť	Novosť	Priaznivosť	Rozličnosť	Užitočnosť	Rizikovosť	Robustnosť	Súkromie	Použiteľnosť	Stabilita	Škálovateľnosť	Rýchlosť učenia
Používateľské preferencie	X	-	+		+				+				+	+		
Správnosť	-	X	+		+	-	-	-	+	+	-	-			+	+
Pokrytie	+	+	X			+	+	+	+			-			+	
Spoľahlivosť				X												
Dôveryhodnosť	+	+			X					+	+		+	+		
Novosť		-	+			X	+	+		-			+			
Priaznivosť		-	+			+	X	+		-			+			
Rozličnosť		-	+			+	+	X		-						
Užitočnosť	+	+	+						X	+						+
Rizikovosť		+			+	-	-	-	+	X	+	+				
Robustnosť		-			+					+	X			+		-
Súkromie		-	-							+		X				
Použiteľnosť	+				+	+	+						X			
Stabilita	+				+						+			X		
Škálovateľnosť		+	+												X	
Rýchlosť učenia		+							+		-					X

Obrázok 6.1 Matica vzájomného ovplyvňovania charakteristík [6]

6.3 Zhrnutie

V tejto časti sme sa venovali základným charakteristikám a metrikám, ktoré sa využívajú pri vyhodnocovaní a porovnávaní odporúčačov. Uviedli sme šestnásť najfrekventovanejších z nich. Charakteristiky môžu byť kvantitatívne, ktoré sa dajú merať, alebo kvalitatívne. Na výpočet kvalitatívnych sa využívajú rôzne metriky, najčastejšie meranie odchýlky alebo relevantnosti odporúčaných položiek. Jednotlivé charakteristiky odporúčačov sa môžu navzájom ovplyvňovať, teda zvyšovanie hodnoty jednej charakteristiky spôsobuje zmenu (zvyšovanie alebo znižovanie) inej charakteristiky. Pri výbere odporúčača je potrebné zvážiť, ktoré charakteristiky sú podstatnejšie a na základe požiadaviek a preferencií vybrať konkrétny.

Zdroje

[1] Ge, M., Delgado-Battenfeld, C., Jannach, D.: Beyond accuracy: evaluating recommender systems by coverage and serendipity. In: Proceedings of the ACM Conference on Recommender Systems, (2010), pp. 257–260.

[2] Herlocker, J.L., Konstan, J.A.,Terveen, L.G.,Riedl, J.T.: Evaluating collaborative filtering recommender systems. ACM Trans. Inform. Syst. vol. 22 no. 1, (2004), pp. 5–53.

[3] Laclavík, M. a Šeleng, M.: Vyhľadávanie Informácií. Nakladateľstvo STU, Bratislava, (2012). ISBN: 978-80-227-3829-3

[4] Mayer-Schnberger, V., Cukier, K.: Big Data: A Revolution that will Transform how We Live, Work and Think. John Murray Publishers, UK, (2013).

[5] O'Mahony, M., Hurley, N., Kushmerick, N., Silvestre, G.: Collaborative recommendation: a robustness analysis. ACM Trans. Inter. Tech. 4(4), 344–377 (2004).

[6] Robbilard, M.P., Maalej, W., Walker, R.J., Zimmermann, Th.: Recommendation Systems in Software Engineering. Springer-Verlag Berlin Heidelberg, (2014). ISBN: 978-3-642-45134-8.

[7] Smyth, B., McClave, P.: Similarity vs. diversity. Proceedings of the International Conference on Case-Based Reasoning. Lecture Notes in Computer Science, vol. 2080, (2001), pp. 347–361.

[8] Tucker, P.: The Naked Future: What Happens in a World That Anticipates Your Every Move? Penguin Group, New York, (2014). ISBN: 978-1-101-59946-4.

7 Výkonnostné testovanie

V predchádzajúcej kapitole sme sa venovali metrikám a meraniu typických charakteristických vlastností odporúčačov. Spomínali sme vlastnosti ako presnosť odporúčania, pokrytie, rýchlosť učenia, robustnosť, užitočnosť a niekoľko ďalších. V tejto kapitole nadviažeme na porovnávanie a vyhodnocovanie jednotlivých kvalitatívnych charakteristík v súvislosti s meraním celkového výkonu odporúčačov. Pri hľadaní ideálneho odporúčača musíme zohľadniť viacero charakteristík súčasne a taktiež nemôžeme ignorovať špecifické požiadavky vyplývajúce z prostredia, v ktorom bude odporúčač nasadený. Takéto merania odporúčačov sa označujú pojmom výkonnostné testovanie. Pri výkonnostnom testovaní zohľadňujeme nielen kvalitatívne charakteristiky, ale aj ďalšie hľadiská odporúčačov, ktoré vplývajú na ich použitie.

Výkonnostné testovanie je systematický proces porovnávania a merania produktov s cieľom ich neustáleho vylepšovania. Pôvodne sa tento pojem používal v doméne zlepšovania podnikových procesov hľadaním spôsobov, ako efektívnejšie dosiahnuť podnikové ciele. Keďže ide o zložité procesy, je potrebné zvoliť iteratívny postup, pričom sa očakáva, že každá ďalšia iterácia tento proces aspoň trochu zefektívni. Preto sa označuje ako nepretržitý a systematický proces porovnávania a merania produktov, procesov a metód vlastnej organizácie s tými, ktoré boli uznané ako vhodné pre toto meranie, za účelom určiť ciele zlepšenia vlastných aktivít [6].

V poslednej dobe začína byť tento pojem populárny aj vo vedeckej oblasti softvérového inžinierstva, kde sa výkonnostným testovaním označuje proces hľadania optimálneho algoritmu pre konkrétny vedecký problém alebo optimalizáciu životného cyklu vývoja softvéru vrátane jeho automatizovaného testovania [3]. Ide o konštrukčný prístup zvyšovania kvality v oblasti manažmentu a inžinierstva s cieľom nájsť najhodnejšie riešenie pre konkrétny problém [2]. Autori publikácie zameranej na odporúčanie v oblasti softvérového inžinierstva [7] identifikujú výkonnostné testovanie ako metodológiu vhodnú pre zaistenie kvality odporúčačov v softvérovom inžinierstve. Zameriavajú sa prevažne na vyhodnocovanie charakteristík ako presnosť, úplnosť, užitoč-

nosť a niektoré ďalšie [4]. Na základe týchto charakteristík sa dá potom porovnávať a vylepšovať kvalitu odporúčačov. Kvalitatívne charakteristiky však vyjadrujú výkon odporúčača len z hľadiska kvality jednotlivých odporúčaní. Pojem výkon odporúčača zastrešuje aj ďalšie pohľady na výkon, napr. z hľadiska rýchlosti odozvy odporúčača na konkrétnu požiadavku alebo výkon v zmysle počtu predaných produktov, ktoré sa predali vďaka odporúčaniam.

7.1 Odporúčanie z pohľadu zainteresovaných používateľov

Keďže pri vyhodnocovaní odporúčačov treba brať do úvahy účel samotného odporúčača, spomenieme použitie odporúčačov v niektorých typických softvérových systémoch. Pre softvérových inžinierov sú najznámejšie nástroje na odporúčanie, ktoré pomáhajú pri tvorbe samotných softvérových systémov. Ide zvyčajne o poskytovanie odporúčaní v podobe dopĺňania zdrojového kódu, hľadania a opravovania pachov v kóde, návrhu knižníc a funkcií vhodných na použitie v aplikácii a podobne. Používateľmi týchto odporúčačov sú vývojári softvérových systémov. Iné požiadavky sa kladú na odporúčače, v ktorých vystupujú iné skupiny používateľov. Jednu skupinu zvyčajne tvoria zákazníci odporúčačov. Na týchto typoch odporúčačov, ktoré musia zohľadňovať požiadavky rozdielnych používateľských skupín, sa najlepšie prezentuje dôležitosť protichodných požiadaviek, ktoré sú kladené na odporúčače jednotlivými skupinami používateľov.

V oblasti obchodu môžeme kategorizovať používateľov na dve skupiny: poskytovateľov služieb (napr. majitelia firiem) a zákazníkov. Zákazníci sa zaujímajú o samotné produkty, resp. služby, ich cieľom je nájsť čo najužitočnejší produkt alebo službu pre uspokojenie ich potrieb. Predajca má záujem dosiahnuť čo najvyšší zisk, získať nových zákazníkov, dobré meno na trhu a pod. Z hľadiska odporúčania dochádza ku konfliktu, do akej miery zohľadňovať požiadavky jednotlivých skupín. Príkladmi odporúčacích systémov s viacerými používateľskými skupinami sú:

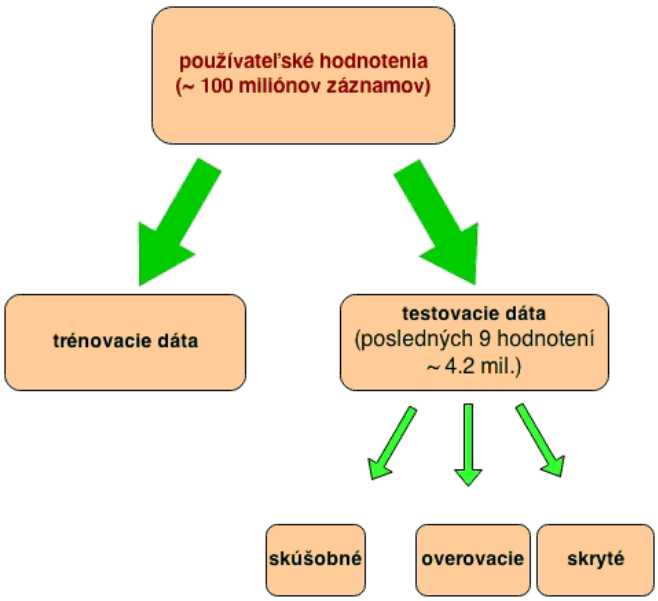
- *Internetové obchody a požičovne* – využívajú odporúčanie pri navrhovaní položiek. Odporúčače by mali navrhovať položky, pri ktorých je veľká pravdepodobnosť, že by mal zákazník záujem o ich kúpu alebo požičanie. Pre majiteľov obchodov a poskytovateľov služieb je zaujímavý najmä počet objednávok z pohľadu zisku, a preto je pre nich výhodnejšie, ak odporúčače odporúčajú viac ziskové položky.
- *Reklamné systémy* – skrývajú zložité odporúčacie algoritmy, ktoré musia brať do úvahy záujmy niekoľkých používateľských skupín súčasne. Do súboja vstupujú záujmy predajcov produktov, poskytovateľov reklamného priestoru, konzumentov reklám a často aj sprostredkovateľov reklamných jednotiek. Preto treba pri odporúčaní zohľadniť nielen relevantnosť reklamy, ktorá sa určuje na základe histórie návštev a nedávnej aktivity používateľa, ale aj ziskovosť reklamy pre predajcu a poskytovateľa reklamnej plochy. Celý tento proces zastrešuje sprostredkovateľ reklamnej kampane, ktorý sa taktiež snaží o maximalizáciu ziskov. Kritický je aj čas výpočtu, nakoľko reklama sa zobrazuje asynchrónne.

- *Spravodajské služby* – poskytujú články na prečítanie. Vo väčšine prípadov sú články dostupné zadarmo, a preto je príjem týchto služieb závislý od množstva čitateľov a ich času stráveného zobrazovaním jednotlivých článkov. Úloha odporúčača obsahu je na prvý pohľad zrejmá – zobrazovať ďalšie články, ktoré by mohli byť pre čitateľa zaujímavé. Tu sa stretávame s protichodnými zámermi používateľských skupín. Čitatelia majú záujem stráviť čítaním čo najmenej času a dozvedieť sa len podstatné informácie, pre poskytovateľov obsahu je zase výhodné, aby si čitatelia zobrazili čo najviac článkov.

Rôzne softvérové systémy vyžadujú rôzny prístup k odporúčaniu. Kvalitou odporúčania sa dlhodobo zaoberajú mnohé veľké spoločnosti, ktoré neustále pracujú na vylepšovaní odporúčacích algoritmov [5]. Vo firme Amazon zistili, že odporúčanie kníh na základe atribútov je oveľa presnejšie a lacnejšie, ako odporúčanie na základe manuálneho kategorizovania odbornými recenzentmi. Snahu o neustále zlepšovanie odporúčačov prejavuje aj firma Netflix[13]. Ide o americkú firmu poskytujúcu požičiavanie médií, ktorá investuje údajne až 150 miliónov dolárov na zlepšenie odporúčania svojich produktov (najmä filmov a televíznych relácii)[14]. Súčasťou tejto snahy bolo organizovanie súťaže[15], v ktorej ponúkali odmenu milión amerických dolárov pre tím, ktorý ako prvý vylepší ich odporúčací algoritmus (nazvaný Cinematch) aspoň o 10 percent. Netflix má k dispozícii milióny záznamov od svojich používateľov, ktorí hodnotia jednotlivé položky (filmy, televízne relácie). Hodnotenie položiek je realizované bodovým hodnotením na stupnici od 1 do 5 bodov. Na základe takto zozbieraných dát sa následne generujú odporúčania ďalším používateľom. V súťaži boli k dispozícii informácie o tom, aké hodnotenie dali používatelia konkrétnym položkám. Zozbierané informácie o hodnotení (približne 100 miliónov záznamov) sa rozdelili na dve množiny (Obrázok 7.1). Trénovacia množina bola zverejnená a súťažiaci ju mohli využívať pri tvorbe algoritmu. Testovacia množina obsahovala približne 4,2 milióna záznamov. V nej sa nachádzalo posledných 9 hodnotení jednotlivých používateľov. Táto sa náhodne rozdelila na 3 rovnaké časti. Prvá tretina sa poskytla súťažiacim na overenie algoritmu po natrénovaní. Na zvyšných dvoch podmnožinách testovacej množiny sa realizovalo vyhodnotenie algoritmu, pričom používatelia sa dozvedeli len strednú kvadratickú odchýlku na druhej testovacej množine. Výsledky odporúčacieho algoritmu na tretej časti testovacích dát si ponechala spoločnosť pre svoje účely, aby nebolo možné na základe výsledkov algoritmus upraviť.

V roku 2009 sa podarilo súťažnému tímu navrhnúť odporúčací algoritmus, ktorý zlepšil odporúčanie. Algoritmus nanešťastie nemohol byť nasadený, lebo nebol dostatočne škálovateľný v produkčnom prostredí. Takže sme sa opäť dostali k problému, že pri hľadaní vhodného odporúčača nie je vždy postačujúce len hľadisko kvality odporúčania, ale treba myslieť aj na ďalšie obmedzenia, akými sú napr. technické možnosti.

[13] Netflix, https://www.netflix.com
[14] https://gigaom.com/2014/10/09/netflix-spends-150-million-on-content-recommendations-every-year/
[15] Súťaž spoločnosti Netflix, http://www.netflixprize.com



Obrázok 7.1: Model znázorňujúci rozdelenie dát poskytnutých spoločnosťou Netflix na vylepšenie odporúčacieho algoritmu (diagram prevzatý z [1]).

7.2 Porovnávanie odporúčačov z rôznych hľadísk

Požiadavky kladené na softvérové systémy rozdeľujeme na funkcionálne a nefunkcionálne. Funkcionálne požiadavky súvisia s kvalitou odporúčaní, čo je hlavná funkcia odporúčača. Nefunkcionálne požiadavky sa týkajú zvyšných oblastí systému, najmä požiadavky na technické parametre (rýchlosť odozvy, výpočtová náročnosť, pamäťová náročnosť) a požiadavky na dosahovanie cieľov podniku vďaka systému (napr. dosiahnutie určitého obratu, počtu zákazníkov, splnenie podnikových cieľov).

Všetky požiadavky na systém sa musia pri porovnávaní a meraní výkonu odporúčača vyhodnocovať súčasne. Keďže sa často stáva, že sú požiadavky protichodné, treba nájsť kompromis. Ďalej sa budeme zameriavať na vyhodnocovanie výkonu odporúčačov z troch základných hľadísk, ktoré navrhujú zohľadniť autori publikácie [9]. Ich model výkonnostného testovania meria kvalitu odporúčačov z *používateľského, technického* a *obchodného* hľadiska. Tieto hľadiská treba zohľadňovať vzájomne, pričom miera a priorita zohľadnenia jednotlivých hľadísk závisí od konkrétneho systému.

7.2.1 Používateľské hľadisko

Používateľom odporúčača rozumieme človeka, ktorému odporúčania zobrazujú. Z pohľadu používateľa je najdôležitejšia kvalita odporúčaných položiek. Kvalita sa dá vyhodnotiť na základe kvalitatívnych charakteristík odporúčačov, ktoré sme podrobne opísali v predchádzajúcej kapitole. Išlo najmä o charakteristiky ako správnosť odporúčania, pokrytie odporúčača, užitočnosť odporúčaných položiek a spoľahlivosť odporúčania. Rovnako sme spomenuli niekoľko metrík, akými sa dajú tieto kvalitatívne charakteristiky odmerať, napr. presnosť, úplnosť alebo F-skóre.

7.2.2 Obchodné hľadisko

Pre majiteľov podnikov a poskytovateľov služieb je hlavným cieľom dosahovať čo najvyšší zisk. Je viacero možností, ako sa pri tom dajú využiť odporúčače. Najčastejšie používaným spôsobom je snaha o zvýšenie predaja navrhovaním relevantných položiek, o ktoré by zákazník mohol mať záujem na základe toho, aké položky si prezeral alebo kúpil v minulosti. Odporúčač môže vďaka nazhromaždeným informáciám lepšie odhadnúť, o čo by mal zákazník záujem. Následne môže navrhovať príslušenstvo k už zakúpeným produktom. Keď si používateľ kúpi tlačiareň, tak mu odporúčač môže pravidelne navrhovať náplň do tejto tlačiarne.

Aby sme vedeli určiť, ako dobre plní odporúčač úlohu z obchodného hľadiska, potrebujeme nejakým spôsobom odmerať, či odporúčané položky reálne vzbudzujú záujem zo strany zákazníka a zvyšujú predajnosť, čiže zisky podniku. Proces nákupu odporúčanej položky končí z pohľadu systému finálnym odoslaním objednávky, resp. z pohľadu predaja zaplatením a prevzatím produktu. V rámci interakcie zákazníka so systémom sa skladá proces objednávky z viacerých krokov. Obrázok 7.2 znázorňuje príklad tohto procesu. V prvom kroku si zákazník kliknutím na navrhovanú položku odporúčania zobrazí náhľad odporúčaného tovaru. V druhom kroku má možnosť vložiť tovar do košíka a v treťom kroku prejde k vytvoreniu a odoslaniu objednávky. V ľubovoľnom kroku však môže prerušiť svoj zámer uskutočniť obchod. Rovnako sa môže stať, že bude zákazník pred odoslaním objednávky ešte chvíľu váhať a medzičasom prezerať iný tovar v obchode. Preto sa zvykne úspešnosť odporúčania merať nielen mierou prekliknutia medzi jednotlivými krokmi, ale aj počtom vytvorených objednávok. Poznáme teda metriky:

- *miera prekliknutia* (angl. click through rate, CTR) – meria sa ako pomer počtu kliknutí na odporúčanú položku k počtu všetkých zobrazení daného odporúčania (túto mieru je možné sledovať pre každý preklik medzi krokmi procesu samostatne),
- *prepočítavací koeficient* (angl. conversion rate) – meria sa ako pomer zobrazení položky medzi odporúčanými položkami k počtu uskutočnených objednávok.

Obrázok 7.2: Proces objednávky – na objednanie tovaru treba vykonať tri prekliky.

7.2.3 Technické hľadisko

Pri výbere odporúčača treba zohľadniť aj technické požiadavky, ktoré vyplývajú najmä z prostredia, v ktorom bude odporúčač nasadený. Aj keby sa nám podarilo vytvoriť najpresnejší odporúčač na svete, bez zohľadnenia hardvérových a softvérových požiadaviek nemusí byť v praxi použiteľný (ako sme spomenuli aj v prípade víťazného algoritmu odporúčania pre spoločnosť Netflix). Tieto požiadavky zvyčajne charakterizujeme ako obmedzenia, najmä:

- obmedzenia kladené na dáta,
- systémové obmedzenia.

Odporúčače zvyčajne pracujú s veľkým objemom dát, a preto treba zohľadniť spôsob, ako sa budú dáta uchovávať a ako bude prebiehať ich prenos. Najmä populárne bezdrôtové zariadenia spôsobujú problémy pri potrebe presúvania veľkého objemu dát. Požiadavky na hardvér a softvér taktiež výrazne obmedzujú možnosti odporúčačov. Nestačí riešiť len výpočtový výkon a pamäť, ktorej máme v dnešnej dobe väčšinou dosť, ale aj samotný návrh rozhrania odporúčača, keďže sú čoraz populárnejšie zariadenia s menšími veľkosťami displeja (mobily, tablety a pod.). Základné charakteristiky kladené na odporúčače z technického pohľadu sú:

- rýchlosť odozvy,
- škálovateľnosť,
- rýchlosť učenia,
- robustnosť.

Rýchlosť odozvy vyjadruje čas, za ktorý systém reaguje na požiadavku alebo akciu používateľa. Prijateľná doba odozvy sa uvádza v rozmedzí od 10 do 1000ms. Meranie tejto charakteristiky sa robí spriemerovaním viacerých požiadaviek a označuje sa ako priemerná rýchlosť odozvy.

Škálovateľnosť je schopnosť odporúčača fungovať rovnako stabilne bez ohľadu na veľkosť dát a počet používateľov. Najmä v online aplikáciách je škálovateľnosť dôležitá, keďže vyťaženosť systému nie je rovnomerná. Závisí nielen od jednotlivých dní, ale aj času. Cez deň je systém zvyčajne vyťažený viac, ako v nočných hodinách. Rovnako môžeme sledovať vysokú záťaž serverov napr. pred sviatkami. Meranie škálovateľnosti prebieha tak, že sa za určitý čas odošle veľké množstvo požiadaviek a spočíta sa, koľko z nich zvládne systém spracovať.

Robustnosť a *rýchlosť učenia* sú kvalitatívne a navzájom opačné charakteristiky, ktoré sme spomenuli v predchádzajúcej kapitole. Robustnosť vyjadruje odolnosť systému voči falošným akciám, teda do akej miery je schopný systém odporúčať stabilne. Rýchlosť učenia zase vyhodnocuje, ako rýchlo je systém schopný prispôsobiť odporúčanie pri zmene používateľských preferencií. Meria sa sledovaním času, kedy bude systém opätovne odporúčať správne výsledky, alebo kedy systém dosiahne určitú úroveň správnosti po zmene preferencií.

7.3 Príklad na výkonnostné porovnávanie

V tejto časti uvedieme vzorový príklad, akým sa dá vykonávať výkonnostné testovanie odporúčačov, keď potrebujeme brať do úvahy viaceré hľadiská. Ide o jednoduchý príklad prevzatý z [7], ktorého podstatou je vysvetliť postup porovnávania.

Keď potrebujeme zohľadniť viaceré kritériá súčasne, najlepším spôsobom je využiť váhovanie jednotlivých kritérií – každému kritériu pridelíme určitú váhu. Ak sú pre nás všetky kritériá rovnako dôležité, váhu rozdelíme rovnomerne. Celkovú výkonnosť (označenie U) systému s označením *f* vyjadríme vzťahom:

$$U(f) = \sum_i w_i E_i(f) \tag{7.1}$$

kde *w* je stĺpcový vektor jednotlivých váh (váha je číslo väčšie alebo rovné nule) a *E* vyjadruje odmeranú hodnotu tohto hľadiska. V príklade uvažujeme s tromi odporúčačmi, ktoré využívajú rôzne algoritmy odporúčania:

- k-najbližších susedov (označenie kNN),
- k-najvzdialenejších susedov (označenie kFN),
- náhodný odporúčač (označenie Rnd).

Algoritmus kNN je najrozšírenejší algoritmus, ktorý odporúča na základe podobnosti položiek a používateľských preferencií. Používateľovi odporúča také položky, ktoré vysoko hodnotili iní používatelia s podobnými preferenciami. Nevýhodou algoritmu je, že sú odporúčania málo rozmanité. Položky s väčším množstvom kladných odporúčaní zatienia položky, ktoré nemajú žiadne hodnotenie a často sa stáva, že sa do zoznamu odporúčaní dostáva len obmedzená množina najpopulárnejších položiek.

Algoritmus kFN funguje na opačnom princípe ako algoritmus kNN. Používateľom odporúča položky, ktoré majú záporné hodnotenie od používateľov, ktorí sa správajú opačne. Výhodou algoritmu je, že sa medzi odporúčané položky dostanú aj tie, ktoré nikto doteraz kladne nehodnotil, čím sa dosiahne vyššia rozmanitosť odporúčania.

Náhodný odporúčač vyberá položky náhodne. Jeho výhodou je rýchlosť výberu pri rozmanitosti položiek, nakoľko netreba náročné výpočty. V praxi sa však používa len v prípadoch, keď ešte nie sú dostupné informácie pre personalizáciu odporúčania, napr. pri nových používateľoch.

Vstupné údaje sú zo vzorky 132 používateľov z prostredia odporúčania filmov [8]. Číselné hodnoty obchodného a používateľského hľadiska boli získané cez spätnoväzobný dotazník, kde používatelia mohli vyjadriť svoju spokojnosť s odporúčanými návrhmi, a či by v budúcnosti systém opätovne používali. Z technického hľadiska sa zohľadňoval čas, za ktorý systém zobrazil odporúčané položky. Tabuľka 7.1 uváza namerané a vypočítané hodnoty odporúčačov. Maximálna (najlepšia) možná hodnota je 1 (sto percent). Stĺpce tabuľky predstavujú odporúčacie algoritmy. Prvé tri riadky tabuľky sú hodnoty jednotlivých hľadísk a spodná časť tabuľky obsahuje výsledné hodnotenie odporúčačov na základe váhovania jednotlivých hľadísk. Všetky tri odpo-

rúčače vyhodnocujeme podľa troch spomínaných hľadísk, ktorých kombinácia je vyjadrená vzťahom 7.2:

- *hľadisko používateľa* $E_u(f)$ – správnosť odporúčaných položiek,
- *obchodné hľadisko* $E_b(f)$ – priemerná návratnosť používateľa,
- *technické hľadisko* $E_t(f)$ – priemerná časová náročnosť.

$$U(f) = w^T E(f) = \sum_i w_i E_i(f) = w_b E_b(f) + w_u E_u(f) + w_t E_t(f) \qquad (7.2)$$

Tabuľka 7.1: výkonnostné meranie jednotlivých odporúčačov

	kNN	**kFN**	**Rnd**
$E_u(f)$	0,53	0,52	0,44
$E_b(f)$	0,56	0,51	0,36
$E_t(f)$	0,80	0,80	0,99
U(f), w = (0,3; 0,3; 0,3)	0,63	0,61	0,60
U(f), w = (0,6; 0,3; 0,1)	0,57	0,55	0,47
U(f), w = (0,3; 0,6; 0,1)	0,58	0,54	0,43
U(f), w = (0,1; 0,1; 0,8)	0,75	0,74	0,87

Vzorový výpočet U(f) pre druhý riadok spodnej časti tabuľky (vektor váh: w = (0,6; 0,3; 0,1)) pre odporúčač kNN.

$$\begin{aligned} U(f) &= w_b E_b(f) + w_u E_u(f) + w_t E_t(f) \\ &= 0{,}6 * 0{,}53 + 0{,}3 * 0{,}56 + 0{,}1 * 0{,}8 \\ &= 0{,}566 \approx 0{,}57 \end{aligned}$$

Ako je vidno z vypočítaných hodnôt, pri výbere algoritmu musíme zvážiť váhy jednotlivých hľadísk. Keď sú všetky hľadiská rovnako dôležité (prvý riadok spodnej časti tabuľky, kde je každému hľadisku priradená váha 0,3), tak je najvýhodnejšie zvoliť algoritmus kNN. Tento algoritmus dosahuje najlepšie výsledky z používateľského aj obchodného hľadiska a vo výslednom porovnaní s algoritmami kFN a Rnd je lepší. Výrazne slabší je len v porovnaní s algoritmom Rnd pri zohľadnení technického hľadiska. Algoritmus Rnd sa preto oplatí použiť vtedy, keď je pre nás kritický najmä výpočtový čas (posledný riadok tabuľky, kde je technickému hľadisku priradená váha 0,8). Algoritmus kFN nedosahuje nikdy lepšie výsledky ako ostatné algoritmy.

7.4 Zhrnutie

V tejto časti sme si predstavili výkonnostné testovanie ako proces porovnávania a merania odporúčacích systémov. Pri porovnávaní viacerých odporúčačov súčasne je treba zohľadniť viaceré hľadiská. Zamerali sme sa na tri základné hľadiská. Najdôležitejšie je hľadisko používateľa, aby odporúčač navrhoval relevantné a správne položky. Taktiež však nesmieme zabúdať na technické hľadisko, kde berieme do úvahy najmä požiadavky na systém a v neposlednom rade treba myslieť aj na účel samotného odporúčania z hľadiska podniku. Na záver sme uviedli názorný

príklad, ako sa dajú jednotlivé odporúčače merať a navzájom porovnávať. Na základe výsledkov porovnania dokážeme lepšie vybrať ten najvhodnejší odporúčač pre konkrétny účel.

Zdroje

[1] Bell, R., Koren, Y., Volinsky, C.: Chasing $1,000,000: How we won the Netflix Progress Prize. ASA Statistics Computing and Graphics Newsletter vol. 18, issue 2, (2007). pp. 4–12.

[2] Boxwell, R.J.: Benchmarking for Competitive Advantage. McGraw-Hill Professional Publishing, New York, (1994).

[3] Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, (2012). pp. 178–188.

[4] Herlocker, J.L., Konstan, J.A., Terveen, L.G., Riedl, J.T.: Evaluating collaborative filtering recommender systems. Transactions on Information Systems vol. 22, no. 1, (2004). pp. 5–53.

[5] Mayer-Schnberger, V., Cukier, K.: Big Data: A Revolution that will Transform how We Live, Work and Think. John Murray Publishers, UK. (2013).

[6] Nenadál, J.: Měření v systémech managementu jakosti, Praha: Management Press, 2004. ISBN 80-7261-110-0.

[7] Robbilard, M.P., Maalej, W., Walker, R.J., Zimmermann, Th.: Recommendation Systems in Software Engineering. Springer-Verlag Berlin Heidelberg, 2014. ISBN: 978-3-642-45134-8.

[8] Said, A., Fields, B., Jain, B.J., Albayrak,S.: User-centric evaluation of a K-furthest neighbor collaborative filtering recommender algorithm. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, San Antonio, TX, USA, (2013). pp. 1399–1408.

[9] Said, A., Tikk, D., Shi, Y., Larson, M., Stumpf, K., Cremonesi, P.: Recommender systems evaluation: A 3D benchmark. In: Proceedings of the Workshop on Recommendation Utility Evaluation: Beyond RMSE, CEUR Workshop Proceedings, vol. 910, Dublin, Ireland, (2012). pp. 21–23.

8 Oblastné štúdie

Jedným so spôsobov, ako zrealizovať a vyhodnotiť efektívnosť odporúčacích systémov v softvérovom inžinierstve, je vykonať oblastnú štúdiu. Oblastné štúdie sú dôležitým rozšírením laboratórnych experimentov do reálnych situácií, ktoré prebiehajú v softvérových spoločnostiach, alebo všeobecne v spoločnosti. Prinášajú do problematiky veľký nadhľad, nové nápady, vylepšenia a rozvinutie. Na druhej strane, s vykonávaním oblastnej štúdie je spojených veľa výziev, ako sú napr. implementácia výskumného návrhu, zabezpečenie dostatočnej kontroly, spoľahlivosti a platnosti. Dôležitou výzvou v praxi je aj nájdenie vhodných organizácií, ktoré sa budú študovať.

Oblastná štúdia sa definuje ako štúdia, ktorá sa vykonáva najmä v prirodzenom prostredí predmetu štúdie, skôr ako v laboratórnom prostredí. Zahŕňa výskum, experimenty a interakciu s účastníkmi (používateľmi). Oblastné štúdie sa využívajú najmä v sociálnych vedách, keďže sa v nich skúma správanie a činnosti ľudí. Zameriavajú sa na pochopenie problémov, ktoré sa vyskytujú v praxi a na ich riešenia, ktoré budú implementované do činnosti účastníkov.

Softvérové inžinierstvo je náročný obor, ktorý je bohatý na dáta v zmysle artefaktov a dennodenných rozhodnutí. Preto už dnes existuje mnoho odporúčacích systémov, ktoré asistujú softvérovým profesionálom pri rôznych aktivitách.

Odporúčacie systémy v softvérových organizáciách sú vytvorené k podpore rozhodnutia v neistých situáciách a k uľahčeniu dodania úlohy alebo procesu. Výstup odporúčacieho systému používa celý vývojový tím na prekonanie technologických výziev, ako sú organizovanie technických zdrojov, vylepšenie kvality práce a nájdenie a vyriešenie problémov v procese alebo v kóde priamo. Odporúčacie systémy sú svojimi vlastnosťami bohaté na históriu používania, tým pádom sa dajú na ne aplikovať rôzne techniky analýzy dát. V tomto zmysle sú odporúčacie systémy v softvérovom inžinierstve veľmi vhodné na vytvorenie oblastnej štúdie. Vedenie oblastnej štúdie je prospešné pre výskumníkov aj praktických softvérových vývojárov. Výskumníci sú schopní lepšie pochopiť hlavné výzvy pre praktických softvérových vývojárov počas vývoja softvéru,

zbierať reálne dáta a navrhovať jednoduché riešenia. Praktickí softvéroví vývojári na druhej strane môžu ťažiť z výstupov oblastnej štúdie, ako sú odporúčacie systémy a repozitáre dát.

V tejto práci opíšeme kroky, ktoré sú potrebné spraviť k vedeniu oblastnej štúdie. Oboznámime sa aj s hlavnými výzvami, ktoré sú spojené s vedením oblastnej štúdie.

8.1 Návrh výskumu

Proces výskumu zahrňuje rozhodnutia, aké sa použijú metodológie a metódy. Uvedieme jeden rámec, ktorý kombinuje metódy, metodológie a epistemológie [3]. Metóda sa definuje ako technika alebo procedúra, používaná na získanie a analýzu dát na základe výskumníckych otázok a hypotéz. Metodológia sa definuje ako stratégia alebo návrh pre výber a použitie príslušných metód, aby sme získali požadované výstupy. Teoretická perspektíva je filozofický postoj formujúci metodológie, čiže vytvára kontext pre procesy a určuje ich logiku a kritériá. Epistemológia je teória poznania, ktorá vytvára základ celého výskumného procesu a určuje príslušne zvolenú teoretickú perspektívu. Pre zhrnutie, teoretická perspektíva implikuje výskumnícke otázky a ovplyvňuje metodológiu, ktorá je potom nasledovaná použitím metód, ktoré sú konzistentné s metodológiou.

Niektorí výskumníci spomínajú epistemológiu inak. Niektorí ju nazývajú svetonázor, čiže základný súbor presvedčení, ktorými sa riadi činnosť [2]. Svetonázor bude viesť k zvoleniu kvalitatívneho, kvantitatívneho alebo zmiešaného prístupu vo výskume. Pred hocijakým empirickým výskumom by sa mal výskumník rozhodnúť, ktorý svetonázor zvoliť na základe povahy záujmu výskumníckej otázky. Existujú štyri typy svetonázorov [2]: post-pozitivistický, konštruktivistický, participatívny a pragmatický. Tabuľka 8.1 uvádza rozpis týchto svetonázorov.

Tabuľka 8.1 Výskumné svetonázory

Post-pozitivizmus	**Konštruktivizmus**
Determinizmus	Pochopenie
Redukcionizmus	Význam od viacerých účastníkov
Empirické pozorovania a merania	Sociálne a historické konštrukcie
Overenie teórie	Vytvorenie teórie
Participatívnosť	**Pragmatizmus**
Politický	Dôsledky akcie
Orientovaný na riešenie problémov	Zameraný na problém
Kolaboratívny	Pluralitný
Orientovaný na zmenu	Orientovaný na prax v reálnom svete

8.1.1 Výskumné metodológie

Existujú tri typy výskumného návrhu:

- kvalitatívny,
- kvantitatívny,
- zmiešaný.

Kvalitatívny prístup skúma do hĺbky nejaký jav alebo ľudské správanie. Vo výskumnom procese je objektom účastník. Dôraz sa kladie na platnosť výskumu. Prípadové štúdie, naratívne prístupy, štúdia zakotvenej teórie a fenomenológia sú výskumné metodológie, ktoré nasledujú kvalitatívny prístup. V softvérovom inžinierstve sú prípadové štúdie populárne používané oproti iným z toho pohľadu, že skúma súčasný fenomén v podmienkach reálneho sveta.

Kvantitatívny prístup zahŕňa testovanie teórií alebo skúmanie kauzálnych vzťahov medzi premennými. V kvantitatívnom prístupe je výskum nastavený presne, výskumnými metodológiami sú experimenty a prieskumy, výskumník je teda mimo dejiska. Dôraz sa kladie na spoľahlivosť a zovšeobecňovanie výsledkov.

Zmiešaný návrh zahrňuje elementy kvalitatívneho a kvantitatívneho prístupu, aby prekonal limity jednotlivých metodológií. Výskumníci môžu študovať účastníkov alebo výskumnú otázku jednou metódou (napr. všeobecný prieskum) a použiť tieto znalosti na opýtanie sa správnych otázok inou metódou (napr. prípadová štúdia). Kvalitatívne a kvantitatívne metódy získavania dát môžu viesť k spojeniu do jednej databázy, alebo analýza kvantitatívnych dát môže byť podporená kvalitatívnymi zisteniami.

8.1.2 Výskumné metódy

Výskumné metódy zahŕňajú získavanie dát, analýzu, interpretáciu a prezentáciu výsledkov. V závislosti od metodológie budú aj metódy rozdielne:

- *Kvalitatívne metódy* – založené na rozhovoroch, prieskumoch s otvorenou (ľubovoľnou) odpoveďou a pozorovaniach na nájdenie tém a vzorov. Dáta sú textového typu.
- *Kvantitatívne metódy* – založené na štatistických a matematických modeloch, ktoré využívajú výpočtovú techniku. Používajú sa v nich otázky s jednoznačnou odpoveďou (áno/nie). Dáta sú numerického typu.

8.2 Vedenie oblastného výskumu

Oblastná štúdia v odporúčacích systémoch v softvérovom inžinierstve má typicky štyri kroky:

1. plánovanie a vyjednávanie,
2. zbieranie dát a analýza,
3. vytváranie odporúčacieho systému a kalibrácia,
4. rozvinutie odporúčacieho systému.

8.2.1 Plánovanie a vyjednávanie

Ako sme už spomínali, oblastné štúdie sú prospešné v kontexte odporúčacích systémov, keďže cieľ je navrhnúť praktické riešenia k reálnym výzvam v softvérovej organizácii a napokon aj rozvinúť toto riešenie, ako nástroj v existujúcom systéme. Aby sa tento účel dosiahol, začiatočné kroky oblastnej štúdie identifikujú obchodné ciele, dlhodobé a krátkodobé stratégie a výzvy v softvérovej organizácii.

Počas tohto kroku, priama interakcia so softvérovými praktikantmi je veľmi dôležitá. Interakcie pomocou formálnych a neformálnych rozhovorov a sedení, pozorovanie vývojového prostredia, ktoré ukáže vzory aktivít a cieľov, sú dôvodom k pochopeniu potrieb praktikantov.

V jednej oblastnej štúdii [6] vo veľkej softvérovo vývojovej organizácii bol vytvorený odporúčací systém, ktorý priraďoval ľudí v priebehu veľkých softvérovo-vývojových úloh s prihliadnutí na ich expertízu. Na základe začiatočných rozhovorov s vývojovými skupinami výskumníci zistili, že existujúce techniky pre hľadanie expertov sú vysoko neisté a časovo neefektívne, a že majú tendenciu zbytočne zaťažovať určité individuality (v tomto prípade softvérových architektov). Takže po začiatočných rozhovoroch, ciele tohto odporúčacieho systému (nazvaného Expertise browser) boli stanovené na priradenie expertov na úlohy v rýchlom čase, bez zaťažovania opakujúcich sa expertov a vedeniu používateľov k hľadaniu alternatív, keď niektorí experti nebudú k dispozícií.

Po identifikovaní problému by mali byť vyjednávania s praktikantmi vedené v zmysle aké vstupy potrebuje odporúčací systém, aké majú byť funkcie systému, výstupy dodávané ako odporúčania a detailný plán, koľko času budeme tráviť získavaním dát, analýzou a rozvíjaním samotného odporúčacieho systému. Oblastné štúdie potrebujú častú komunikáciu medzi výskumníkmi a praktikantmi v každom kroku, najmä kvôli zamedzeniu sklamania z finálneho výstupu, preto treba plánovanie stretnutí brať vážne.

8.2.2 Zbieranie dát a analýza

Existujú rôzne stratégie zbierania dát. Záleží to na zvolenom výskumnom prístupe (kvalitatívny alebo kvantitatívny) a výskumných otázkach. Podľa množstva kontaktu potrebného medzi výskumníkmi a praktikantmi existujú tri stratégie získavania dát [8]:

- priama,
- nepriama,
- nezávislá.

Priama technika vyžaduje najväčšie množstvo interakcie s praktikantami. Expertne sústredené skupiny, rozhovory a dotazníky sú tri z mnohých príkladov techník získavania dát. Všetky z nich sú bežne používané počas konštrukcie a vyhodnocovania odporúčacieho systému. Expertne zamerané skupiny sú veľmi podobné tvorivej technike nových myšlienok (*angl.* brainstorming), ale v tomto prípade sa skôr zameriava na konkrétny problém, ako na generovanie myšlienok koľko je možných. Rozhovory sú naopak viac riadené konverzácie medzi výskumníkom a respondentom. Rozhovory vedia poskytnúť požadované informácie, ak sa výskumník drží otázok s otvorenou odpoveďou. Dotazníky sú podávané na základe starostlivo spísaných otázok.

Nepriame techniky pomáhajú výskumníkom získavať dáta cez implementované systémy, ktoré vývojový tím aktívne používa. Tieto techniky vyžadujú veľmi málo času od praktikantov a tým pádom sú vhodné na dlhodobé štúdie.

Existujú aj nezávislé techniky, ktoré zhromažďujú výstupy a vedľajšie produkty vývoja bez hocijakej účasti softvérových praktikantov. Príkladmi takýchto výstupov sú zdrojové kódy a dokumentácie. Príkladmi vedľajších produktov sú pracovné žiadosti a záznamy o zmene. Dáta

zozbierané nezávislými technikami môžu byť potom transformované ako vstupy odporúčacieho systému. Odporúčacie systémy v softvérovom inžinierstve sa stali populárnejšími s použitím verejne dostupných dát, ako zdrojový kód a repozitáre získané nepriamymi a nezávislými technikami [7].

Nakoniec, odporúča sa držať zmiešaného prístupu získavania dát pri konštrukcii a kalibrácii odporúčacieho systému. Príkladom je odporúčací systém Sibyl, ktorý bol vytvorený v oblastnej štúdii na redukciu času a úsilia potrebného na vyhodnotenie chybného hlásenia, ktorý komponent ho spôsobil a aj ktorému vývojárovi priradiť toto hlásenie. Iniciálny prototyp Sibylu bol zhotovený na základe dát nazbieraných vo webovom rozhraní pre Bugzillu a presnosť a užitočnosť odporúčaní bola overená na základe dotazníkov, ktoré boli dodané štyrom tímom vývojárov.

Techniky analýzy dát sú rozdelené na základe typu dát na kvalitatívne alebo kvantitatívne:

- Kvalitatívna analýza dát sa používa na prieskumné či potvrdzujúce štúdie a vizualizáciu.
- Kvantitatívna analýza dát sa používa na opis, porovnávanie alebo predikciu [8].

V oblastných štúdiách výber techniky analýzy dát závisí od výskumných otázok a metodológie zbierania dát. Ak je priama technika používaná vo forme rozhovorov, dotazníkov a pod., tak je potrebná dôkladná analýza a často vyžaduje viac času a úsilia ako je očakávané (napr. transformovanie všetkých odpovedí z prieskumu na merateľné jednotky). Ak boli použité nepriame a nezávislé techniky, tak sa využíva kvantitatívna analýza, ako napr. štatistické testovanie hypotéz (s ohľadom na predpoklady o pravdepodobnostnom rozdelení dát) a algoritmy čistenia dát (detekcia alebo odstraňovanie šumu).

Výsledky z analýzy dát sa odporúča prediskutovať s praktikantmi, ktorí môžu naznačiť, či sú výsledky z analýzy presné, alebo upozorniť na potenciálne odľahlé pozorovania v dátach.

8.2.3 Vytvorenie OSSI a kalibrácia

Pred vytvorením odporúčacieho systému by výskumníci mali dôkladne pracovať na návrhu vstupov, výstupov a samotného jadra systému, ktorý bude počítať odporúčania. Jadro odporúčacieho systému je reálna znalosť (produkt) poskytnutá výskumníkmi počas tretej fázy vedenia oblastnej štúdie. Táto časť sa hlavne skladá zo zhromažďovania dát a analýzy, implementácie predikčného modelu a výberu algoritmu usporiadania, ktorý systematicky zaradí odporúčania od najviac hodnotnú po najmenej. Jadro odporúčacieho systému môže byť ohodnotené niekoľkými spôsobmi:

- offline,
- používateľská štúdia,
- online.

Offline metóda ohodnotenia sa robí bez interakcie používateľov. Používajú sa v nej existujúce dátové sady z verejných dátových repozitárov, open source systémov alebo použitím vopred zhromaždených dátových sád od softvérovej organizácie. Presnosť systému je začiatočne overená počas offline experimentov. Offline experimenty sú užitočné na testovanie kandidátskej mno-

žiny algoritmov s nízkymi nákladmi oproti iným typom experimentov, napr. na overenie odporúčacieho systému použitím repozitárov dát.

Používateľská štúdia je vykonávaná výberom menšej skupiny subjektov, ktoré budú používať systém alebo pilotný projekt v softvérovej organizácie. Pokým užívatelia plnia svoje úlohy, výskumníci zaznamenávajú ich správanie, zbierajú štatistiky a získavajú spätnú väzbu po používaní. Príkladom je odporúčač zvaný APIExplorer, ktorý bol ohodnotení pomocou hĺbkovej štúdie so 32 sedeniami s účastníkmi. APIExplorer bol vyvinutý ako funkcia v Eclipse IDE, ktorá objavovala a odporúčala metódy alebo typy v API [4].

Online vyhodnotenie je vykonané, keď je systém používaný reálnymi používateľmi, ktorí produkujú reálne úlohy. Je najvierohodnejšie porovnať rôzne algoritmy usporiadania online. Online ohodnotenie je ale možné vykonať len po rozvinutí odporúčacieho systému.

8.2.4 Rozvinutie odporúčacieho systému

Rozvinutie je náročný proces a potrebuje „vieru“ vo vzniknutý odporúčací systém. Hlavné body procesu rozvinutia sa dajú zosumarizovať takto.

Integrácia s existujúcim systémom. Na základe návrhu odporúčacieho systému môže operovať nezávisle ako samostatná aplikácia alebo plugin. Pred rozvinutím odporúčacieho systému by sa malo objasniť a prediskutovať s praktikantmi, ako bude odporúčací systém komunikovať s inými systémami. Je to z toho dôvodu, že odporúčací systém sa môže, alebo je nutné rozšíriť na väčší systém, a tým pádom treba vytvoriť nový návrh. Príkladom je odporúčací systém Mylar, ktorý bol navrhnutý na sledovanie programátorovej aktivity pomocou vývojového prostredia [5]. Rozvinutá verzia Mylaru, označovaná ako Mylyn, už zvláda komunikáciu a interakciu medzi existujúcimi systémami, historickými úložiskami a Mylynom.

Kalibrácia po začatí používania. Príkladom takéhoto postupu je systém DebugAdvisor [1], ktorý vie byť kalibrovaný na základe používateľskej kvalitatívnej spätnej väzby. Autori nástroja nechali používateľov označovať odporúčania buď za užitočné alebo nie. Systém sa potom automaticky sám vylepšoval pre budúce odporúčania.

8.3 Výzvy spojené s vedením oblastnej štúdie

Tabuľka 8.2 uvádza hlavné výzvy spojené s vedením oblastnej štúdie, na ktoré sa treba sústrediť.

Tabuľka 8.2 Výzvy pri vedení oblastnej štúdie

Organizačné	Dátové	Návrhové
Plánovanie stretnutí Agenda stretnutia Výber účastníkov Zbieranie dát z dotazníkov Málo dát Chýbajúce dáta Veľkosť organizačného tímu- Organizačná kultúra	Bezpečnosť a ochrana súkromia Získavanie dát Techniky zvládajúce chýbajúce dáta	Škálovanie Zásady ochrany osobných údajov Použiteľnosť OSSI Správanie sa OSSI pri zašumených dátach

8.4 Zhrnutie

V tejto práci sme sa venovali oblastným štúdiám v kontexte odporúčacích systémov v softvérovom inžinierstve. Naším cieľom bolo ukázať, ako viesť krok za krokom oblastnú štúdiu na vytvorenie odporúčacích systémov.

Na začiatku sme predstavili aké metódy, metodológie a epistemológie sa využívajú na výskum všeobecne. Tento prehľad a poznatky by nám mali dať určitú perspektívu a konzistenciu v oblastnej štúdii. Ďalej sme opísali, aké kroky zahŕňajú vedenie oblastnej štúdie. Nakoniec sme zvýraznili výzvy tohto procesu.

Oblastné štúdie sú jediným spôsobom, ako výskumníci môžu pochopiť oblasti svojho záujmu a navrhnúť technické riešenia pre reálne potreby praktikantov. Bolo zistené, že technické výzvy sa ľahko prekonávajú, za to sociálne, organizačné a poznávacie výzvy sú tie, ktoré robia prijatie a použitie odporúčacie systému ťažším. Preto je potrebné viesť viac oblastných štúdií na prekonávanie týchto problémov a navrhnúť ich riešenia. Kombinácia softvérového inžinierstva, strojového učenia a dolovania dát vie pomôcť prístupu k veľkým dátam a hĺbkovej analýze. Preto by odporúčacie systémy nemali byť vytvorené obmedzeným počtom offline experimentov. Potrebujeme pochopiť základné pojmy a skúmať ich s dostupnými dátami a modelmi. V tomto zmysle sú oblastné štúdie jediným spôsobom k prekonaniu limitov a predpokladov.

Literatúra

[1] Ashok, B., Joy, J., Liang, H., Rajamani, S.K., Srinivasa, G., Vangala, V.: DebugAdvisor: a recommender system for debugging. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 373–382 (2009). doi:10.1145/1595696.1595766

[2] Creswell, J.W.: Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 3rd edn. Sage, Beverly Hills (2009).

[3] Crotty, M.J.: The Foundations of Social Research: Meaning and Perspective in the Research Process. Sage, Beverly Hills (1998).

[4] Duala-Ekoko, E., Robillard, M.P.: Using structure-based recommendations to facilitate discoverability in APIs. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 79–104 (2011). doi:10.1007/978-3-642-22655-7_5

[5] Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11 (2006). doi:10.1145/1181775.1181777

[6] Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 503–512 (2002). doi:10.1145/581339.581401

[7] Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software 27(4), 80–86 (2010). doi:10.1109/MS.2009.161

[8] Shull, F., Singer, J., Sjoberg, D.I.K. (eds.): Guide to Advanced Empirical Software Engineering. Springer, Berlin (2008). doi:10.1007/978-1-84800-044-5

9 Odporúčacie systémy orientované na znovupoužitie kódu

Počas vývoja softvéru býva snahou vývojárov minimalizovať opakovanie písania rovnakého zdrojového kódu. Vďaka analýze už predtým vytvorených artefaktov zdrojového kódu nám môže odporúčací systém navrhnúť znovupoužitie celých knižníc, prípadne iba jej niektorých častí. S využitím existujúcich častí zdrojového kódu je možný rýchlejší vývoj softvéru. V tejto kapitole sú prezentované princípy a základy odporúčacieho systému orientovaného na znovupoužitie kódu. Zároveň sa predstaví aj jeden z možných architektonických návrhov pre vytvorenie odporúčacieho systému.

Určite každý vývojár softvéru si pri svojej práci aspoň raz uvedomil, že aktuálne vytváranú časť zdrojového kódu už predtým niekde vytvoril, a preto ak by ju našiel a opäť použil, tak by sa vyhol známemu výroku „znova vynájdeniu kolesa“ [1]. Aby vývojári vyhľadali užitočné časti zdrojových kódov, musia prerušiť aktuálnu činnosť (teda vyvíjať) a investovať úsilie a čas do vyhľadávania. Výsledok ich hľadania nemusí byť vždy prínosný, niektoré nájdené časti nemusia vyhovujú podmienkam či potrebám a pre použitie iných je zas potrebné prispôsobiť zdrojový kód. Dokonca je možné, že niekedy sa vhodné časti ani nenájdu.

Ak chceme navrhnúť efektívny systém umožňujúci znovupoužitie existujúceho častí kódu, musíme vyriešiť tri hlavné problémy:

- *problém repozitára* – kde nájsť dostatočné množstvo znovupoužiteľného materiálu,
- *problém reprezentácie* – ako uchovať a reprezentovať znovupoužiteľný materiál,
- *problém získavania* – ako formulovať a realizovať dopyty na repozitár v jednoduchom a presnom spôsobe.

9.1 Podstatné časti odporúčacích systémov

Znovupoužitie zdrojového kódu záleží na schopnosti objaviť čo najväčšie množstvo užitočných časti zdrojových kódu. Preto sa v tejto časť bližšie pozrieme na pozadie odporúčacieho systému.

9.1.1 Proces znovupoužitia softvéru

Hlavnou úlohou vyhľadávača je nájsť znovupoužiteľné časti zdrojových kódov na základe určitých kritérií. Jednou z najjednoduchších foriem hľadania zhodnosti je možné použiť spôsob používaný spoločnosťou Google, ktorý hľadá na základe kľúčových slov v texte. Napr. vývojár chce nájsť funkciu v zdrojovom kóde ktorá vypočíta vzdialenosť, preto pre nájdenie takejto funkcie bude vyhľadávať pomocou kľúčových slov „getDistance int“ (slovenské pomenovanie funkcie = získajVzdialenosť). Avšak, ak chce vývojár nájsť špecifickejšiu funkciu, tak hľadanie pomocou Google spôsobu, alebo pomocou jednoduchých kritérií nenachádza veľmi presné výsledky. Vyhľadávanie preto treba skonkretizovať, prípadne doplniť o ďalšie špecifikácie. Tabuľka 9.1 zobrazuje ďalšie možnosti štyroch vyhľadávacích techník, kde je možné vidieť, že vyhľadávanie na základe kľúčových slov našlo mnoho nevhodných častí zdrojového kódu. Na druhej strane, vyhľadávanie založené na základe rozhraní má najväčšiu mieru zhodností žiadaných znovupoužiteľných artefaktov.

Tabuľka 9.1 Presnosť techník vyhľadávania kódu.

	Zhody podpisu	**Na základe kľúčových slov**	**Na základe názvov**	**Na základe rozhrania**
Priemerná presnosť	0,9 %	16,3 %	17,2 %	53,7 %
Štandardná odchýlka	1,8 %	21,9 %	19,3 %	22,4 %

Obrázok 9.1 ilustruje proces znovupoužitia, kedy vývojár sa počas budovania softvéru rozhodne nájsť vhodný znovupoužiteľný artefakt (stav S1). Sformuluje svoje požiadavky na nájdenie vhodnej časti zdrojového kódu v tvare opisu, respektíve jeho vlastností (stav S2). Na základe opisu požiadaviek algoritmus vyhľadávača hľadá vhodné časti zdrojových kódov (stav S3). Z týchto nájdených vhodných kandidátov si vývojár vyberie takú nájdenú časť, ktorú pokladá za užitočnú. Niekedy sa stáva, že aktuálne vytváraný zdrojový kód je potrebné upraviť, predtým než bude možné znovupoužiť vybraný užitočný kus kódu. Avšak to vývojár nezistí, dokým ho skutočne nevyberie pre použitie a otestuje vhodnosť jeho použitia. Za ideálnych podmienok by sa takto mali otestovať všetci kandidáti (stav S4). Po vybratí, použití znovupoužiteľného kódu a prípadnej úprave cieľového zdrojového kódu je proces znovupoužitia na konci (stav S5) a novovytvorený (prípadne spojený a upravený) kus kódu je pridaný do databázy pre prípadné ďalšie použitie.

Celý tento proces znovupoužitia, od stavu S1 až po stav S5, sa môže počas vývoja softvéru opakovať aj niekoľkokrát.

Obrázok 9.1 Všeobecný pohľad na proces znovupoužitia.

9.1.2 Vyhľadávanie softvéru

Algoritmy vyhľadávačov sa líšia vzhľadom na účel ich použitia. Väčšina týchto vyhľadávačov sa združila okolo vývojových prostredí pre ktoré sú určené, ako napr. Code Recommenders[16].

Existujú dva hlavné dôvody, prečo používateľ chce začať vyhľadávať častí zdrojového kódu. Prvý nastáva v prípade, ak používateľ chce hľadať za účelom *znovupoužitia existujúceho kódu bez modifikácie,* kde predmet hľadania možno rozdeliť na tieto kategórie:

- úryvky kódu, obaľovače alebo syntaktické analyzátory,
- znovupoužiteľné dátové štruktúry, algoritmy a widgety grafického používateľského rozhrania (ďalej už len GUI) začleneného do implementácie,
- znovupoužiteľné knižnice začlenené do implementácie,
- znovupoužiteľné systémy ako začiatočný bod implementácie.

Druhým dôvodom však môže byť vyhľadávanie za účelom nájdenia *referenčných príkladov* z ktorých je možné sa niečo naučiť. V tomto prípade predmet hľadania možno kategorizovať na:

- použiteľný blok kódu ako príklad,
- ukážka implementácie dátovej štruktúry, algoritmu alebo prvku rozhrania,
- ukážka použitia knižnice,
- náhľad na podobné systémy pre inšpiráciu a nápad.

V tejto kapitole sa zameriavame na prípady znovupoužitia týchto častí:

- úryvkov kódu,
- komponentov,
- knižnice,
- skúšobného prípadu.

[16] CodeRecommenders, http://www.eclipse.org/recommenders/

9.2 Všeobecné charakteristiky odporúčacieho systému

V tejto časti si predstavíme „dobré praktiky“, ktoré sa môžu použiť pri návrhu a vytváraní nového odporúčacieho systému. Ich validita bola nezávislé potvrdená ako predmet štúdií v mnohých nezávislých prácach.

9.2.1 Charakteristika prípadov použitia odporúčača

V minulosti bol štandardný spôsob znovupoužitia kódu v ručnom vyhľadávaní zaujímavých častí zdrojového kódu a po nájdení sa skopíroval a vložil do aktuálne vyvíjaného projektu. Opakom štandardného spôsobu znovupoužitia kódu je pokus o automatizovanie procesu, kedy sa jednotlivé kroky namiesto používateľa vykonáva odporúčací systém, ktorý by využíval znalostí z už existujúcich zdrojových kódov.

Znovupoužitie komponentov

Najčastejšie prípad použitia odporúčacieho systému je pri návrhu znovupoužitia predtým napísaného kódu, či už ide o útržky kódu, alebo metód a tried k celým subsystémom a systémom, inak povedané ako komponentov. Tento typ znovupoužitia sa spája hlavne s komponentovo riadeným vývojom softvéru [2].

Znovupoužitie knižníc

Špeciálne v objektovo orientovanom vývoji projektov vývojári neustále navrhujú funkcionality ako bloky, ktoré poskytujú v tvare knižníc. Z hľadiska používania takto kompaktného kusu softvéru sa na prvý pohľad môže zdať ako uľahčenie pre vývojára z dôvodu jeho veľkého množstva potencionálne znovupoužiteľných častí, avšak v prípade použitia iba niektorých častí mávajú nevyriešené závislosti na pôvodnú knižnicu.

Pri tomto type znovupoužiti knižníc, frameworkov, alebo API vyvstávajú ešte ďalšie otázky, či si je vývojár vedomý ako sa daná knižnica používa, ktoré objekty sú pre neho potrebné, ako sú vytvárané a tak podobne.

Znovupoužitie testových prípadov

Tento typ odporúčania a vyhľadávania súvisí s agilnou metodológiou zvanou extrémne programovanie[17], kde vývoj softvéru sa riadi prostredníctvom testov [3]. Test, ktorý bol v minulosti použitý na otestovanie určitého zdrojového kódu, môže byť na základe svojej podobnej formulácie nového testu znovupoužitý pre nájdenie prislúchajúceho kódu.

[17] Extreme Programming, http://www.extremeprogramming.org/

9.2.2 Charakteristika návrhu

Navrhnúť dobrý odporúčací systém býva veľmi náročné, pretože používatelia vnímajú citlivo akýkoľvek zásah do ich práce, najmä ak ide o cudzí element. Ak užívatelia nadobudnú negatívne pocity, napr. že ich odporúčací systém brzdí v práci, tak ho zvyknú vypnúť, alebo priamo odinštalovať. Odporúčací systém by mal byť obzvlášť rýchly, poskytujúci kvalitnú nápoveď a mal by byť neobťažujúci pomocník pri riešení problémov.

Integrácia a použiteľnosť

Používateľ bude pracovať s koncovým rozhraním odporúčacieho systému a prostredníctvom neho ovládať aj samotný odporúčací systém, preto je dôležitou podmienkou vhodne navrhnúť jeho prívetivosť a použiteľnosť.

Softvéroví vývojári pri práci väčšinu svojho času trávia s integrovaným vývojovým prostredím. Preto najvhodnejší spôsob prijatia odporúčača používateľmi z hľadiska prívetivosti a vnímania je integrovať ho do samotného IDE, kde bude mať vývojár sústredenú všetku svoju činnosť. Integrácia môže napr. predstavovať formu doplnku, ktorý jednoducho rozšíri používateľove IDE.

Z hľadiska použiteľnosti by mal odporúčací systém ponúknuť vývojárovi riešenia a pomoc iba vtedy, ak sú užitočné v danom kontexte (napr. vloženia dodatočnej pomocnej premennej). Samozrejme ak to nevyhovuje, musí byť ponúknutá aj možnosť pre odmietnutie ponuky riešenia.

Autonómny agent v pozadí

Ako už bolo predtým naznačené, vyhľadávanie by nemalo v žiadnom prípade prerušiť aktuálnu činnosť vykonávanú vývojárom. Vďaka monitorovaniu činnosti vývojára si agent zbiera údaje vo forme zaznamenávania zmien vykonaných v aktuálne vyvíjanom projekte, čím si vytvára povedomie o aktuálnom kontexte. Vo vhodných momentoch agent obnoví svoj kontext a pošle dopyt na vyhľadávací systém, aby našiel možné zhody v minulých údajoch. Vyhľadávanie podobností môže trvať aj dlhší čas a práve preto by tieto úkony mali prebiehať asynchrónne na pozadí a používateľ by na ne nemal čakať. Najhorší scenár nastáva, ak používateľ v záujme vyhľadať predchádzajúce podobnosti zdrojového kódu musí prerušiť svoj aktuálny vývoj softvéru a to z dôvodu, že odporúčací systém aktuálne vyhľadáva podobnosti.

Byť si vedomý kontextu

Pri monitorovaní vývojára je dôležité pozbierať čo možno najviac užitočných informácií o kontexte aktuálne vyvíjaného softvéru alebo projektu, čo môže skvalitniť výber a ponuku odporúčaní. Vedomosť o kontexte sa získava napr. sledovaním polohy kurzora v zdrojovom kóde, alebo blokov a častí kódov, taktiež aj zdrojových súborov. Táto vedomosť o kontexte pomáha lepšie formovať dopyt na vyhľadávač a odbremeňuje vývojára od ručného špecifikovania požiadaviek, ktoré by želaný nájdený zdrojový kód mal obsahovať a byť nájdený.

Vyhodnotenie a umiestňovanie

Odporučenia, ktoré navrhne odporúčací systém, by nemali byť prezentované ako zhluk dát, ktoré našiel vyhľadávač, ale mali by mať určitú prezenčnú formu, pretože tá vplýva na vnímanie odporučení používateľom.

Predtým než sú výsledky odporučené mali by sa vždy predspracovať. Nájdené užitočné časti zdrojových kódov môžu byť zoradené napr. podľa počtu použitia, alebo vhodnosti voči aktuálnemu zdrojovému kódu, alebo projektu. Vhodnosť je dobré vyjadriť prostredníctvom poradia v ponuke pre vývojára, napr. čím vyššie umiestnenie tým vhodnejšie pre použitie.

Pripravený na požiadanie

Ako sme už predtým spomenuli, vďaka autonómnemu agentovi v pozadí môže odporúčací systém pracovať úplne asynchrónne a spúšťať vyhľadávanie užitočných častí bez zásahu používateľa. Výsledky takýchto paralelných hľadaní môžu byť dočasne uchovávané v pozadí, dokým si ich používateľ nevyžiada, alebo dokým nie je správny moment pre ich zobrazenie (napr. používateľ prestal na moment písať). Rozhodnutie zobraziť ponuku znovupoužiteľných častí môže byť spustené aj používateľom, napr. vo forme použitia kombinácie klávesových skratiek, čím sa celý proces zozbierania dát, vyhľadania v databáze, zotriedenia a ponúknutia dá do pohybu. To nastáva v prípade, keď odporúčací systém nemá pred načítané údaje.

9.3 Načrtnutie implementácie odporúčacieho systému

Odporúčací systém môže byť vytvorený mnohými spôsobmi. Doteraz spomínaných charakteristické vlastnosti odporučacieho systému použijeme pre jeho praktickejšie vytvorenie.

Z hľadiska správneho návrhu by mal byť odporúčač rozdelený na viacero menších častí, z ktorých by každá časť plnila svoju špecifickú úlohu, za ktorú by bola aj zodpovedná. Keby bol odporúčač vytvorený ako jedna veľká súčasť, mohol by svojou činnosťou spomaľovať vývojové prostredie používateľa. Ďalší kladný efekt modularity je, že v prípade potreby môže byť odporúčač pozmenený v konkrétnych menších častiach (moduloch) než zasiahnutia celého systému.

Z pomedzi rôznych vývojových prostredí ako základné, na ktorom bude odporúčač vytvorený, je možné použiť multiplatformové prostredie, napr. Eclipse[18] alebo NetBeans[19].

Agent v pozadí spúšťajúci vyhľadávanie

Počas písania zdrojového kódu býva používateľ pozorovaný agentom, ktorý vyhodnocuje jeho aktivitu. V pravidelných momentoch agent zozbiera novo pridané údaje, čím si vytvára a upresňuje povedomie o kontexte, ktorý aktuálny používateľ vytvára. Agent tento nový dátový obsah sformuluje ako podmienky a tie potom odovzdá ďalšiemu modulu pre spustenie vyhľadávania.

[18] Eclipse, https://eclipse.org
[19] NetBeans, https://netbeans.org/

Takto je zabezpečené, že odporúčací systém si je vždy vedomý kontextu, v ktorom používateľ aktuálne vyvíja a chcel získať odporúčenie.

Zbieranie dát s vytváraním kontextu a následným spustením vyhľadávania musí prebiehať v rozumných časových intervaloch a s mierou. Nevhodným spôsobom je začatie zbierania kontextu vždy po stlačení akéhokoľvek tlačidla klávesnice, čo má za následok neúmerné zaťaženie systému. Princípom celého tohto procesu je pred pripraviť si zoznam znovupoužiteľných kúskov zdrojových kódov v nápovedi ešte pred tým, ako sa používateľ rozhodne ich nájsť.

Okrem samostatného zberu údajov, ktorý prebieha paralelne popri aktuálnej činnosti používateľa, je možné agentovi dať vedieť záujem používateľa o vyhľadanie znovupoužiteľných kúskov kódov, napr. prostredníctvom klávesovej skratky. Väčšinou to bývajú prípady, keď používateľ chce poradiť, či aktuálne vytvorený zdrojový kód nie je podobný niektorému už predtým vytvorenému fragmentu.

Infraštruktúra vyhľadávania

Jednotlivé kúsky zdrojových kódov je potrebné niekde uchovávať, aby bolo možné s nimi vykonávať ďalšie operácie, napr. vyhľadávanie, porovnávania existujúcich kúskov kódu s novými, príp. pridávať nové kúsky kódu.

Uchovávanie kúskov zdrojových kódov je dobré oddeliť od zvyšnej časti odporúčacieho systému do formy databázy, ktorá bude uchovávať a pracovať s veľkým množstvom malých častí kusov zdrojového kódu. K jednotlivým častiam kusov zdrojových kódov je možné si napr. značiť ich užitočnosť a vhodnosť, čím sa spresní výber pri nápovede.

Ako databázu je možné použiť akúkoľvek relačnú databázu, napr. Oracle, Microsoft SQL Server, MySQL alebo PostgreSQL. V prípade použitia tohto riešenia je dobré umiestniť databázu na samostatný server, ktorý poskytuje dostatočný výkon bez obmedzovania ostatných častí, pričom odporúčací systém musí komunikovať a udržiavať spojenie s databázou prostredníctvom internetového pripojenia, čo môže spôsobiť určité spomalenie.

Avšak z hľadiska výkonu a minimalizácie času potrebného na vyhľadávanie je vhodnejšie použiť dokumentovo orientované textové databázy, napr. Lucene/Solr, alebo NoSQL databázy, akou je MongoDB. Tieto databázy je možné použiť aj v rámci odporúčača bez potreby použitia dodatočného hardvéru.

Vyberanie a radenie odporučení

Len čo sa z databázy vytiahnu výsledky, je možné ich považovať za finálne a ponúknuť používateľovi na použitie, napr. vo forme prvých desiatich nájdených zhôd.

Avšak používateľ pri požiadavke o nájdenie podobností očakáva, že nájdené podobnosti sa mu odporučia zoradeným spôsobom zostupne, kde na prvom mieste bude kandidát s najvyššou zhodou podobnosti. To je možné za pomoci takzvanej špekulatívnej analýzy [4], keď pred samotným zobrazením budú výsledky ohodnotené a upraví sa poradie zobrazovania. Tento modul pre úpravu zobrazovania výsledkov môže byť vytvorený dvoma spôsobmi:

- *Meranie vzdialenosti* – najjednoduchšia metóda je porovnávanie rozdielov nájdených výsledkov kusov zdrojových kódov. Porovnáva sa opis rozhrania aktuálne vyvíjanej triedy s opismi rozhraní v nájdenom zozname kúskov zdrojových kódov z databázy. Napr. keď aktuálne vyvíjaná trieda s metódami, ktoré prijímajú argumenty, bude mať rovnaký opis ako jedna v zozname, tak odchýlka medzi nimi bude nulová. To má za následok, že v novom upravenom zozname ponuky nápovede bude umiestnenie nájdenej triedy s nulovou vzdialenostnou odchýlkou na prvom mieste. Keď sa pri porovnaní aktuálne vyvíjanej triedy s inou v zozname bude zhodovať, napr. iba názvom triedy, tak jej vzdialenostná odchýlka bude väčšia, tak iná trieda v zozname bude umiestnená nižšie.
- *Testom riadené znovupoužitie* – v prípade, že projekt sa vyvíja prostredníctvom testov [3], je možné ich využiť ako porovnávací a hodnotiaci nástroj. Napr. pre aktuálne vyvíjanú triedu systém nájde jej testy, spúšťa ich a testuje na jednotlivých kúskoch zdrojových kódov nájdených v zozname z databázy. Podľa výsledkov zo spustenia testov na jednotlivých nájdených kandidátov sa na prvých miestach vhodnosti umiestnia zdrojové kódy, ktoré pre svoje použitie nepotrebujú žiadne ďalšie úpravy, čiže poskytujú rozhranie zhodné s tým, ktoré sa nachádza v teste. Na ďalších miestach sa umiestnia tie, ktoré ak by sa použili, tak je nutné rozšíriť ich o dodatočnú funkcionalitu vo forme adaptéra.

Bez ohľadu na použitie vyššie spomínaných techník v module pre vyberanie a usporiadanie, pre presnejšie usporiadanie je dobré použiť aj dodatočné techniky, podľa ktorých sa jednotlivé nájdené kusy zdrojových kódov zoradia, napr. podľa dĺžky času vykonávania, cyklickej komplexity[20], alebo počtu riadkov kódu.

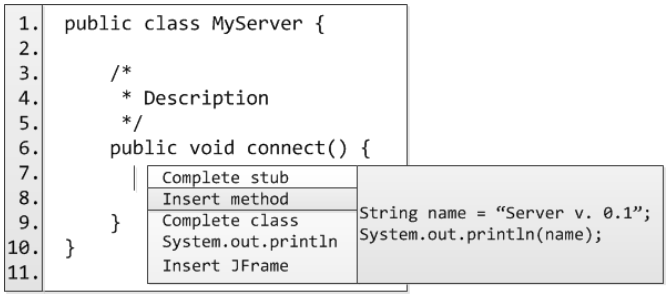
Praktická integrácia odporúčania

Posledným modulom v architektúre odporúčača je koncové rozhranie, prostredníctvom ktorého používateľ pracuje s odporučenými výsledkami kusov zdrojových kódov. Z hľadiska vnímania informácií je pre používateľa dôležité, aby odporúčač nepôsobil rušivo na jeho činnosť.

Vo väčšine vývojových prostredí existuje funkcionalita automatického dopĺňania textu, ktorá používateľovi automaticky doplní slovo, ktoré už raz napísal. V prípade, že novo písané slovo sa podobá na viac ako jedno predtým napísané, ponúkne sa mu zoznam slov, z ktorých si vyberie. Toto správanie je možné využiť aj pre zakomponovanie odporúčača.

Ak teda používateľ pri budovaní nejakej triedy sa rozhodne, že skúsi vyhľadať či už niečo podobné neexistuje, tak odporúčaču dá vedieť, že chce vyhľadávať, napr. stlačením klávesovej skratky. Na pozadí sa vykonajú všetky procesy opísané v predchádzajúcich častiach. Obrázok 9.2 znázorňuje zobrazenie odporúčania používateľovi formou ponuky doplnenia kódu.

[20] Metrika pre hodnotenie komplexnosti a zložitosti programov.



Obrázok 9.2 Odporúčanie integrované v ponuke automatického dopĺňania vývojového prostredia.

Štandardné správanie pri ovládaní tohto modulu zahŕňa: vybratie konkrétneho zdrojového kódu býva umožnené pomocou kurzora myši, alebo šípok na klávesnici, pričom použitie vybraného odporučenia sa potvrdí klávesou Enter a zrušenie ponuky klávesou Escape.

Okrem priamočiareho znovupoužitia jednotlivých kusov kódu pre aktuálne vyvíjaný súbor je možné vďaka testom použiť sofistikovanejšie znovupoužitie aj dodatočných súborov. Príkladom je situácia kedy sa používateľ rozhodne, že z ponúkaných volieb v odporúčači si zvolí posledné odporúčanie, ktoré do aktuálneho súboru vloží okrem znovu použiteľnej časti aj ďalšie potrebné súbory, ktoré s ním súvisia, príp. vytvorí adaptéry.

9.4 Zhrnutie

Vďaka open source projektom a ich voľnému prístupu k veľkému zdroju dát zdrojových kódov, otvorila sa pre softvérové inžinierstvo cesta ku skvalitneniu vývoja softvéru za pomoci vytvorenia odporúčacích systémov, ktoré dokážu analyzovať predchádzajúce vytvorené súčasti a podľa potrieb navrhnúť ich znovupoužitie. Týmto by sa docielil jednoduchší a hlavne rýchlejší vývoj.

Bohužiaľ, odporúčacie systémy snažiac sa túto myšlienku zrealizovať vznikali a ešte aj dnes vznikajú hlavne na akademickej pôde a v praxi sa len malé množstvo z nich aj reálne uplatní. Mnoho odporúčacích systémov, ktoré boli v minulosti vytvorené sa kvôli nedostatku kapacít nepodarilo udržať v aktuálnosti a zostarli – čiastočne ich algoritmus a čiastočne aj zoznam užitočných kusov kódu. Do budúcna sa otvára otázka, ako čo možno najlepšie vytvoriť odporúčací systém, ktorý by bol všeobecne prijímaný, udržovateľný a dostatočne užitočný.

Okrem mnohých možných realizácií odporúčacieho systému sme rozobrali aj jednu z nich, kde sme náš odporúčací systém rozdelili na moduly z ktorých každý plnil svoju čiastkovú úlohou pre konečný výsledok, t.j. navrhnutie znovupoužiteľných častí.

Literatúra

[1] Krueger, C. W. Software reuse. In: ACM Computing Surveys (CSUR), 1992. Zv. 24.

[2] Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D. Modeling Components and Component-Based Systems in KobrA. The Common Component Modeling Example. In: Springer Berlin Heidelberg, 2008.

[3] Beck, K. Test Driven Development: By Example. In: Addison-Wesley Professional, 2002.

[4] Brun, Y., Holmes, R., Ernst, M. D., Notkin, D. Speculative analysis: Exploring future states of software. In: Proceedings of the 2010 Foundations of Software Engineering Working Conference on the Future of Software Engineering Research, 2010.

10 Odporúčanie zmien v softvéri

Zmeny v softvéri nezodpovedajú dobrým návrhovým princípom, pretože často nevychádzajú z návrhu, ale sú aplikované na existujúci softvérový systém. Preto kvalita softvéru klesá, softvér je stále ťažšie udržiavať a dôsledok je nižšia kvalita, ktorá súvisí s nižšou produktivitou. Takýto softvér potom musíme prerobiť, čo vyžaduje veľké úsilie. Preto odporúčanie refaktorovacích operácií môže prispieť k zvýšeniu kvality softvéru. Aplikáciou refaktoringu môžeme zlepšiť čitateľnosť, znížiť zložitosť zdrojového kódu, lepšie vyjadriť vnútornú architektúru a softvér je lepšie rozširovateľný. Odporúčaním refaktoringu môžeme programátorovi ponúknuť alternatívu k jeho vlastnému uvažovaniu. Často zložitosť softvéru je vysoká a preto refaktoring, ktorý berie do úvahy celý softvérový systém sa môže zdať pre programátora príliš náročný, no aplikáciou algoritmov môžeme extrahovať z tohto zložitého softvérového systému podstatné atribúty a následne tie odporučiť, ktoré v konečnom dôsledku môžu programátorovi pomôcť.

Zdrojový kód prechádza, najmä pri agilných prístupoch, množstvom zmien. Už pri prvej iterácii sa nám dostávajú prvé zárodky kódu, ktoré potom pri ďalších iteráciách sa zväčšujú, narastajú na zložitosti a časom sa stávajú neprehľadné alebo sa úplne zmenia alebo transformujú. Žiaľ, súčasné techniky modularizácie nevedú k samo-udržateľnému, prehľadnému a modulárnemu zdrojovému kódu a techniky refaktoringu stále musíme aplikovať.

Zmeny v softvéri nie sú iba na úrovni zdrojového kódu, sú prítomné a pochádzajú zväčša od vyšších miest, od klienta alebo ako odpoveď na súčasný stav alebo poznanie problematiky realizačného tímu. Potom sa tieto zmeny transformujú cez mnoho ľudí v procese vývoja, napr. cez dizajnérov, architektov až k programátorom. Snaha vývojového tímu je potom navrhnúť čo najmodulárnejší a najlepší návrh, aby navrhovaná modularizácia mohla ľahko reflektovať zmeny z vrchu a tiež, aby nebola príliš komplikovaná a nespôsobila záťaž na rozpočet projektu. V pod-

state ide o kompromis, ktorý sa odvíja od skúseností tímu, požiadaviek a veľký vplyv na to má aj financovanie projektu.

Nielen počas vývoja, ale aj počas údržby sú zmeny prítomné. Akoby zmeny charakterizovali softvérový systém, preto modularizácia a architektúra softvérového systému je veľmi dôležitá. Avšak, návrh a modularizácia často prichádza iba počas vývoja a v údržbe táto úloha je zväčša presunutá na programátora, ktorý využíva refaktoring na „nahradenie“ dizajnéra a architekta. Zmeny v softvéri neodpovedajú dobrým návrhovým princípom. Kvalita softvéru klesá, softvér je stále ťažšie udržiavať [17] a výsledkom je nižšia kvalita, ktorá súvisí s nižšou produktivitou [3]. Softvér potom musíme prerobiť, čo vyžaduje veľké úsilie.

Ukazuje sa, že analýzou zdrojového kódu a extrakciou dôležitých atribútov môžeme odporučiť refaktorovacie operácie, alebo dokonca priamo vykonať za programátora, a tak zlepšiť vlastnosti softvéru. Aplikáciou refaktoringu môžeme zlepšiť čitateľnosť, znížiť zložitosť zdrojového kódu, lepšie vyjadriť vnútornú architektúru, a prispieť k rozširovateľnosti softvéru [14].

Hoci objektovo-orientovaný zdrojový kód prejde refaktoringom, stále nie je dobre čitateľný a programátor musí kód študovať, najmä kvôli modularizácii, ktorá roztrúsi podobný kód do objektov. Teda, refaktoring predstavuje akési „zaplátanie“ problému čitateľnosti a modularizácie softvéru v objektovo-orientovaných programovacích jazykoch (kód nie je čitateľný a modulárny prirodzene). Teda, refaktoringom neriešime problém, ale iba symptómy. Vytvárame nad objektovo-orientovaným prístupom záplaty, ktorými problém riešime len dočasne.

Dôsledok odporúčania týchto „záplat“ potom vedie k riadeniu programátora odporúčacím systémom. Programátor nielen stráca prehľad o systéme, ale nepotrebuje rozmýšľať a stáva sa bez odporúčacieho systému nesamostatný. Podobne, ako pri integrovaných vývojových prostrediach (IDE), ktoré umožňujú automatizácie mnohých procesov, napr. refaktorovacie operácie, a programátor nevie riešiť úlohy bez daného vývojového prostredia, napr. konflikty viacerých verzii v zdrojovom kóde a spájanie týchto verzií.

Táto práca sa venuje odporúčaniu zmien v populárnych objektovo-orientovaných programovacích jazykoch. Práca ďalej opisuje a diskutuje odporúčacie systémy pre refaktoring, ich implementácie a možný benefit pre programátora.

10.1 Zmeny v softvéri a refaktoring

Ako sme poukázali v úvode, zmeny v softvéri sú prítomné stále. Preto v systéme modularizovaného do objektov môžu tieto zmenyspôsobiť pachy v kóde a anti-vzory. Pachy v kóde sú symptómy možných návrhových problémov v zdrojovom kóde:

- duplikovaný kód,
- dlhá metóda,
- veľká trieda,
- veľa parametrov,
- iné.

Anti-vzory sú časté zlé a neefektívne riešenia častých problémov [11, 22]. Refaktoring sa usiluje o riešenie na zlepšenie kvality softvéru – refaktoring je proces zmeny softvérového systému tak, že nezmení jeho vonkajšie správanie, ale zlepší jeho štruktúru [14]. Z predchádzajúcej definície sa vynárajú zjavné výhody refaktoringu [14]:

- zlepšenie čitateľnosti,
- zníženie zložitosti zdrojového kódu,
- lepšie vyjadrená vnútorná architektúra,
- lepšia softvérová rozširovateľnosť.

10.2 Refaktorovacie operácie

V tejto sekcii rozoberieme jednotlivé refaktorovacie operácie, ktoré upravujú zdrojový kód tak, aby odstránili pach v zdrojovom kóde alebo anti-vzor:

- *vyňatie triedy* – rozdelenie zložitej a málo súdržnej triedy do nových tried s lepšie definovaným správaním,
- *vyňatie balíku* – rozdelenie zodpovednosti balíku do rozdielnych súdržných balíkov s rovnakým správaním,
- *vyňatie metódy* – rozdelenie zložitej metódy do nových metód, kde každá implementuje špecifickú a izolovanú akciu,
- *presunutie metódy* – presunutie metódy do triedy obsahujúcej bližšie správanie.

10.2.1 Vyňatie triedy

Táto refaktorovacia operácia sa snaží o elimináciu anti-vzoru Kvapka (angl. Blob anti-pattern) rozdelením tried na dve alebo viaceré triedy. Kvapka je veľká trieda, ktorá má veľa atribútov a operácií. Dôsledky sú tieto [4]:

- negatívny dopad na pochopenie kódu a údržbu,
- nedá sa znovu použiť, pretože je príliš zložitá,
- drahá na načítanie.

Vyňatie triedy sa realizuje ako postupnosť [4]:

1. Analýza metód vo veľkej triede.
2. Identifikácia metód, ktoré implementujú podobnú funkcionalitu.
3. Distribúcia atribútov do metód.
4. Rozdeľovanie triedy do nových tried, ktoré obsahujú vybrané metódy a atribúty.
5. Uistenie sa, že nenastali žiadne zmeny v správaní softvéru.

Ako prvé analyzujeme metódy triedy a vyberáme metódy, ktoré implementujú podobnú funkcionalitu. K týmto metódam vyberáme aj atribúty. Z týchto podobných metód a atribútov vzniknú nové triedy. Potom sa už iba uistíme, či nenastali zmeny vo vonkajšom správaní softvérového systému a anti-vzor Kvapka by sa mal rozplynúť v týchto triedach.

Podľa Fokaefs ide o klasický klastrový analyzačný problém a vieme ho vypočítať [13]. Pre každý atribút vyberieme *Ei* – metódy používajúce atribút. Potom, pre každú metódu vyberieme *Ej* – metódy, ktoré sú ňou volané a všetky atribúty ku ktorým metóda pristupuje. Medzi týmito dvomi skupinami vypočítame Jaccardovu vzdialenosť:

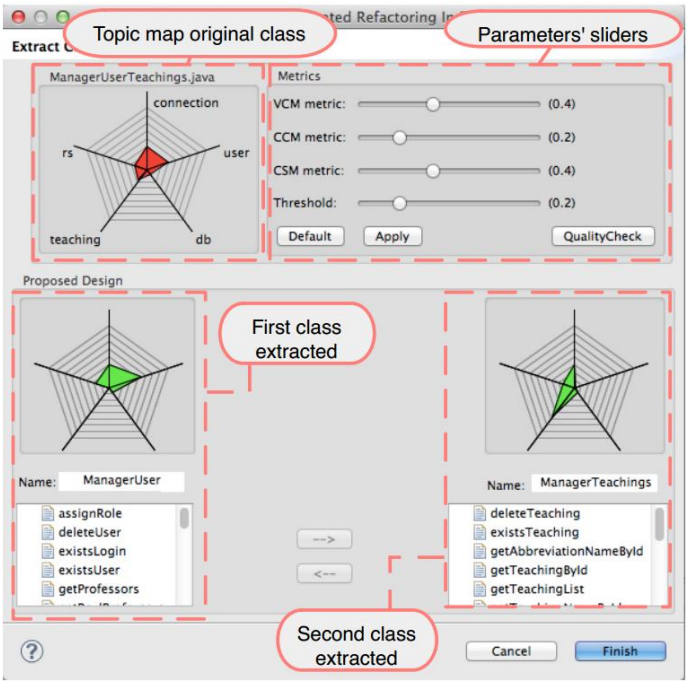
$$J(Ei, Ej) = 1 - \frac{\text{počet rovnakých Ei a Ej}}{\text{počet všetkých Ei a Ej}} \tag{10.1}$$

Vzdialenosťami medzi metódami a atribútmi vieme určiť podobné prvky v triede. Tento algoritmus (Jaccardova vzdialenosť) je implementovaný v zásuvnom module JDeodorant pre Eclipse. Podľa Fokaefs empiricky táto metóda je presná na 67 %.

Bavota a kol. použili na vypočítanie riešenie založené na teórii grafov [5]. Prístup sa pozerá na problém ako na veľký graf, kde každá metóda je inak podobná tej druhej. Na základe tejto podobnosti vieme povedať, ktoré sú bližšie ku sebe a nakoniec, ktoré sú podgrafy. Podgrafy potom tvoria nové triedy.

Podobnosti sú vypočítané metódami SSM, CDM a CSM, ktoré vysvetľujeme ďalej. Štrukturálna podobnosť (SSM) je podľa počtu zdieľaných premenných. Závislosť podľa volaní (CDM) je podľa počtu volaní medzi sebou. Konceptuálna podobnosť (CSM) je podľa textovej podobnosti. Tieto podobnosti sú skombinované na základe váh do matice. Celkový postup je teda nasledujúci:

1. Vypočítanie podobností metód (SSM, CDM, CSM).
2. Identifikácia zreťazenia metód podľa matice do podgrafov.
3. Identifikácia triviálnych reťazí a ich rozpojenie.
4. Podgrafy ako nové triedy.

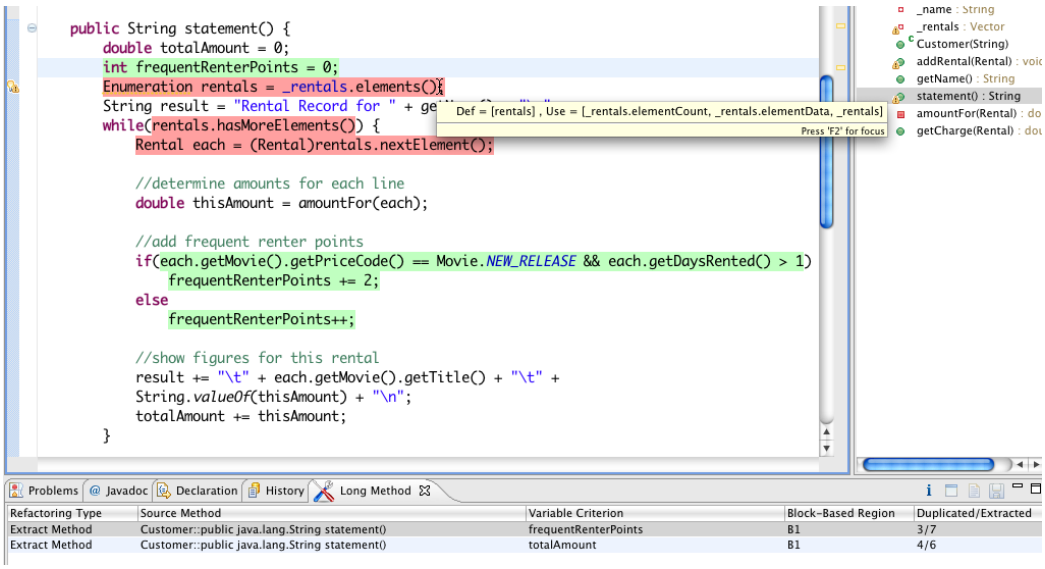


Obrázok 10.1 ARIES zásuvný modul do Eclipse [7]

Toto riešenie založené na teórii grafov je implementované v zásuvnom module ARIES do Eclipse. Empiricky je podľa autorov presné na 91%. Obrázok 10.1 znázorňuje zásuvný modul ARIES, kde vpravo hore vyberáme váhy pre SSM, CDM a CSM, vľavo je pôvodná trieda a vpravo je navrhovaná vybraná trieda.

10.2.2 Vyňatie balíku

Refaktorovacia operácia *Vyňatie balíku* sa podobá operácii *Vyňatie triedy*. Rozdiel je v inej úrovni granularity. Teda sa snažíme rozdeliť balíky, ktoré spájajú rozdielne triedy. Bavota a kol. aplikovali riešenie založené na teórii grafov na refaktoring *Vyňatie balíku* a vyhodnotili riešenie na 5 systémoch [6]. Refaktorovacie operácie mali zmysel pri 77 %.



Obrázok 10.2: JDeodorant zásuvný modul do Eclipse[21]

10.2.3 Vyňatie metódy

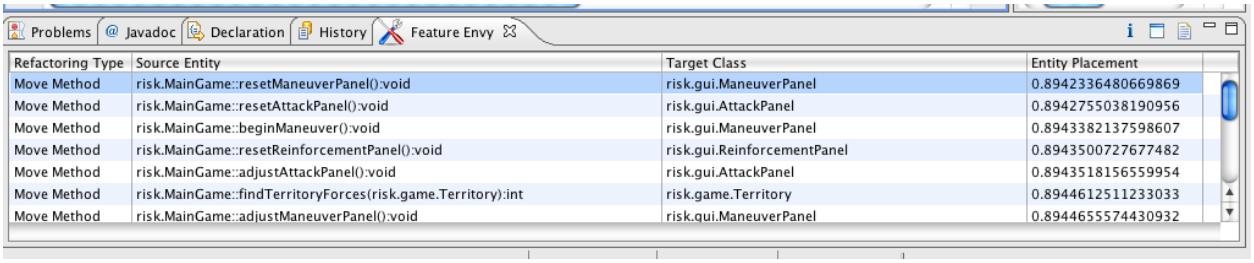
Maruyama a kol. počítali refaktoring *Vyňatie metódy* riešením založeným na historickej informácii. Vytvorili váhovaný graf závislostí WPDG [18], kde jeden uzol je jeden príkaz. Hrany boli sily závislosti podľa histórie modifikácie zmien. Takto mali jednotlivé príkazy medzi sebou vzťah a tie ktoré nemajú, alebo ho majú príliš slabý, môžu byť vybrané do novej metódy.

Weiser a kol. použili riešenie založené na krájaní [24]. Plátok (akoby odkrojený kus zdrojového kódu) je spustiteľná podmnožina príkazov, ktorá zachová pôvodné správanie metódy, ale tento plátok je extrahovaný do novej metódy. Existuje viac druhov krájania:

- *jemné krájanie* – identifikovanie vykonateľných plátkov [1],
- *plátok kompletného výpočtu* – identifikácia príkazov ovplyvňujúcich premennú [23],
- *plátok objektového stavu* – zachytenie príkazov, ktoré ovplyvňujú daný objekt [4].

[21] About JDeodorant. URL: http://www.jdeodorant.com/

Metóda krájania bola aplikovaná a implementovaná v zásuvnom module JDeodorant pre Eclipse. Obrázok 10.2 znázorňuje príklad extrakcie založenej na krájaní. Zeleným a červeným na obrázku sú odlíšené riadky, ktoré spolu nesúvisia.



Refactoring Type	Source Entity	Target Class	Entity Placement
Move Method	risk.MainGame::resetManeuverPanel():void	risk.gui.ManeuverPanel	0.8942336480669869
Move Method	risk.MainGame::resetAttackPanel():void	risk.gui.AttackPanel	0.8942755038190956
Move Method	risk.MainGame::beginManeuver():void	risk.gui.ManeuverPanel	0.8943382137598607
Move Method	risk.MainGame::resetReinforcementPanel():void	risk.gui.ReinforcementPanel	0.8943500727677482
Move Method	risk.MainGame::adjustAttackPanel():void	risk.gui.AttackPanel	0.8943518156559954
Move Method	risk.MainGame::findTerritoryForces(risk.game.Territory):int	risk.game.Territory	0.8944612511233033
Move Method	risk.MainGame::adjustManeuverPanel():void	risk.gui.ManeuverPanel	0.8944655574430932

Obrázok 10.3: JDeodorant zásuvný modul, odporúčanie metód, ktoré majú mnoho referencií [1]

10.2.4 Presunutie metódy

Atkinson a kol. počítali *Presunutie metódy* refaktoring na základe analýzy abstraktného syntaktického stromu [2]. Každému atribútu počítali množinu metód, ktoré ho používajú. Ak má veľa takýchto referencií, metóda by mala byť preskúmaná ručne. Prístup implementovali v zásuvnom module JDeodorant pre Eclipse. Obrázok 10.3 uvádza odporúčanie metód s veľkým počtom referencií pomocou JDeodorant.

10.3 Vytváranie odporúčacieho systému pre refaktoring

Na vytvorenie odporúčacieho systému najskôr potrebujeme poznať vzťahy medzi komponentmi kódu. Potom z nich môžeme počítať nejakým algoritmom informáciu, ktorú môžeme odporučiť. Táto sekcia sa venuje tomu, ako vytvoriť taký odporúčací systém.

10.3.1 Zachytávanie vzťahov medzi komponentmi kódu

Môžeme použiť nasledovné zdroje informácii pre získavanie vzťahov medzi komponentmi [4]:

- volania metód,
- zdieľané premenné,
- vzťah dedičnosti,
- informácia o pôvodnej dekompozícii,
- sémantická podobnosť.

Ako už bolo spomenuté, tieto informácie môžu byť použité na získanie dát, z ktorých môžeme niečo vyvodiť podľa algoritmu a tak pomôcť programátorovi, napr. urobiť odporúčanie.

10.3.2 Algoritmus na generovanie odporúčaní pre refaktoring

Rôzne prístupy sa dajú použiť na rôzne refaktorovacie operácie. Ako sme už ukázali v práci, máme klastrovo-založené, grafovo-založené, heuristicky-založené, založené na vyhľadávaní,

založené na krájaní [4]. Na každú refaktorovaciu operáciu sa dá použiť iný prístup. Zatiaľ ešte neboli vysvetlené heuristicky-založené a založené na vyhľadávaní:

- *heuristicky-založené* – používajú sa na riešenie refaktoringu *Presunutie metódy* heuristickou aproximáciou optimálneho počtu klastrov [21],
- *založené na vyhľadávaní* – používajú sa na riešenie refaktoringov *Presunutie metódy* a *Presunutie triedy* [19].

10.4 Vyhodnotenie

Môžeme použiť tri techniky vyhodnocovania odporúčacieho systému [4]. Prvá z nich je založená na metrikách kvality. Meraním pred aplikáciou refaktoringu a po aplikácii refaktoringu môžeme vyhodnotiť, či refaktoring mal zmysel, napr. meranie vzťahov medzi triedami založené na volaniach (MPC) priamo koreluje s úsilím, ktoré treba vynaložiť na údržbu [16]. Meranie vzťahov medzi metódami založené na sémantike (CCBC) – pravdepodobnosť kedy nastane zmena, ak som zmenil podobnú metódu, znižuje čas potrebný na nájdenie zmeny [20].

Druhá technika môže byť vyhodnotenie založené na historickej informácii [4]. Tu ako prvé identifikujeme refaktorovacie operácie programátorom. Potom aplikujeme refaktorovaciu operáciu. Identifikujeme refaktorovacie operácie strojom pred aplikáciou a meriame zhodu.

Posledná tretia technika je vyhodnotenie s programátormi [4], kedy sa porovnávajú hodnotenia odporúčaní pôvodnými a externými programátori.

10.5 Zhrnutie

Odporúčanie zmien v softvéri je stále vo vývoji a ešte bude vyžadovať veľa úsilia, aby sa tieto odporúčacie systémy stali naozajstnými pomocníkmi pri programovaní. Obmedzené možnosti v integrovaných vývojových prostrediach ukazujú ich nevyspelosť a otázny benefit.

Je veľa štúdii, ktoré sa venovali refaktoringu. V štúdii od Bavota a kol. analyzovali aplikáciu 12 922 refaktorovacích operácii [8], pričom 15% z nich boli chyby v systéme. Ďalšia štúdia od Kim a kol., kde vzorka bola 328 Microsoft programátorov [15], zistili toto:

- 50% nedefinuje refaktoring ako operáciu, ktorá zachováva správanie,
- 51% robí refaktorovacie operácie manuálne,
- 71% programátorov sa bojí, že refaktorovaním spôsobia chybu.

Problém odporúčacích systémov, ako ukazuje aj literatúra a štúdie, by sme videli v tom, že tieto systémy neobsahujú informáciu, prečo navrhli danú refaktorovaciu operáciu. A tiež, aktuálne, žiadny nástroj nepodporuje integrovanú podporu všetkých refaktorovacích operácií.

Literatúra

[1] Abadi, A., Ettinger, R., Feldman, Y.A.: Fine slicing: theory and applications for computation extraction. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 7212, pp. 471–485 (2012). doi:10.1007/978-3-642-28872-2_32?

[2] Atkinson, D.C., King, T.: Lightweight detection of program refactorings. In: Proceedings of the Asia–Pacific Software Engineering Conference, pp. 663–670 (2005). doi:10.1109/APSEC.2005.76

[3] Basili, V.R., Briand, L., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE T. Software Eng. 22(10), 751–761 (1995). doi:10.1109/32.544352

[4] Bavota G. et al., Recommending Refactoring Operations in Large Software Systems. In: Recommendation Systems in Software Engineering, pp. 387-419, doi: 10.1007/978-3-642-45135-5

[5] Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Automating extract class refactoring: an improved method and its evaluation. Empir. Software Eng. (2013a, in press). doi:10.1007/s10664-013-9256-x

[6] Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Using structural and semantic measures to improve software modularization. Empir. Software Eng. 18(5), 901–932 (2013b). doi:10.1007/s10664-012-9226-8?

[7] Bavota G. et al., ARIES: An Eclipse plug-in to Support Extract Class Refactoring, URL: http://2013.eclipse-it.org/proceedings/3_eclipse-IT-bavota.pdf

[8] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O.: When does a refactoring induce bugs?: an empirical study. In: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 104–113 (2012a). doi:10.1109/SCAM.2012.20

[9] Briand, L.C., Wuest, J., Lounis, H.: Using coupling measurement for impact analysis in object-oriented systems. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 475–482 (1999a). doi:10.1109/ICSM.1999.792645

[10] Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)

[11] Budgen, D. (2003). Software design. Harlow, Eng.: Addison-Wesley. p. 225. ISBN 0-201-72219-4. "As described in Long (2001)

[12] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE T. Software Eng. 20(6), 476–493 (1994). doi:10.1109/32.295895

[13] Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: Identification and application of extract class refactorings in object-oriented systems. J. Syst. Software 85(10), 2241–2260 (2012). doi:10.1016/j.jss.2012.04.013

[14] Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley, Reading, MA (1999)

[15] Kim,M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 50:1–50:11 (2012). doi:10.1145/2393596.2393655

[16] Lee, Y.S., Liang, B.S.,Wu, S.F.,Wang, F.J.:Measuring the coupling and cohesion of an object oriented program based on information flow. In: Proceedings of the International Conference on Software Quality, pp. 81–90 (1995)

[17] Li, W., Henry, S.: Maintenance metrics for the object oriented paradigm. In: Proceedings of the International Software Metrics Symposium, pp. 52–60 (1993). doi:10.1109/METRIC.1993.263801

[18] Maruyama, K., Shima, K.: Automatic method refactoring using weighted dependence graphs. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 236– 245 (1999). doi:10.1145/302405.302627

[19] O'Keeffe, M., Ó Cinnéide, M.: Search-based software maintenance. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 249–260 (2006). doi:10.1109/CSMR.2006.49

[20] Poshyvanyk, D., Marcus, A., Ferenc, R., Gyimóthy, T.: Using information retrieval based coupling measures for impact analysis. Empir. Software Eng. 14(1), 5–32 (2009).

[21] Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J. Comput. Appl. Math. 20, 53–65 (1987). doi:10.1016/0377-0427(87)90125-7

[22] Scott W. Ambler (1998). Process patterns: building large-scale systems using object technology. Cambridge, UK: Cambridge University Press. p. 4. ISBN 0-521-64568-9.

[23] Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Software 84(10), 1757–1782 (2011). doi:10.1016/j.jss.2011.05.016

[24] Weiser, M.: Program slicing. IEEE T. Software Eng. 10(4), 352–357 (1984). doi:10.1109/TSE.1984.5010248

11 Odporúčanie systematických zmien zdrojového kódu

Jedným z hlavných cieľov správneho objektovo-orientovaného návrhu softvéru je zamedzenie opakovania sa zdrojového kódu a dosiahnutie vysokej súdržnosti a nízkej zviazanosti súčiastok. Tým sa však nevyhneme opakujúcim sa implementáciám, napr. kvôli obmedzeniam zvoleného programovacieho jazyka, alebo oprave podobnej chyby na viacerých miestach. Častokrát je táto činnosť závislá od detailov úlohy, ktorú programátor rieši, od samotnej implementácie a najmä od vedomostí programátora. Tieto zmeny vykonáva ručne a systematicky. Môžeme mu však prácu uľahčiť odporúčaním alebo ich na pokyn programátora automatizovať.

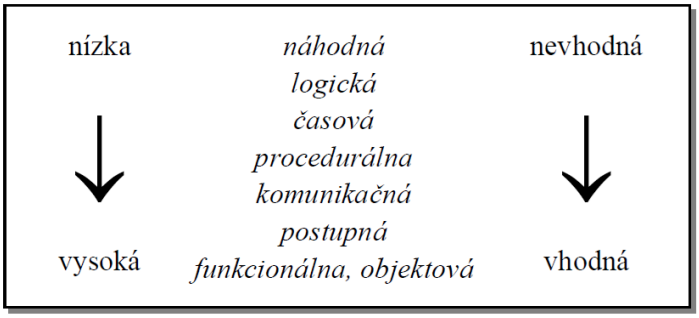
Počas vývoja softvéru sa programátori stretávajú so situáciami, keď zavádzajú rovnaké alebo podobné úpravy na viacerých miestach v zdrojovom kóde. Takéto zmeny označujeme ako *systematické zmeny* [3], t.j. systematické vykonávanie podobných, no čiastočne odlišných zmien na viacerých miestach v zdrojovom kóde. Pri správnom objektovo-orientovanom návrhu sa snažíme vyhnúť opakovaniu zdrojového kódu vhodnou dekompozíciou. To je však častokrát nemožné aj z týchto dôvodov:

- obmedzenia použitých technológií – napr. jazyky Java alebo C# nepodporujú viacnásobné dedenie a rovnakú funkcionalitu musíme implementovať v triedach súčasne,
- opakovanie sekvencií volaní vo viacerých metódach, aj naprieč rôznymi triedami,
- použitý jazyk implementácie neumožňuje odčlenenie funkcionality a jej znovupoužitie, napr. pri používateľskom rozhraní v značkovacom jazyku, napr. HTML, XML, XAML.

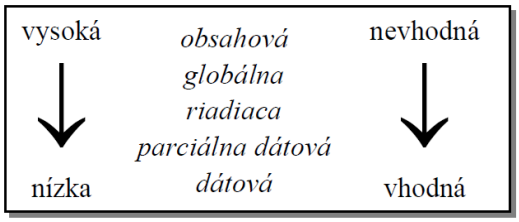
Z týchto dôvodov a mnohých ďalších, programátor pri opravení problému alebo doplnení funkcionality na jednom mieste v zdrojovom kóde musí vykonať rovnakú alebo podobnú úpravu aj v inej súčiastke zdrojového kódu. Príkladmi sú situácie:

- zmena štruktúry súčiastky si vyžiada zmenu všetkých súčiastok, ktoré sú na nej závislé,
- opravenie rovnakej alebo podobnej chyby vo viacerých súčiastkach naraz,
- pridanie podmienky overujúcej hodnoty premennej pred volaním metódy.

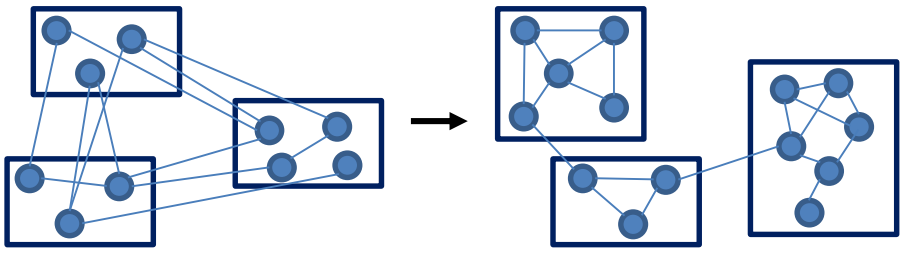
Keďže vykonávanie systematických zmien závisí od programátorovej znalosti zdrojového kódu a pozornosti, vidíme možnosť uplatnenia odporúčacích systémov. Odporúčanie systematických zmien môžeme vykonať na rôznych úrovniach – *programovanie príkladom, návrh miest na úpravu* a *transformácia z príkladov* [3]. V tejto kapitole sa venujeme uvedeným typom odporúčania systematických zmien, možných kritérií porovnania existujúcich systémov, a aj príkladom použitia.



Obrázok 11.1 Úrovne súdržnosti súčiastok zdrojového kódu a ich vhodnosť. Obrázok prevzatý z [1].



Obrázok 11.2 Úrovne zviazanosti súčiastok zdrojového kódu a ich vhodnosť. Obrázok prevzatý z [1].



Obrázok 11.3 Príklad závislostí súčiastok zdrojového kódu. Naľavo nevhodná nízka súdržnosť a vysoká zviazanosť, na pravej strane vhodná vysoká súdržnosť a nízka zviazanosť.

11.1 Motivácia

V prípade objektovo-orientovaného návrhu softvérového produktu sa snažíme o odčlenenie funkcionality do samostatných a voľne prepojených súčiastok, t.j. dosiahnuť ich vysokú súdrž-

nosť[22] a nízku zviazanosť[23]. Ako uvádzame na obrázkoch 11.1 a 11.2, poznáme viacero úrovní splnenia týchto vlastností [1], ktorých dosiahnutím očakávame, že pri následnom upravovaní zdrojového kódu robíme len lokálne úpravy v zasiahnutých súčiastkach. Obrázok 11.3 uvádzame príklad závislostí súčiastok zdrojového kódu v hraničných situáciách.

```
public IEnumerable<Product> RefreshProducts()
{
   if (Utils.HasInternetConnection())
   {
      SvcClient client = CreateSvcClient();
      using (DatabaseContext db = new DatabaseContext())
      {
         var products = client.GetLatestProducts();

         ProductList.Clear();
         foreach (Product product in products)
         {
            ProductList.Add(product);
            db.Products.Add(product);
         }
         db.SaveChanges();
      }
      NotifyProductsChanged();
      return products;
   }
   else
   {
      NotifyError();
   }
   return Enumerable.Empty<Product>();
}
```

Ukážka 11.1 Zdrojový kód príkladu metódy volania webovej služby a uloženia výsledkov do databázy v jazyku C#.

Aj v správne štruktúrovanom kóde sa však nevyhneme opakujúcim sa postupnostiam závislostí na iné súčiastky, napr. volania metód v danom poradí. Tok riadenia musí byť zachovaný pre isté operácie a výmena ich poradia ovplyvní výsledok, napr. nemôžeme zapísať záznamy do databázového systému skôr, ako sa k nemu pripojíme. Ukážka 11.1 uvádza príklad zdrojového kódu metódy v jazyku C#, v ktorej sa postupne vykonávajú tieto činnosti:

- kontrola pripojenia k internetu,
- prístup k databáze,
- volanie webovej služby,
- spracovanie získaných dát z webovej služby,
- uloženie záznamov,
- oznámenie o výsledku aplikácii.

```
public ILaunchConfiguration[] getLaunchConfigurations
      (ILaunchConfigurationType type) throws CoreException {
   Iterator iter = getAllLaunchConfigurations().iterator();
   List configs = new ArrayList();
   while (iter.hasNext()) {
```

[22] Súdržnosť – miera uzavretosti súčiastok, ich nezávislosti od ostatných.
[23] Zviazanosť – miera prepojenia súčiastok, ich vzájomných väzieb.

```
            ILaunchConfiguration config = (ILaunchConfiguration)iter.next();
            if (config.getType().equals(type)) {
               configs.add(config);
            }
         }
         return (ILaunchConfiguration[])configs.toArray
            (new ILaunchConfiguration[configs.size()]);
      }

      protected List getLaunchConfigurations(IProject type) {
         Iterator iter = getAllLaunchConfigurations().iterator();
         List configs = new ArrayList();
         while (iter.hasNext()) {
            ILaunchConfiguration config = (ILaunchConfiguration)iter.next();
            IFile file = cfg.getFile();
            if (file != null && file.getProject().equals(project)) {
               cfgs.add(cfg);
            }
         }
         return cfgs;
      }
```

Ukážka 11.2 Zdrojový kód metód pre získanie konfiguračných nastavení vo vývojárskom prostredí Eclipse (jazyk Java).

Jednotlivé kroky sa môžu opakovať aj v iných súčiastkach zdrojového kódu. Ukážka 11.2 uvádza podobný príklad v zdrojových kódoch vývojárskeho prostredia Eclipse, kde sa riadiaca logika v uvedených metódach podobá. Zmena v logike vytvárania zoznamov konfiguračných nastavení (napr. overenie konfigurácií pred pridaním do zoznamu) v jednej metóde by si pravdepodobne vyžiadala zmenu aj v druhej metóde. Ukážka 11.3 uvádza príklad tejto zmeny so zvýraznením pridaných aj odstránených riadkov.

Vykonávanie systematických zmien v zdrojovom kóde je náročná úloha programátora z viacerých uhlov pohľadu [3]:

- Znalosť programátora o všetkých miestach v zdrojovom kóde, ktoré musí skontrolovať, príp. aj upraviť podľa jeho zmeny [2],
- Časová náročnosť úprav na viacerých miestach súčasne [5, 7],
- Nezáživná práca opakovaného vykonávania rovnakej zmeny na viacerých miestach [5],
- Vysoká pravdepodobnosť chyby programátora z nepozornosti [9].

Jednou z možností zvýšenia efektívnosti programátora je použitie nástrojov pre refaktoring (vyňatie metódy, premenovanie názvu premennej alebo preusporiadanie parametrov funkcie), aké nájdeme napr. v nástrojoch Eclipse alebo Microsoft Visual Studio. Tie však nie je možné použiť na zložité transformácie zdrojového kódu, ktoré sa navyše neopierajú ani o sémantiku zdrojového kódu [5]. Systematické zmeny sú časté pri riešení chýb, kedy štúdie ukázali, že 17-45% opráv chýb sú opakujúce sa zmeny [8], alebo 75% zmien v zdrojových kódoch je štrukturálne podobných [4].

```
public ILaunchConfiguration[] getLaunchConfigurations
            (ILaunchConfigurationType type) throws CoreException {
  Iterator iter = getAllLaunchConfigurations().iterator();
  List configs = new ArrayList();
```

```
+ ILaunchConfiguration config = null;
  while (iter.hasNext()) {
-    ILaunchConfiguration config = (ILaunchConfiguration)iter.next();
+    config = (ILaunchConfiguration)iter.next();
+    if (!config.inValid()) {
+       config.reset();
+    }
     if (config.getType().equals(type)) {
        configs.add(config);
     }
  }
  return (ILaunchConfiguration[])configs.toArray
     (new ILaunchConfiguration[configs.size()]);
}

protected List getLaunchConfigurations(IProject type) {
  Iterator iter = getAllLaunchConfigurations().iterator();
  List configs = new ArrayList();
  while (iter.hasNext())
  {
-    ILaunchConfiguration cfg = (ILaunchConfiguration)iter.next();
+    cfg = (ILaunchConfiguration)iter.next();
+    if (!cfg.inValid()) {
+       cfg.reset();
+    }
     IFile file = cfg.getFile();
     if (file != null && file.getProject().equals(project)) {
         cfgs.add(cfg);
     }
  }
  return cfgs;
}
```

Ukážka 11.3 Systematická zmena v oboch metódach z ukážky 2 pridaním riadkov „+“ a odstránením „–“.

11.2 Odporúčanie systematických zmien

Existujúce systémy na podporu vykonávania systematických zmien môžeme rozdeliť do troch kategórií podľa miery zapojenia programátora [3]:

- *programovanie príkladom* – identifikácia a odporúčanie vzorov (skratiek) úprav zdrojového kódu z aktivity programátora, napr. vhodné pre textové dokumenty, pretože neuvažujeme syntax ani sémantiku jazyka,
- *návrh miest na úpravu* – identifikácia miest a ich odporúčanie, na ktorých by mal programátor vykonať podobnú zmenu ako spravil, napr. ako identifikácia duplicít,
- *transformácia z príkladov* – programátor identifikuje príklady, z ktorých sa odporučia transformácie zmien a programátor ich môže použiť s minimálnym ďalším zásahom.

Ako ďalšie kritérium systémov pre odporúčanie systematických zmien môžeme sledovať *typ zmeny*, ktorú dokážu vykonať. Najjednoduchšou možnosťou sú textové zmeny, za ktorými nasledujú zmeny rozlišujúce syntax, alebo dokonca aj sémantiku. Textové zmeny sú málo použiteľné pre upravenie zdrojového kódu, pretože môžu spôsobiť problémy so syntaktickou správnosťou zdrojového kódu. Nachádzajú si však uplatnenie pri automatizácii úprav textových dokumentov, alebo tabuliek v tabuľkových procesoroch (napr. Microsoft Excel). Rozlišovanie syn-

taxe pri textových úpravách sa vykonáva použitím abstraktného syntaktického stromu [6] zdrojového kódu, čím zabezpečíme neporušenie syntaxe výrazov daného jazyka (napr. cykly, podmienky, zápisy metód a tried). Zapojenie sémantiky však už prináša do systému väčšiu znalosť o upravovanom zdrojovom kóde. Systém tak napr. dokáže rozoznať iné názvy premenných v podobných kódoch alebo inak prispôsobiť vykonávanú zmenu, aby neporušila sémantiku upraveného kódu.

11.3 Vstupy a výstupy transformácie zdrojového kódu

Systémy pre odporúčanie systematických zmien pre transformáciu zdrojového kódu môžu vychádzať z viacerých rôznych zdrojov údajov. Vstupnými údajmi môžu byť:

- zaznamenané programátorove interakcie upravovania zdrojového kódu,
- rozdiely vo verziách dokumentov,
- interakcie vyhľadávania a nahradzovania,
- demonštrácia úpravy programátorom,
- príklady správneho kódu a chybného kódu.

Typ vstupných údajov samozrejme závisí aj od možnosti zásahu programátora do procesu transformácie, typu vykonávaných zmien aj abstrakcie úprav, t.j. či je navrhnutá zmena konfigurovateľná alebo nemenná.

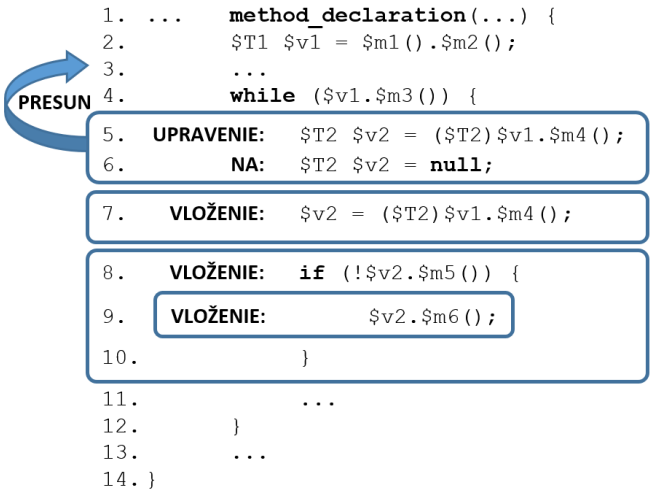
11.4 Existujúce riešenia

Spomedzi mnohých riešení pre odporúčanie systematických zmien uvedieme príklad použitia systému Sydit [6], ktorý identifikuje abstraktný vzor transformácie z príkladu zadaného programátorom s uvažovaním jeho kontextu. V úvode kapitoly sme uviedli príklad zmeny metódy `getLaunchConfigurations` (ukážky 11.2 a 11.3). Sydit reprezentuje zdrojový kód metódy pomocou abstraktného syntaktického stromu, zvlášť pre pôvodnú verziu a zvlášť pre upravenú. Koreňom abstraktného syntaktického stromu metódy je predpis metódy, postupné riadky metódy sa nachádzajú v potomkoch koreňa. Podobným spôsobom sa potom rozvetvujú cykly a podmienky.

Sydit porovná stromy a identifikuje medzi nimi rozdiel $\Delta_A = [e_1, e_2, \dots, e_n]$ ako postupnosť úprav *e* syntaktického stromu – operácie *vloženie vrcholu, zmazanie, upravenie* alebo *presunutie*.

Identifikovanú transformáciu Δ_A následne Sydit zovšeobecní nahradením názvov premenných, typov a metódy všeobecnými identifikátormi, napr. `$M1$, $M2$, $V1$, $T1$, $T2$`. Tak dokážeme transformáciu použiť aj v iných súčiastkach zdrojového kódu. Na obrázku 15 uvádzame príklad identifikovanej transformácie vo všeobecnom tvare s operáciami *presunutia, upravenia* a *vloženia*. Použitie transformácie následne tkvie z automatickej identifikácie miest v zdrojovom kóde, ktoré zodpovedajú abstraktnému syntaktickému stromu úpravy a jej aplikácie. Všeobecné symboly sa zamenia podľa identifikovaného miesta, napr. `$T2$ $v2$ = null` na `ILaunchConfiguration cfg = null`.

Sydit ponúkne programátorovi miesta, kde sa môže vzor transformácie uplatniť a ten ich po zvážení použije. Programátor tak nemusí identifikovať miesta pre transformáciu sám a transformácia sa vykoná automaticky pomocou vzoru, čo mu značne uľahčuje prácu a potrebu poznať miesta, kde je vzor možné uplatniť.



Obrázok 11.4 Abstraktný vzor transformácie zdrojového kódu podľa kódu v ukážkach 2 a 3, obrázok vychádza z [3].

11.5 Zhrnutie

Vykonávanie systematických zmien je častokrát málo zaujímavá činnosť a náchylná na chyby. Ako riešenie tejto problematiky sme uviedli zavedenie systémov pre odporúčanie transformácií zdrojového kódu. Transformácia sa identifikuje z interakcií programátora alebo vykonaných zmien medzi verziami zdrojového kódu ako prevod z pôvodnej verzie na novú verziu. Túto transformáciu je možné zovšeobecniť, ako sme uviedli na príklade systému Sydit, a následne aplikovať aj v iných podobných miestach zdrojového kódu. Odporúčaním v tomto prípade bol vzor transformácie a miesta, kde ju možno aplikovať. Na odporúčacie systémy v tejto oblasti sa preto môžeme pozerať z viacerých pohľadov po vykonaní prvej zmeny, ktorá by mala byť transformáciou:

- odporúčanie miest v zdrojovom kóde, ktoré by mal programátor transformovať,
- odporúčanie vzoru transformácie z vykonanej zmeny,
- odporúčanie použitia transformácie na zvolenom mieste.

Existujúce riešenia tejto problematiky však narážajú na viaceré problémy. Najvážnejším sú granularita a kontext zmeny. Uvedený príklad transformácie systémom Sydit je možné vykonať len v rámci metódy v zdrojovom kóde. Systematické zmeny sa však častokrát vykonávajú aj na vyššej úrovni, napr. hierarchia tried, alebo naprieč viacerými metódami súčasne. Problémom odlíšenia kontextu zmeny je príklad upravenia cyklu `for`, kedy by sa nesprávne odporúčali všetky podobné cykly v zdrojovom kóde. Z tohto vyplýva aj problém presnosti odporúčania, keď môžeme programátora zahltiť odporúčaniami.

Aj napriek potrebe ďalšej práce na odporúčacích systémoch transformácií zdrojového kódu pre systematické zmeny stúpa ich popularita. Systém Sydit, ktorý sme uviedli v tejto kapitole, naznačuje potenciál vo vhodnosti jeho použitia.

Literatúra

[1] Bieliková, M.: Softvérové inžinierstvo: Princípy a manažment. FEI STU Bratislava, Vydavateľstvo STU, (2000).

[2] Eaddy, M., Zimmermann, T., Sherwood, K.D et al.: Do crosscutting concerns cause defects? In: IEEE T. Software Eng. 34(4). IEEE CS Press, (2008), pp. 497-515.

[3] Kim, M., Meng, N.: Recommending Program Transformations. In: Recommendation Systems in Software Engineering (Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T., Eds.), Chapter 16, Springer Berlin Heidelberg, (2014), pp. 421-453.

[4] Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: Proc. of the 22nd ACM/IEEE International Conference on Software Engineering (ICSE ’09). ACM, (2009), pp. 309-319.

[5] Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE ’12). ACM, (2012).

[6] Meng, N., Kim, M., McKinley, K.S.: Sydit: creating and applying a program transformation from an example. In: Proc. of the 13th European Software Engineering Conference and the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE ’11). ACM, (2011), pp. 440-443.

[7] Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Eng. 38(1). IEEE CS Press, (2011), pp. 5-18.

[8] Nguyen, T.T., Nguyen, H.A., Pham, N.H., et al.: Recurring bug fixes in object-oriented programs. In: Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE ’10). ACM, (2010), pp. 315–324.

[9] Weißgerber, P., Diehl, S.: Are refactorings less error-prone than other changes? In: Proc. of the 2006 International Workshop on Mining Software Repositories (MSR ’06). ACM, (2006), pp. 112-118.

12 Odporúčanie vo fáze analýzy požiadaviek

Vo fáze analýzy požiadaviek vznikajú rôzne predstavy a ciele o produkte viacerých účastníkov, ktorí, ešte navyše, nemusia byť vždy dostupní geograficky. Preto vzájomná dohoda môže byť problematická a prináša rôzne výzvy. Riešením sa môžu zdať online fóra, avšak tie trpia viac problémami, ako sa zdá. Jedným z príkladov môže byť nízka efektivita dohadovania sa na fóre ústiaca do neschopnosti účastníkov sa dohodnúť. Odporúčaním sa však dajú zmierniť rôzne, aj tieto problémy, napr. *odporúčaním účastníkov–expertov, tém diskusii, či dokonca funkcií podľa podobného produktu. Dohadovanie potom naberá iný rozmer; systém prispieva k formovaniu produktu. Tento článok diskutuje a opisuje postupy, existujúce systémy a algoritmy pre odporúčanie vo fáze analýzy požiadaviek.*

Formovanie softvérového produktu je zložitý proces, v ktorom pôsobí mnoho ľudí. Za cieľ si kladú explicitnú identifikáciu potrieb, požiadaviek a túžob účastníkov. Práve tu je dôležité si uvedomiť, že títo účastníci zväčša orientovaní na podnikanie nemajú technické pozadie a preto sú potrební aj ďalší účastníci–práve tí s technickým pozadím. V súčte je to potom týchto mnoho ľudí, kde každý je orientovaný na inú problematiku, ktorí sa spoločne podieľajú na formovaní analýzy požiadaviek a teda identifikáciu potrieb, požiadaviek a túžob.

Všetci títo ľudia sa musia dohodnúť a často sa nenachádzajú na rovnakom mieste, alebo majú rovnaký časový priestor. Preto systém vo fáze analýzy požiadaviek musí umožniť formovať požiadavky v iteráciách a asynchrónne. Okrem spomenutého, každý účastník môže mať iné predstavy a ešte navyše do toho vstupuje expert, ktorý môže vo veľkej miere ovplyvniť rozhodovanie.

Tu sa naskytuje otázka, ako riešiť tieto problémy a ako ich pozitívne ovplyvniť. Asynchrónny charakter majú napr. fóra, kedy komunikácia nie je okamžitá, čo môže rapídne znížiť produktivitu oproti osobnému kontaktu. Treba sa teda pozrieť, ako môžeme zlepšiť procesy v rámci fázy analýzy požiadaviek vývoja softvéru.

Vhodným odporúčaním a odporúčaním na správnych miestach môžeme podporiť takéto fórum, kde účastníci môžu produktívnejšie a rýchlejšie spolupracovať. Ako sme naznačili, vysoká variabilita vedomostí účastníkov hovorí o tom, že každý môže prispievať k rôznym témam. Odporučením účastníkov – expertov vieme dať do pozornosti tých správnych ľudí. Odporučiť vieme aj témy diskusii pre požiadavku alebo podobné funkcie podľa produktu.

Otázne však ostáva, či tento prístup nie je viac negatívny ako pozitívny, nakoľko podnikateľské prostredie ovládané strojom nemusí znamenať vždy úspech na trhu, najmä keď nejde o inovácie a teda monetizácia môže týmto utrpieť značný zásah v podobe rovnako–potom už nezaujímavých služieb. Produkt už viac nebude jedinečný, ale skopírovaný.

Táto práca sa venuje odporúčaniu vo fázy vývoja softvéru analýzy požiadaviek. Opisuje a diskutuje odporúčacie systémy, ich postupy, procesy a porovnania.

12.1 Analýza požiadaviek a odporúčanie

Ako sme už poukázali v úvode, analýza požiadaviek je explicitná identifikácia potrieb, požiadaviek a túžob účastníkov. Pričom vznikajú nasledovné problémy [1]:

- účastníci sú geograficky distribuovaní,
- fyzické stretnutie je problematické a často krát nie je možné,
- účastníci majú rôzne predstavy a ciele.

Riešením môžu byť online systémy, ktoré preberajú prvky služieb sociálnych sietí. Problém týchto fór je v tom, že si vyžaduje veľa úsilia. Príklad môže byt vývoj open source softvéru, ktorý je častokrát pomalý vo svojej podstate. Avšak, práve v týchto online fórach môžeme pridať odporúčania, ktoré zrýchlia prácu účastníkom.

Príklady takých fór pre zachytávanie požiadaviek môžu byť tieto systémy:

- wiki stránky – MediaWiki,
- projekt manažment systémy SCRUM – JIRA,
- nástroje na zachytávanie chýb v softvéri – Redmine, Mantis, Bugzilla.

Každý jeden z týchto programov môže vo fáze analýzy požiadaviek zachytiť požiadavku a je možné ju v ňom komunikovať, napr. vytvorením wiki stránky alebo pridaním požiadavky na novú funkcionalitu alebo pridaním novej chyby v systéme.

Odporúčať môžeme pri pridávaní príspevku, napr. relevantnej témy. Príklad je portál Stackoverflow (stackoverflow.com), kde po zadaní otázky alebo príspevku sa zobrazia podobné príspevky a používateľ sa môže pozrieť, či už takáto otázka nebola už niekedy zodpovedaná. Tiež odporúčanie účastníkov (expertov) na danú tému a odporúčanie nových funkcii systému podľa rovnakej domény cez asociačné pravidlá.

Avšak, problémy online systémov – diskusných fór sú tieto:

- mnoho malých diskusných tém [2],
- redundantné témy,
- degenerácia do Q&A fór,
- veľké množstvo príspevkov a relevantná informácia je skrytá.

Fórum sa častokrát ocitne v situácii, kedy je roztrúsené do mnohých malých diskusných tém. Informácie sú potom fragmentované a je potrebné vynaložiť veľké úsilie a hľadať informáciu, ktorá je pre nás relevantná.

Potom týchto mnoho malých diskusných tém vyúsťuje v mnoho redundantných tém, čo značne robí fórum neprehľadné, no dokonca môže ísť aj o rozdielne odpovede a záveri a môže teda medzi účastníkmi nastať rozpor. Ďalším problémom, ktorý súvisí z predchádzajúcimi dvoma je degenerácia do takzvaných „otázka-odpoveď“ fór. V takýchto fórach sa nerieši problém, ale iba sa odpovedá na otázky.

Ako sme vyššie spomenuli pri portáli Stackoverflow, pri zadaní otázky sa zobrazia podobné otázky, čím eliminujeme redundanciu. Podobný problém, ale v inej úrovni, sa nachádza vo veľa príspevkoch. Tento problém je často viditeľný napr. vo fóre XDA-developers, kde mnoho ľudí diskutuje, avšak relevantná informácia je skrytá.

Ďalším explicitným, nie automatickým, riešením na redundantné príspevky môže byť obohatenie systému o funkciou označenia fóra ako redundantného. Potom, po vyriešení sa dostaneme priamo k odpovedi a nemusíme zaťažovať s otázkou ďalej účastníkov.

Príklad procesu odporúčania v online-fórach môže byť proces OPCI (anglicky Organizer and promoter of collaborative ideas). Je to nástroj na online diskusiu [1]. Jeho fungovanie je takéto:

1. Účastník vloží požiadavku na funkciu.
2. Systém nájde podobné požiadavky.
3. Požiadavka je zaradená do skupiny.
4. Odporúčací systém vygeneruje potencionálne témy.
5. Účastníci kolaborujú a transformujú požiadavku ďalej.
6. Systém sleduje požiadavky a navrhuje ich rozdelenie do nových témy.

Teda, ako sme už poukázali, ide o odporúčanie podobných požiadaviek a potenciálnych tém pre tieto požiadavky. Tieto témy a ich odporúčanie je založené na obsahu alebo na kolaborácii alebo na účastníkov a ich tém podobných s týmito témami.

12.2 Odporúčanie v online fórach

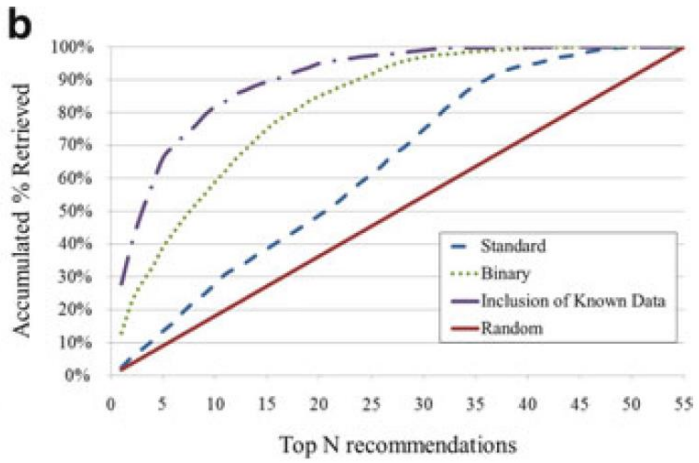
Veľmi dôležitým je vedieť pre koho odporúčame, aby sme vedeli odporučiť personalizovane. Nemá zmysel odporúčať programátorovi podnikateľsky orientovanú požiadavku, programátorovi by mala byť odporučená požiadavka, ku ktorej môže povedať čas realizácie nejakého technického a implementačného detailu. Preto vytváranie a poznanie profilu používateľa je veľmi dôležité.

Menzies a kol. reprezentovali profil používateľa, kde každý vstup používateľa predstavoval jeho záujem [5]. Vytvorili maticu, kde na osiach boli používatelia k témam. Takto môžeme reprezentovať profil používateľa k témam.

Ďalej profil používateľa môže byť zachytený na základe váh; váhy podľa počtu príspevkov a podľa obsahu príspevku a to tak, že meriame podobnosť medzi príspevkami a hlavným–prvým príspevkom [1]. Teda, účastníci, ktorí sa vyjadrujú k podnikateľsky orientovanej téme zrejme nebudú mať záujem o tému technickú. Ďalším vhodným ukazovateľom na vytvorenie profilu používateľa sú roly projektu, kde môžeme vytvárať susedov, napr. programátor bude mať zrejme bližšie k téme technickej ako podnikateľsky orientovanej.

Bavota a kol. vyhodnotili profil používateľov na štyroch množinách dát použitím metód – náhodná, štandardná, binárna a inklúzia známych dát. Množiny boli nasledujúcich veľkostí:

- 50 tém, 3392 príspevkov, 2120 používateľov,
- 29 tém, 223 príspevkov, 36 používateľov,
- 60 tém, 885 príspevkov, 523 používateľov,
- 55 tém, 1652 príspevkov, 132 používateľov.



Obrázok 12.1 Bavota a kol. – vytváranie profilu používateľa [1].

Výsledky sa dajú zobraziť krivkou, kde na osi x bol počet vyžiadaných odporúčaní a na osi y bol počet správne odporučených – Obrázok 12.1. Pri náhodnej metóde náhodne vyberali, čo daného účastníka môže zaujímať. Pri štandardnej metóde išlo o maticu (každý vstup používateľa predstavoval jeho záujem), ako sme už uviedli vyššie. Pri binárnej metóde bolo pridané váhovanie (váhy podľa počtu príspevkov a podľa obsahu príspevku). Nakoniec, inklúzia známych dát znamená pridanie známych informácii o používateľoch (rolí používateľov a vytvorenie susedov).

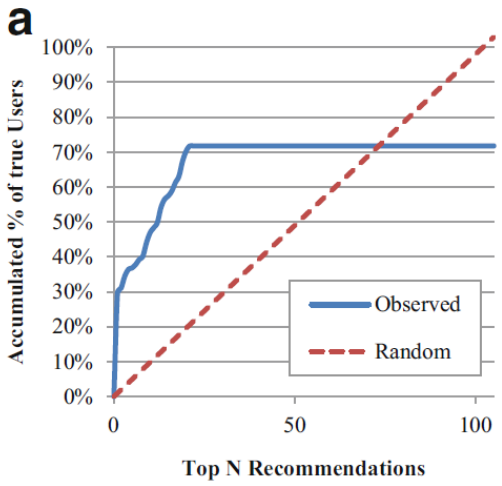
12.3 Odporúčanie účastníkov a funkcii

Veľké množstvo príspevkov nikdy nedostane odpoveď a aj keď dostane, nikdy nevieme, či táto odpoveď je relevantná a pochádza od kvalifikovaného zdroja. Vzniká tu tak priestor pre odporúčanie účastníkov expertov. Účastníkov môžeme rozdeliť do 3 skupín [1]:

- *priami účastníci* – priamo prispeli k téme,
- *nepriami účastníci* – podobnosť na základe príspevkov do rovnakých tém,
- *odvodení účastníci* – preukázali vzory záujmu, ktoré naznačujú záujem o danú tému.

Takáto klasifikácia účastníkov a stupne podobných účastníkov môžu byť použité v odporúčacom systéme. Prirodzene priami účastníci majú k sebe najbližšie a takíto susedia sú k sebe najbližšie. Potom nepriami účastníci, keď pre podobnú problematiku prispeli svojim príspevkom. A nakoniec odvodení účastníci, keď preukázali vzory záujmu, napr. ak niekoho zaujímajú počítače, tak veľká pravdepodobnosť, že ho bude zaujímať aj softvér.

Bavota a kol. vykonali experiment [1], kde vymazali príspevky a hľadali podobných účastníkov priamych, nepriamych a odvodených. Ich hybridný systém vybral už pri 25 vyžiadaných odporúčaní 70% podobných používateľov k danej téme. Obrázok 12.2 znázorňuje tieto výsledky.



Obrázok 12.2 Experiment vymazanie príspevkov a hľadanie účastníkov, hybridná metóda [1]

Odporúčanie funkcii realizujeme v procese doménovej analýzy, teda musíme byť v rovnakej doméne a hľadáme v dostupných informáciách. Teda hneď sa vyskytuje problém, že musí existovať analýza požiadaviek, ktorá často krát nie je dostupná pre danú doménu. Tento problém sa volá problém studeného štartu. Nemáme dostatočné množstvo dát, aby sme odporučili funkciu alebo nemáme žiadne. Preto potrebujeme predbežné dáta, z ktorých môžeme dolovať asociačné pravidlá [1].

Celý proces odporúčania funkcií pozostáva z krokov:

- *trénovanie* – získavanie opisu požiadaviek, zoradenie požiadaviek do skupín, vytvorenie modelu (grafu alebo matice),
- *používanie* – spätná väzba od používateľov.

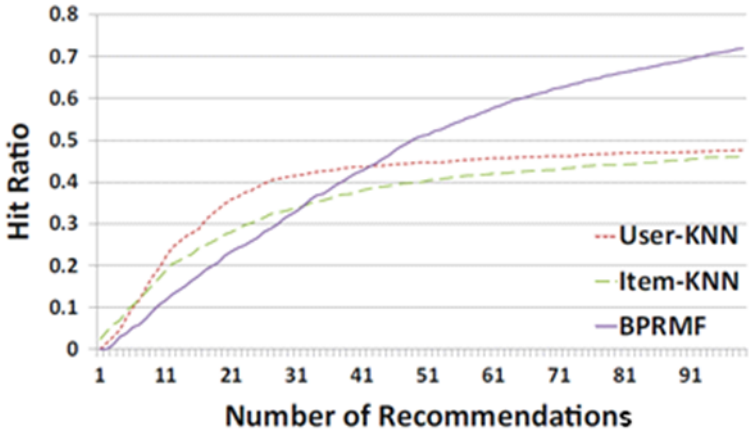
Ako sme už poukázali, ako prvé musíme mať dáta, na základe ktorých môžeme odporučiť danú funkciu. Štúdiom materiálov, napr. dokumentácie alebo opisu požiadaviek, získame dáta. Tieto dáta zoradíme do skupín a vytvoríme model, ktorá funkcia je podobná ktorej.

Ako druhý krok v celkovom procese používame odporúčací systém a získavame spätnú väzbu od používateľov, na základe čoho môžeme ďalej zlepšovať odporúčací systém.

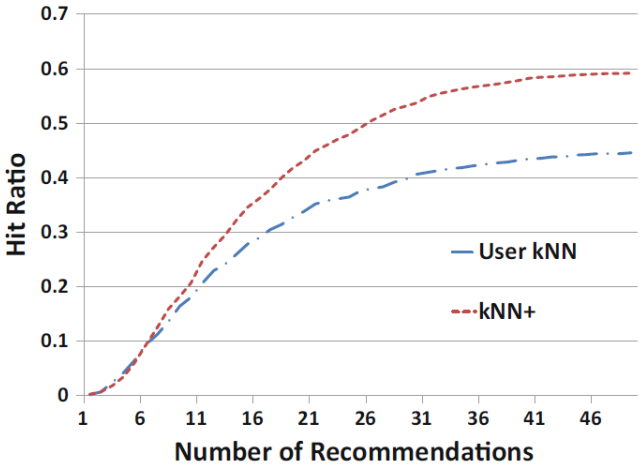
Dolovanie funkcii zo špecifikácií je vlastne iniciálne získanie dát. Potrebujeme predspracovať vety, napr. zmazaním slov. Potom vety sú vektory a môžeme použiť jeden z klastrovacích algoritmov na dolovanie, napr. K-means, Kmedoid [3] alebo sférický K-means [4]. Príkladom je, keď z algoritmov vyjdú kľúčové slová zo špecifikácie, napr. „updat“, „databas“, „automat“, „viru“. Potom názov funkcie môže byť *„Virus definition update and automatic update supported“*.

Ricci a kol. realizovali porovnanie algoritmov odporúčacích systémov (Fivefold cross-validation experiment) [1]. Obrázok 12.3 uvádza výsledky trénovania domény. Vytvorili maticu produktov k funkciám a počítali odporúčacie skóre podľa daného algoritmu. Pre každý produkt boli vybrané 3 funkcie náhodne, ktoré reprezentovali profil produktu. Ďalšia jedna bola cieľová funkcia. Experiment sa zaoberal týmito algoritmami:

- *BPRMF* – dekompozícia matice (dobré výsledky pre N väčšie ako 45),
- kNN podľa používateľov – susedia účastníci,
- kNN podľa položiek – susedia funkcie.



Obrázok 12.3 Porovnanie algoritmov odporúčacích systémov [1]



Comparison of hit ratio for the hybrid method and the user-based kNN

Obrázok 12.4 Porovnanie algoritmov odporúčacích systémov – kNN+ [1]

Z obrázku môžeme vidieť, že odporúčanie podľa účastníkov bolo úspešnejšie ako odporúčanie podľa funkcie. Obrázok 12.4 uvádza ešte ďalší algoritmus kNN+, ktorý má ešte lepšie výsledky odporúčanie podľa účastníkov a funkcii.

12.4 Zhrnutie

Venovali sme sa odporúčaniu vo fáze softvérového vývoja analýzy požiadaviek. Ukázali sme, že existuje mnoho systémov, ktoré môžu byť použité v tejto fáze. Na týchto systémoch je potom možné odporúčanie účastníkov a funkcií. Spomenuli sme problémy zvyčajných fór, pričom odporúčanie riešilo niektoré z nich, napr. odporučenie podobnej témy znižuje počet redundantných príspevkov vo fóre. Diskutovali sme tiež o postupoch, pomocou ktorých môžeme odporúčať.

Ukázali sme rôzne zdroje dát, pomocou ktorých môžeme urobiť odporúčania. Avšak, stále je prítomný problém studeného štartu z ktorých vyvodíme odporúčanie. Tiež musia byť prítomné predbežné dáta, na základe nich potom algoritmami vydolujeme jednotlivé odporúčania. Pomocou trénovania môžeme vylepšiť výsledky a nielen, ale aj s spätnou väzbou od používateľa.

Odporučením účastníkov expertov môžeme zasiahnuť a zrýchliť proces analýzy požiadaviek. Odporučením funkcií zas môžeme upozorniť na tie, ktoré sú zrejmé, no nemusia byť vždy úplne viditeľné a bez nich by softvér nebol úplný.

Ukázalo sa, že odporúčanie založené na profiloch používateľov a im podobných je presnejšie ako odporúčanie založené na funkciách. Hybridný prístup môže tiež odhaliť odporúčania, ktoré jednotlivo prístupy neodhalia.

Literatúra

[1] Bavota G. et al.: Recommendation Systems in Requirements Discovery. In: Recommendation Systems in Software Engineering, pp. 455-476. doi: 10.1007/978-3-642-45135-5

[2] Cleland-Huang, J., Dumitru, H., Duan, C., Castro-Herrera, C.: Automated support for managing feature requests in open forums. Commun. ACM 52(10), 68–74 (2009). doi:10.1145/1562764.1562784

[3] Dhillon, I.S., Modha, D.S.: Concept decompositions for large sparse text data using clustering. Mach. Learn. 42(1–2), 143–175 (2001). doi:10.1023/A:1007612920971

[4] Manning, C.D., Raghavan, P., Schutze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008).

[5] Menzies, T.: Data mining: a tutorial. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering. Springer, Heidelberg, Chap. 3 (2014).

13 Odporúčacie systémy pre manažment problémov

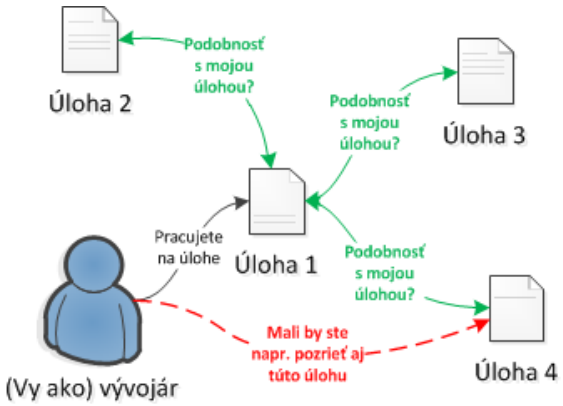
Počas vývoju softvéru hrá efektívne rozdelenie úloh významnú rolu. Vývojárom na to slúžia systémy pre správu úloh. Okrem úloh obsahujú aj takzvaný repozitár, čiže miesto pre ukladanie zdrojových kódov a úlohy s nimi môžu byť prepojené. Rozdelenie úloh sa môže stať náročným v prípade ich veľkého počtu. Z dôvodu veľkého počtu záznamov úloh, sa môžu nováčikovia ťažšie adaptovať v projekte. Podobne ťažké rozhodovanie nastáva aj v prípade rozhodovania o duplicitných úlohách, kedy sa pátra po predchádzajúcich podobnostiach a rozhoduje či sa nová úloha je alebo nie je duplicitná. Táto kapitola prezentuje, ako možno vyriešiť problémy spojené s systémami pre manažment problémov.

Okolo systému pre manažment problémov sa sústreďujú všetci účastníci vývoja softvéru, nielen v rámci lokálnej budovy, ale aj globálne. Okrem úloh sú zapísané aj požiadavky na zmeny, chybové správy a návrhy na smerovanie projektu. V projekte s veľkou rozsiahlosťou môže byť v repozitári aj viac ako tisíc položiek. V takomto prípade môžu nastať problémy s ohľadom na strategické smerovaním projektu.

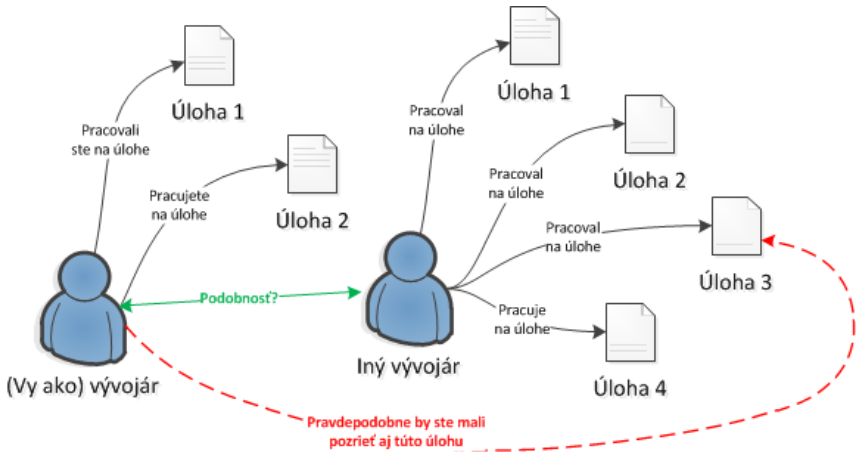
Odporúčacie systémy softvérového inžinierstva môžu podporiť manažment a rozhodovanie o úlohách. Odporúčacie systémy využívajú dva základné prístupy a to:

- *Filtrovanie na základe obsahu* – generovanie odporúčania na základe podobných vlastností aktuálneho dokumentu (Obrázok 13.1). Každá úloha je reprezentovaná určitými vlastnosťami, ktoré v minulosti boli súčasťou ich textového opisu. Vlastnosťou úlohy môže byť záväznosť, dátum vytvorenia, zodpovedný vývojár, ovplyvnené zdrojové kódy atď.

- *Kolaboratívne filtrovanie* – generovanie odporúčania na základe podobných preferencií ostatných používateľov (Obrázok 13.2). Každému používateľovi, napr. vývojárovi, sa na základe interakcií so súbormi vytvára profil. Keď sa k používateľovi nájde podobný užívateľ, tak sa mu ponúknu úlohy, s ktorými interagoval tento druhý používateľ. Profilová podobnosť môže byť založená buď na predchádzajúcej interakcií s úlohou alebo vlastnosťami ako napr. tímu, role, lokácie atď.



Obrázok 13.1 Filtrovanie na základe obsahu. Odpoveď na otázku: ktoré ostatné úlohy sú podobné mojej?



Obrázok 13.2 Kolaboratívne filtrovanie. Odpoveď na otázku: na ktorých podobných úlohách pracujú kolegovia vzhľadom k mojej úlohe?

13.1 Riešenie problematiky triedenia úloh za pomoci odporúčania

Počas analýzy a triedenia nových úloh sa vynárajú otázky typu: bolo to už predtým nahlásene? Mal by sa tento problém opraviť? Kedy a kto by tento problém mal opraviť? V malých projektoch si môže vývojár udržať nadhľad nad aktuálne vyvíjaným softvérom a podobné problémy vedieť vyriešiť, adresovať a ohodnotiť. Pri veľkých projektoch sa triedenie úloh stáva problémom, pri ktorom treba vynaložiť veľké úsilie pre správne riešenie.

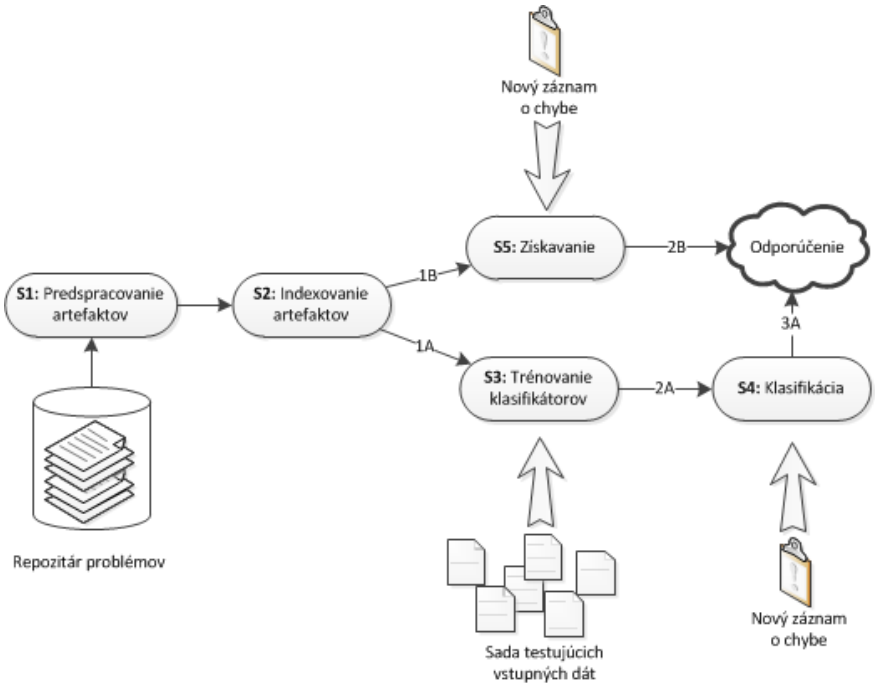
13.1.1 Sú duplikáty zaťažujúce alebo prínosné?

V údržbovej časti životného cyklu projektu sa väčšina činností sústreďuje okolo repozitára problémov. V prípade nájdenia existencie už podobného problému, môže nový problém lepšie opisovať situáciu a môže byť označený ako duplicitný, hoci aj keď neprináša nové informácie. S problémom neúmerného množstva duplicít je možné stretnúť sa v „open source“ projektoch, kde je veľké množstvo vytvorených problémov (za rok aj viac ako 300). Napr. v projekte Eclipse z celkového počtu nahlásených chýb je až 13% duplicitných a v projekte Mozilla Firefox až 30% [1]. Podobný fenomén je možné vidieť napr. aj v projekte Sony Ericsson Mobile Communications, kde miera duplicít bola na úrovni 10% [2]. Z tohto dôvodu odporúčanie detekcií duplicít môže výrazne zlepšiť adresovateľnosť problémov a zminimalizovať počet duplicít.

13.1.2 Systém pre detekciu duplikátov

Predtým ako býva nový záznam o chybe uložený do systému pre manažment problémov, je vhodné preveriť jeho podobnosť voči existujúcim záznamom v repozitári problémov. Pri rozhodovaní o duplicite záznamov o chybách môžeme rozhodovanie prirovnať k odpovedania na otázku „máme tento záznam o chybe a ktoré záznamy o chybách sa k môjmu záznamu najviac javia ako duplicitné?“ (Obrázok 13.1). Najvhodnejšou technikou pre detekciu duplicít je *filtrovanie na základe obsahu*. Pri samotnom porovnávaní textových popisov problémov medzi staršími a novšími dátami, je možné rozdeliť problematiku na:

- *problém získavania informácií* (stav S5),
- *klasifikačný problém* (stav S4).



Obrázok 13.4. Pohľad na odporúčacie systémy pre detekciu duplicít

Oba prístupy zdieľajú prvé dva kroky cesty *S1->S2*. Vo väčšine prípadov pod pojmom predspracovanie (stav S1) je myslená procedúra zahŕňajúca kroky:

- *normalizácia textu* – prevedenie všetkých písmen textu na malé, odstránenie špeciálnych znakov, skrátenie bielych znakov (medzier) na maximálne jednu medzera medzi výrazmi.
- *vylúčenie nepotrebných slov* – odstránenie slov, ktoré sa vyskytujú v texte často, avšak nemajú žiadnu výpovednú informáciu, napr. v slovenčine to môžu byť spojky, predložky, zámena, niektoré slovesá a častice,
- *stemovanie* – úprava slov na ich základný tvar, nazývaný aj koreň, napr. používateľ hľadá slovo „operátormi“, vyhľadávanie však prebieha so slovom „operátor“.

Výstupom z predchádzajúceho kroku býva zoznam neusporiadanej kolekcie slov pre dokument. Tieto kolekcie sa následne zindexujú (stav S2) a vytvorí sa pre ne matica, koľkokrát sa daný výraz nachádza v ktorých dokumentoch (ako zobrazuje Tabuľka 13.1).

Tabuľka 13.1Výraz dokument matica.

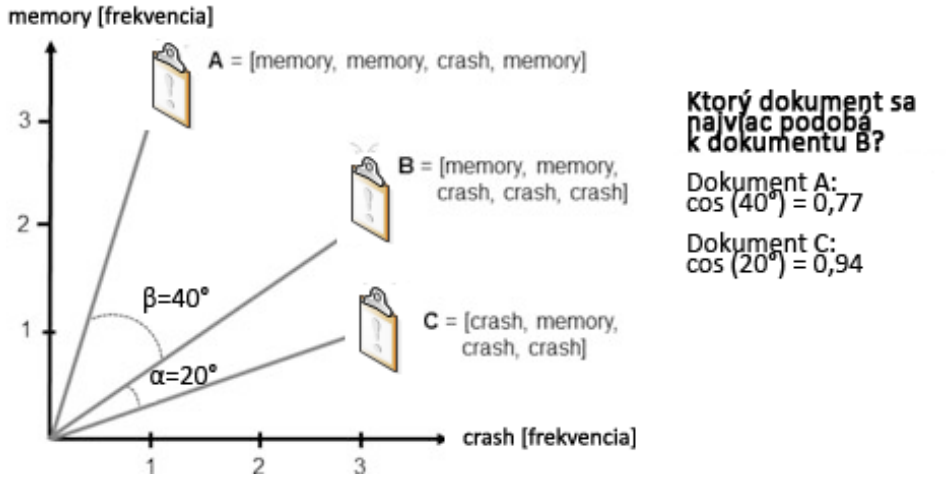
Výraz	Celková frekvencia	Dokument číslo/frekvencia
Memmory	6x	A -> 3x B -> 2x C -> 1x
Crash	7x	A -> 1x B -> 3x C -> 3x
...		

13.1.3 Detekcia duplikátov z pohľadu problematiky získavania informácií

Pre proces rozhodovania o duplikátoch pomocou získavania informácií (cesta *1B->S5->2B*) je dôležité analyzovať čo možno najväčšie možné množstvo dát pre presnejšiu detekciu.

Do procesu rozhodovania o duplikátoch pomocou získavania informácií (stav S5) nám prichádzajú informácie v podobe matice s výskytom výrazov v dokumentoch (Tabuľka 13.1). Jednotlivé údaje z matice je možné interpretovať ako vektory dokumentov a množinu dokumentov interpretovať ako vektorový priestor čím dostaneme model, ktorý ilustruje Obrázok 13.3. Pomocou kosínusovej miery je možné veľmi jednoducho zistiť podobnosti medzi dokumentmi, kde vstupným údajom je uhol, ktorý zvierajú vektory dokumentov a výstupom je hodnota určujúca zhodu v intervale 0 až 1 [3].

Keď príde nový záznam o chybe, musí byť pre neho vykonaný rovnaký postup transformácie ako s už existujúcimi údajmi v repozitári (cesta *S1->S2->1B*), aby vo výsledku mohol byť tento nový záznam o chybe porovnaný kosínusovou mierou na duplicitu (stav S5).



Obrázok 13.3 Vektorový priestorový model (angl. Vector Space Model) a výpočet zhodnosti kosínusovou mierou.

13.1.4 Detekcia duplikátov z pohľadu klasifikačného problému

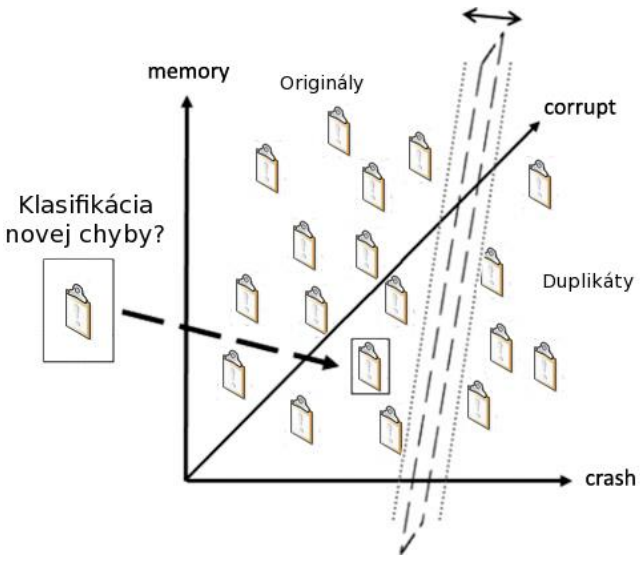
Zatiaľ čo v predchádzajúcej sekcií 13.1.3 boli pre rozhodovanie o duplicite použité iba aktuálne dostupné informácie, pri rozhodovaní o duplicite je možné použiť aj strojové učenie (cesta *1A->S3->2A->S4->3A*).

Pre toto riešenie je potrebné vytvoriť takzvaný klasifikátor (stav S4), ktorý by dokázal priradiť[24] prijatý záznam o chybe do správnej triedy, čiže či je duplicitný alebo originálny. Pre vytrénovanie klasifikátora je vhodné použiť metódu učenia sa s učiteľom, kedy učenie klasifikátora prebieha na sade testovacích vstupných dát (duplicitné a originálne) ktoré majú priradenými správnymi výsledkami [4].

Pri učení sa s učiteľom sa na rozhodovanie kedy sa jedná o duplikát obvykle aplikuje metóda Support Vector Machines – SVM (Obrázok 13.4), ktorá je podobná VSM (Obrázok 13.3). V tomto prípade model obsahuje záznamy o chybách v priestore vo forme vektorov, kde každý výraz môže predstavovať dimenziu (v našom ilustračnom prípade máme tri výrazy). SVM následne hľadá nadrovinu, ktorá by mala vhodne oddeľovať sadu testovacích vstupných dát na dve triedy. Metóda sa okolo tejto nadroviny sa snaží vytvoriť na obidve strany maximálny odstup, aby rozhodovanie bolo čo najjednoznačnejšie.

Podobne ako v predchádzajúcej sekcií 13.1.3, po príchode nového záznamu o chybe, musí byť pre neho vykonaný rovnaký postup transformácie ako s už existujúcimi údajmi v repozitári (cesta *S1->S2* a preskočiť až k stavu S4) , aby vo výsledku mohol byť tento nový záznam klasifikovaný buď ako duplicitný, alebo originálny (stav S4).

[24] Inak povedané klasifikovať. Od toho je odvodený aj názov klasifikátor.



Obrázok 13.4 Metóda podporných vektorov (angl. Support Vector Machines, SVM). Optimálna nadrovina je zobrazená dlhými pomlčkami, pričom maximálny odstup je zobrazený bodkovanou (a obojsmernou šípkou).

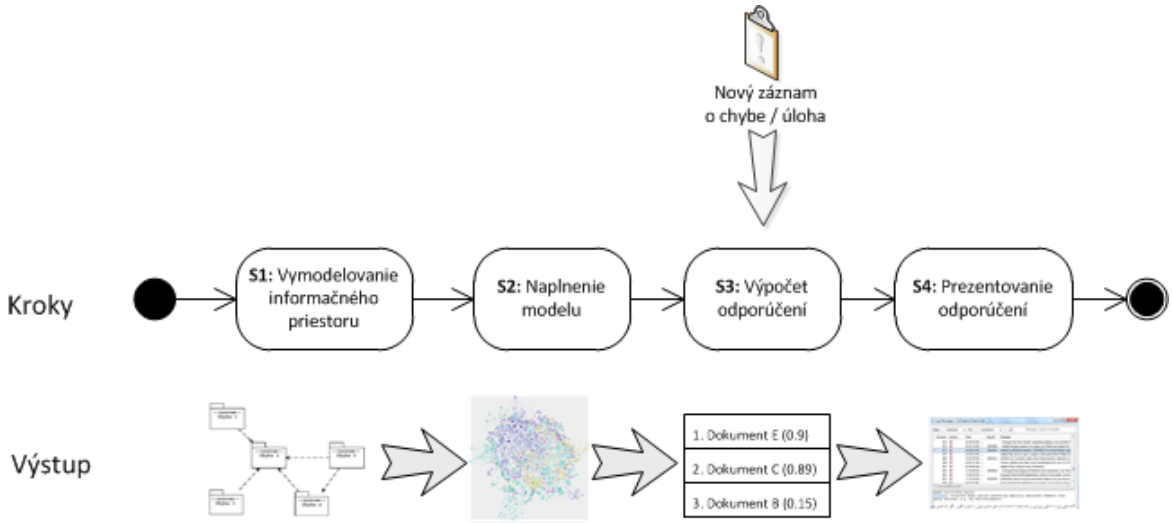
13.2 Navigovanie v záznamoch o chybách počas vývoja softvéru

Ako bolo spomenuté v úvode, systém pre manažment problémov býva využívaný vo veľkom rozsahu vývojármi softvéru a to nielen v rámci lokálnej budovy, ale aj globálne. Málokedy v systéme pre manažment problémov býva iba zopár projektov, obvykle ich bývajú desiatky až stovky. Medzi klasickými problémami, ktoré nastávajú pri spravovaní a práci s takýmto systémom býva:

- *zahltenie informáciami* – človek stráca prehľad o kontexte z dôvodu prepínania sa pri práci s informáciami medzi rôznymi zdrojmi projektov,
- *problém nováčika* – nový člen vývojárskeho tímu si pri potrebe naštudovania problematiky z predchádzajúcich projektov na začiatku štúdovania uvedomí, že nevie kde má začať hľadať. Pod veľkým návalom informácií sa ľahko pridá aj predchádzajúci menovaný problém. V tomto prípade vlastnosť zvaná prehľadnosť systému je kľúčovou pre ľahšie orientovanie sa v kontexte informácií.

K druhému problému sa ako najvhodnejším alternatívnym riešením javí vedenie nováčika za pomoci skúsenejšieho člena tímu, ktorý vie správne nasmerovať. Avšak nie vždy sú vyhovujúce podmienky na zrealizovanie tohto typ postupu.

Vo všeobecnosti oba menované problémy je možné zmierniť alebo úplne odstrániť vďaka odporúčacím systémom [5], ktoré je možné zakomponovať do už existujúcich projektov. Obrázok 13.5 ilustruje príklad štvorstavového postupu.



Obrázok 13.5 Príklad štvorstavového modelu odporúčacieho systému pre podporu navigovania.

Prvým krokom býva analýza repozitárov a ostatných zdrojov informácií, ktorej výstupom je model toku informácií (Obrázok 13.5, stav S1). Ďalším krokom je naplnenie tohto modelu všetkými (aj historickými) dátami z repozitára (stav S2). Medzi jednotlivými záznamami sa vypočíta podobnosť o chybách na základe aplikovania princípov, ktoré sme si ukázali a uviedli v predchádzajúcich kapitolách (stav S3). V tomto kroku po vypočítaní korelácie zhodností medzi historickými dátami môže prísť do modelu aj nový záznam o chybe/úlohe pre ktorý sa pre ktorý sa nájdu podobné záznamy o chybách/úlohách prostredníctvom porovnania s vypočítaným modelom. Tieto nájdené a zoradené podobnosti sa následne prezentujú ako odporúčania (stav S4).

13.3 Zhrnutie

Systémy pre manažment úloh bývajú centrom, kde sa sústreďujú všetky informácie súvisiace s projektmi a preto akékoľvek zlepšenie pre ich spravovanie a používanie býva veľkým prínosom. Ako sme si ukázali, za pomoci odporúčacích systémov je možné zlepšiť problematiku triedenia úloh, kde sme si predstavili konkrétne dva možné spôsoby riešenia duplicít, ktoré je možné použiť. Ukázali sme si aj postupy a podmienky pre možnú implementáciu do už existujúcich systémov pre manažment úloh, napr. projekt Bugzilla alebo JIRA prostredníctvom rozšírení.

Literatúra

[1] Sureka, A., Jalote, P.: Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In Software Engineering Conference (APSEC), 17th Asia Pacific, pp. 366-374 (2010). doi: 10.1109/APSEC.2010.49

[2] Runeson, P., Alexandersson, M., Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing. In Software Engineering, ICSE 2007. 29th Interna-tional Conf., pp. 499-510.

[3] Laclavík, M., Šeleng, M.: Vyhľadávanie informácií. Slovenská technická univerzita v Bratislave, vo Vydavateľstve STU, Bratislava, Vazovova 5, 2012. ISBN: 978-80-227-3829-3.

[4] Dietterich, T. G.: Machine Learning. In Nature Encyclopedia of Cognitive Science, London: Macmillan, 2003.

[5] Cubranic, D., Murphy, G.C.: Hipikat: recommending pertinent software development arti-facts. In Software Engineering, 2003. Proc. 25th International Conference, pp. 408-418 (2003) doi: 10.1109/ICSE.2003.1201219

Index