Fakulta matematiky, fyziky a informatiky
Univerzita Komenského
Katedra informatiky

Diplomová práca

Boris Slávik

Bratislava 2005

Fakulta matematiky, fyziky a informatiky
Univerzita Komenského
Katedra informatiky

REFAKTOROVANIE AKO METÓDA ÚDRŽBY SOFTVÉRU – SÚČASNÝ STAV

Autor: Boris Slávik
Vedúci diplomovej práce: RNDr. Marián Vittek, PhD.

Bratislava, apríl 2005

Čestne prehlasujem, že diplomovú prácu som vypracoval samostatne s použitím uvedenej literatúry a elektronických dokumentov.

_____

Boris Slávik

Ďakujem svojmu diplomovému vedúcemu RNDr. Mariánovi Vittekovi, PhD. za cenné rady a pripomienky pri vypracovávaní diplomovej práce.

Obsah

1 Úvod

Jedným z hlavných pravidiel procesu vývoja softvéru je, že softvér sa musí neustále prispôsobovať meniacim sa podmienkam a požiadavkám. Dôsledkom týchto zmien je rastúca zložitosť softvéru. Jedným zo spôsobov, ako znížiť zložitosť a zlepšiť štruktúru softvéru, je technika pomenovaná refaktorovanie.

Refaktorovanie je proces, ktorý sa zameriava na evolúciu zdrojového kódu. Umožňuje nám zlepšiť dizajn kódu, spraviť ho viacej flexibilnejším a znovupoužiteľnejším pre ďalšie zmeny a pri tom zachovať správanie programu. Je to disciplinovaný spôsob, ako meniť zdrojový kód programu tak, aby sme doňho nezaniesli chyby. Najjednoduchším príkladom refaktorovania je premenovanie premennej.

Refaktorovanie znamená zmenu dizajnu programu potom, ako už bol implementovaný. Vo svojej podstate je to iteratívna metóda, čo spôsobuje, že nezapadá do klasických modelov vývoja softvéru, akým je napríklad vodopádový model. Presadzuje sa najmä v menej formálnych modeloch vývoja softvéru, ktoré sú založené práve na častých zmenách v dizajne kódu. Príkladom takéhoto modelu vývoja softvéru je extrémne programovanie, v ktorom technika refaktorovania hrá jednu z hlavných úloh pri zlepšovaní dizajnu programu. Refaktorovanie teda môže byť považované za alternatívu k metódam, ktoré sa snažia najskôr vytvoriť presný dizajn programu a až potom ho implementovať.

Dobrí programátori vždy upravovali zdrojový kód programov, aby zlepšili jeho štruktúru. Čiže v podstate refaktorovali softvér bez toho, aby o tom vedeli. Napriek tomu sa refaktorovanie stalo dôležitou súčasťou procesu vývoja softvéru až v posledných rokoch.

1.1 Ciele diplomovej práce

V tejto diplomovej práci sa pokúsim zhrnúť získané poznatky o technike refaktorovania a vytvoriť prehľad o výskume a pokroku v tejto oblasti. Cieľom je vytvoriť prvú slovenskú monografiu o refaktorovaní, ktorá má uviesť čitateľa do tejto problematiky.

V nasledujúcich dvoch kapitolách sa práca zameriava najmä na teoretické výsledky výskumu. Po prečítaní týchto kapitol by mal čitateľ pochopiť základné princípy a mechanizmy refaktorovania (druhá kapitola) a získať prehľad o problémoch zdrojového kódu (tretia kapitola). Štvrtá kapitola sa zaoberá hlavnými cieľmi a nezodpovedanými otázkami výskumu refaktorovania. Takisto sa tu spomínajú niektoré techniky a postupy, ktoré boli navrhnuté ako riešenia niektorých problémov. V ďalšej kapitole sa pozrieme na súčasný výskum a povieme si o niektorých aktuálne prebiehajúcich projektoch. V záverečnej, šiestej kapitole si najskôr povieme o dôležitých kritériách a vlastnostiach softvérových nástrojov na refaktorovanie a potom porovnáme niektoré momentálne dostupné.

1.2 Pohľad do histórie

Pojem refaktorovanie pochádza z matematiky, kde sa faktorizovaním upravujú matematické výrazy do inej formy. Uvažujme napríklad nasledovný výraz $(X+1)*(X-1)=X^2-1$. Aj keď je ľavá strana tejto rovnice dlhšia, je ľahšie pochopiteľná, lebo obsahuje jednoduchšie operátory. Navyše nám poskytuje viac informácií o štruktúre funkcie $f(X)=X^2-1$. Napríklad vidíme, že jej koreňmi sú čísla ±1.

V oblasti softvérového inžinierstva začali refaktorovanie používať ľudia, ktorí museli pracovať s neprehľadným a zložitým kódom programov. Podobne ako používanie softvérových dizajnových vzorov, bolo refaktorovanie prirodzene

používanou technikou, ktorá však musela byť najskôr presne pomenovaná a opísaná, aby sa dostala do povedomia verejnosti a získali z nej výhody aj ostatní programátori.

V 60.-tych a 70.-tych rokoch 20.-teho storočia vznikli prvé techniky podobné refaktorovaniu. Boli vydané vo forme pravidiel pre štruktúrované programovanie a hovorili o tom, ako odstraňovať tzv. go-to príkazy zo zdrojového kódu a ako zjednodušovať riadiacu logiku programov.

Prvými ľuďmi, ktorí explicitne opísali refaktorovanie, boli Ward Cunningham a Kent Beck. V 80.-tych rokoch 20. storočia pracovali s programovacím jazykom Smalltalk a vypracovali proces vývoja softvéru, ktorý integruje refaktorovanie a základné princípy tvorby softvéru. Táto metóda vývoja softvéru sa v súčasnosti označuje ako „Extrémne Programovanie“ (Extreme Programming) [6]. Myšlienky Warda a Kenta neskôr silno ovplyvnili celú komunitu Smalltalku.

Ďalším, kto prispel k pokroku výskumu refaktorovania, bol profesor Illinoiskej univerzity Ralph Johnson, ktorý zistil, ako môže refaktorovanie prispieť k vývoju flexibilných softvérových „skeletov“ (framework).

Jeho študent William Opdyke potom spravil vedecký výskum princípov, ktoré podporujú refaktorovanie [5]. Opdyke vydal v roku 1992 svoju PhD. dizertačnú prácu, v ktorej skúmal zachovanie sémantiky programu ovplyvneného refaktorovaním a tiež v nej uviedol zoznam refaktorovaní.

John Brant, Don Roberts a Ralph Johnson potom uviedli myšlienky W. Opdyka do praxe a vytvorili softvérový nástroj nazvaný Refactoring Browser, ktorý umožňuje refaktorovať programy napísané v jazyku Smalltalk [9].

V roku 1999 bola vydaná prvá komerčná kniha o refaktorovaní od Martina Fowlera, v ktorej zhrnul svoje skúsenosti a vedomosti o refaktorovaní [1].

Viac o výskume refaktorovania prebiehajúcom v posledných rokoch si povieme v 5. kapitole v tejto práce.

1.3 Použité konvencie

V texte diplomovej práce sú použité nasledovné konvencie:

„definícia alebo citát“
Definícia alebo citát sú napísané v úvodzovkách kurzívou.

„preklad“
Názvy a pojmy preložené do slovenského jazyka, ktorých preklad nie je jednoznačný, sú v úvodzovkách. Pôvodný názov alebo pojem je uvedený v zátvorkách v pôvodnom jazyku.

[X]
Odkaz na zdroj, z ktorého boli čerpané informácie o danej téme, je uvedený v hranatých zátvorkách. X označuje poradové číslo referencie, ktorých zoznam sa nachádza na konci práce v prílohe.

Ukážky zdrojových kódov programov
Časti zdrojového kódu použité v diplomovej práci ako príklady sú napísané v programovacom jazyku JAVA 2 a použitý je pre ne iný typ písma. Časti zdrojového kódu, ktoré sú v príklade dôležité, sú zvýraznené tučným písmom.

UML schémy
V texte diplomovej práce je v niektorých prípadoch použitá, namiesto zdrojového kódu, UML schéma. Čitateľ, ktorý nepozná tento spôsob znázorňovania štruktúry programov, by sa mal najskôr s touto tematikou oboznámiť [14].

2 Refaktorovanie

V tejto kapitole sa pokúsime vysvetliť podstatu refaktorovania a pojmy s ním súvisiace. Potom na príkladoch ukážeme niektoré základné refaktorovania a uvedieme výhody a nevýhody, ktoré refaktorovanie prináša do procesu vývoja softvéru.

2.1 Reštrukturalizácia a refaktorovanie

Vytváranie softvéru je náročný a komplikovaný proces, pri ktorom sa často stáva, že výsledný produkt je nekompletný, nesprávny alebo dodaný neskoro. Niekoľko vedeckých štúdií veľkých softvérových systémov ukázalo, že podstatná časť z celkových nákladov určených na vývoj býva použitá na ich údržbu. Prekvapujúcou je však skutočnosť, že podiel zdrojov potrebných na údržbu programov z roka na rok rastie na úkor ostatných fáz vývoja softvéru.

Príčinou tohto efektu je skutočnosť, že požiadavky na vývoj a zlepšovanie moderného softvéru sa stále zvyšujú. Jediným konštantným faktorom v procese vývoja softvéru sa tak stáva potreba neustálych zmien. Úspešnými sa stávajú softvérové systémy, ktoré sa plynulo vyvíjajú a adaptujú na meniace sa požiadavky. Spravovateľnosť a udržovateľnosť softvéru sú preto považované za rozhodujúce faktory kvality softvéru.

Ako sa softvér postupom času mení, stáva sa zložitejším, jeho zdrojový kód sa odchyľuje od pôvodného dizajnu a množstvo zdrojov potrebných na jeho údržbu stúpa. Zlepšovanie metód a nástrojov na tvorbu softvéru, nerieši problém zložitosti, ale len zvyšuje možnosti implementovať do programu viac nových funkcií v rovnakom čase. To nakoniec vedie opäť k väčšej zložitosti výsledného softvéru.

Na nájdenie cesty z tohto bludného kruhu je nutné použiť techniku, ktorá postupným zlepšovaním vnútornej kvality softvéru redukuje jeho zložitosť. Tento prístup bol pomenovaný ako problém reštrukturalizácie zdrojového kódu programu, alebo špeciálne pre prípad vývoja objektovo-orientovaného softvéru ako problém refaktorovania.

Reštrukturalizácia programu je zmena zdrojového kódu v zmysle klasického štruktúrovaného dizajnu za účelom vylepšenia jeho štruktúry. Presne bol pojem reštrukturalizácie definovaný [3]:

„Reštrukturalizácia je transformácia z jednej formy reprezentácie programu na druhú, pri zachovaní vonkajšieho správania systému, čiže jeho funkcionality a sémantiky.“

Refaktorovanie bolo neskôr zadefinované ako špeciálny prípad reštrukturalizácie. Prvý krát ho definoval W. Opdyke vo svojej PhD dizertačnej práci [5]. Definoval ho nasledovne:

„Refaktorovanie je proces zmeny objektovo-orientovaného softvérového systému takým spôsobom, ktorý nezmení vonkajšie správanie programu, ale vylepší jeho vnútornú štruktúru.“

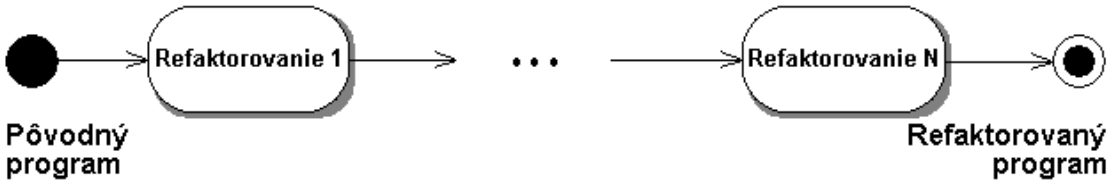
Cieľom refaktorovania je teda vylepšiť dizajn zdrojového kódu a tým spraviť softvér ľahšie pochopiteľný a udržovateľný. Hlavná myšlienka refaktorovania spočíva v rozdelení (faktorizácii) tried, metód a premenných v hierarchii tried tak, aby boli umožnené ľahké zmeny a rozšírenia v budúcnosti.

V programe môžeme vykonať veľa zmien, ktoré nezmenia jeho správanie. Za refaktorovanie však považujeme len také zmeny, ktoré zlepšujú jeho vnútornú štruktúru. Dobrým príkladom pre zmenu softvéru, ktorá nie je refaktorovaním, je optimalizácia výkonnosti. Podobne ako refaktorovanie, aj optimalizácia výkonnosti, väčšinou nemení správanie programu (okrem rýchlosti).

Výsledkom optimalizácie obvykle býva nielen väčšia rýchlosť, ale aj väčšia zložitosť programu, čo je v protiklade s cieľom refaktorovania.

Okrem predošlej definície refaktorovania ako procesu, sa často týmto slovom označuje aj jedna zmena zdrojového kódu, ktorá zlepšuje jeho štruktúru a samozrejme zachováva správanie. Takže, ako programátor môžeme stráviť refaktorovaním niekoľko hodín, počas ktorých aplikujeme niekoľko rôznych refaktorovaní.

Obrázok 1 : Refaktorovanie (ako proces) sa môže skladať aj z viac jednoduchých refaktorovaní (zmien).



Ako sme spomínali v úvode, refaktorovanie má významnú úlohu aj v menej formálnych modeloch vývoja softvéru, akým je napríklad extrémne programovanie. Pri extrémnom programovaní sa často vykonávajú revízie napísaného zdrojového kódu, ktorých cieľom je dosiahnuť najjednoduchší možný dizajn. Refaktorovanie sa využíva práve počas týchto revízií na zjednodušovanie dizajnu programu.

Okrem extrémneho programovania sa refaktorovanie a reštrukturalizácia dajú veľmi dobre použiť aj v procese reverzného inžinierstva [4]. Často sa stáva, že problém, ktorý sa rieši, bol už vyriešený predtým v inom projekte. Cieľom reverzného inžinierstva je pochopiť existujúce riešenie a transformovať ho do inej formy. A práve tu môžeme veľmi dobre využiť refaktorovanie na konvertovanie starého zdrojového kódu do modulárnejšej a štruktúrovanejšej formy, alebo ho môžeme použiť pri prechode na iný programovací jazyk.

Dnes už je refaktorovanie zaraďované k pokročilejším technológiám a patrí k hlavným technikám vo vývoji softvéru. Napriek týmto skutočnostiam nie je ešte dostatočne známe a používané.

Jednou z príčin slabého využívania je, že ľudia nevedia ako správne refaktorovať. Výhody, ktoré refaktorovanie prináša sú skôr dlhodobé a je ťažké uvedomiť si, prečo naňho treba vynakladať úsilie teraz. Ďalšou z príčin slabého používania je obava, že refaktorovaním by sme mohli poškodiť už existujúci program.

Za kľúč k vývoju lepších softvérových systémov je považované pochopenie, kde a ako treba zmeniť zdrojový kód a v poznaní, ako refaktorovanie ovplyvňuje kvalitu softvérového systému

2.2 Proces refaktorovania

Refaktorovaniu predchádza objavenie časti zdrojového kódu programu, ktorá má zlý alebo nevyhovujúci dizajn. Takýto kus kódu sa zvykne označovať ako „páchnuca“ časť kódu (bad smell code). Momentálne neexistuje žiadny všeobecný postup, ktorý by uľahčoval ich vyhľadávanie v programe. Naučiť sa rozpoznávať páchnuce časti programu je teda pre programátora najmä otázka využitia získaných skúseností.

Po identifikácii páchnucej časti programu prichádza fáza, v ktorej sa treba rozhodnúť, ako ju z programu odstrániť, resp. ako zmeniť program tak, aby sme získali lepší dizajn a zachovali vonkajšie správanie. Výber vhodného refaktorovania taktiež závisí hlavne od intuície programátora. Vhodné je však vybrať také refaktorovanie, ktoré transformuje menený zdrojový kód na nejaký dizajnový vzor (design pattern) [8].

Dizajnové vzory nám hovoria, ako riešiť často sa opakujúce problémy z dizajnom programu. Sú to vypracované riešenia konkrétnych problémov,

ktoré môžu nastať pri návrhu dizajnu softvéru. Refaktorovanie zase mení zdrojový kód programu tak, aby mal lepší dizajn. A práve tento „lepší dizajn“ je často nejaký dizajnový vzor. Takže v mnohých prípadoch končí refaktorovanie tým, že sa zmení páchnuca časť kódu na inštanciu nejakého dizajnového vzoru.

Ak už vieme, čo a ako chceme zmeniť, tak môžeme začať refaktorovať program. Pre samotné refaktorovanie je najlepšie použiť nejaký softvérový nástroj, ktorý ho spraví za nás. Ak softvérový nástroj nemáme alebo neexistuje, potom musíme program refaktorovať manuálne.

Prvým krokom pri manuálnom refaktorovaní by malo byť vytvorenie množiny testovacích programov. Testovacie programy nám pomôžu predchádzať a odhaľovať chyby, ktoré by sme mohli počas refaktorovania spraviť. Manuálne refaktorovanie je najlepšie robiť v malých krokoch. Po každom vykonanom kroku by sme mali otestovať, či sme do programu nevniesli chybu alebo, či sme nezmenili jeho správanie. Ak sme aj náhodou niečo poškodili, tak sa môžeme ľahko vrátiť k predchádzajúcemu kroku.

Obrázok 2 : Schéma postupu pri refaktorovaní programu.

2.3 Kedy použiť refaktorovanie

Refaktorovanie rozdeľuje čas potrebný na vývoj softvéru na dve odlišné fázy: pridávanie funkčnosti a refaktorovanie. Počas fázy pridávania funkčnosti by sa nemal meniť existujúci kód. Mal by sa len písať kód pre nové funkcie programu. Naopak, počas refaktorovania sa mení len kód programu a jeho

funkčnosť sa zachová. Je dobré nemiešať tieto dve fázy a vždy mať na pamäti, v ktorej z nich sa práve nachádzame.

Prirodzeným teda býva použitie refaktorovania pred alebo po implementácii nových funkcií do programu. Refaktorovaním pred pridaním novej funkčnosti si môžeme uľahčiť jej implementáciu do programu. Refaktorovaním po pridaní novej funkčnosti do programu by sme zase mali program upraviť tak, aby sme udržali kvalitu jeho dizajnu.

Existujú prípady, kedy by sa program nemal refaktorovať. Ak je napríklad zdrojový kód programu príliš chaotický, môže byť jednoduchšie napísať celý program od začiatku. Jasným prípadom, kedy by sme nemali program refaktorovať, ale prepísať ho odznova, je prípad, keď program nefunguje alebo obsahuje príliš veľa chýb.

2.4 Katalóg refaktorovaní

Väčšina z refaktorovaní sú jednoduché transformácie a ich mená určujú aj ich cieľ. Vo všeobecnosti sa skladajú z malých krokov, ktoré robia dizajn viac objektovo-orientovaným. M. Fowler opísal vo svojej knihe [1] katalóg s viac ako 70 často používanými refaktorovaniami. Cieľom katalógu refaktorovaní je opísať refaktorovania tak, aby bolo jasné kedy a ako sa majú používať. Navyše každé refaktorovanie v tomto katalógu má jednoznačné meno, čo pomáha pri vytváraní jednotného slovníka tvorcov softvéru.

Katalóg obsahuje refaktorovania na úpravu metód a premenných (Extract Method, Inline Method, ...), presúvanie funkcionality a organizovanie dát (Move Method, Move Field, Hide Delegate, Replace Data Value with Object, ...). Ďalej sa zaoberá logikou v programoch (Replace Conditional with Polymorphism, Introduce Null Object, ...), vytváraním objektov (Replace Constructor with Factory Method, ...) a dedičnosťou (Replace Inheritance with

Delegation, Replace Conditional with Polymorhism, ...). Všetky refaktorovania sú v ňom rozdelené celkovo do šiestich skupín.

Pozrime sa teraz na niektoré refaktorovania. Pre každé refaktorovanie si opíšeme kontext jeho použitia, jeho ciele a príklad na ilustráciu. Podrobný mechanizmus aplikovania sa nachádza v spomínanej knižke M. Fowlera.

Najskôr si ukážeme refaktorovania, ktoré môžu zlepšiť zrozumiteľnosť a čitateľnosť zdrojového kódu.

Premenovanie symbolov (Rename variable / method)
K najjednoduchším, ale účinným refaktorovaniam patrí premenovanie symbolov v kóde programu. Najčastejšie ide o premenovanie premennej, metódy alebo triedy. Cieľom tohto refaktorovania je dosiahnuť lepšiu čitateľnosť a pochopiteľnosť kódu programu. Dobrý kód by mal jasne hovoriť o tom, čo robí. Preto by sme sa nikdy nemali báť premenovať programové symboly tak, aby sme zvýšili jednoduchosť programu.

Pozrime sa na nasledujúcu metódu. Čo je jej cieľom? Čo tento kus programu robí? Programátor, ktorý ju vidí prvý krát, rozumie jednotlivým príkazom, ale nevie, čo je jej účelom ako celku a prečo to robí práve takým spôsobom?

Príklad 1 : Časť programu zo zlým pomenovaním metódy a jej vstupných parametrov.

```
double zistiKoef (int v, int d) {
        if (v > 60 or d == 6) {
                return 0.15;
        }
        else {
                return 0;
        }
}
```

Zrozumiteľnosť a cieľ metódy zistiKoef je veľmi nízka, a preto ju premenujeme, aby sme ju zlepšili. Takisto premenujeme aj jej vstupné parametre, ktorých názvy nám tiež nič nehovoria.

Príklad 2 : Refaktorovaná metóda z príkladu 1.

```
double zistiZlavu (int vekZakaznika, int denVTyzdni) {
        if (vekZakaznika > 60 or denVTyzdni == 6) {
                return 0.15;
        }
        else {
                return 0;
        }
}
```

Keď je refaktorovanie metódy zistiZlavu hotové, nesmieme zabudnúť opraviť aj všetky volania pôvodnej funkcie vo zvyšku programu. Ako vidíme na tomto triviálnom príklade, ak je metóda dobre pomenovaná, tak vôbec nemusíme čítať aj jej implementáciu, aby sme vedeli, čo robí. Zrozumiteľné a výstižné pomenovanie programových symbolov je vec, na ktorej by si mal dať záležať každý programátor.

Extrahovanie metódy (Extract method)

Cieľom extrahovania metódy je presunúť kus zdrojového kódu do samostatnej metódy a lepšie tak určiť zámer tohto kódu. Takto refaktorovať sa oplatia najmä metódy, ktoré sú príliš dlhé alebo metódy, ktoré sú nejasné a obsahujú mnoho komentárov.

Príklad 3 : Refaktorovanie programu extrahovaním novej metódy vytlacDetaily z metódy vytlacFakturu.

```
void vytlacFakturu (int vyrobok, String zakaznik, double cena) {
        vytlacVyrobok(vyrobok);

        // vytlac detaily
        System.out.println ("Meno:" +zakaznik);
        System.out.println ("Cena:" + cena);
}
```


```
void vytlacDetaily (String zakaznik, double cena) {
        System.out.println ("Meno:" +zakaznik);
        System.out.println ("Cena:" + cena);
}


void vytlacFakturu (int vyrobok, String zakaznik, double cena) {
        vytlacVyrobok (vyrobok);
        vytlacDetaily (zakaznik, cena);
}
```

Problémy pri tomto refaktorovaní sú najmä s lokálnymi premennými, ktorých obsah sa mení v extrahovanom kóde. Ak takéto premenné extrahovaný kód obsahuje, musíme zabezpečiť, aby sa ich zmena prejavila aj v pôvodnej metóde.

Vloženie metódy (Inline method)

Vloženie metódy je inverzné refaktorovanie k extrahovaniu metódy. Používa sa, ak natrafíme v zdrojovom kóde na metódu, ktorej implementácia je tak jasná ako jej meno. Vtedy je vhodné nahradiť volanie metódy jej kódom a metódu zrušiť.

Pred tým, ako začneme samotné refaktorovanie, musíme ešte overiť, či vybraná metóda náhodou nie je polymorfná. Zrušenie polymorfnej metódy spôsobí chybu, lebo podtriedy, ktoré ju predefinovávali, ju už viac nebudú môcť nájsť.

Príklad 4 : Refaktorovanie programu vložením metódy viacAkoTriChyby do metódy zistiHodnotenie.

```
boolean viacAkoTriChyby () {
        return ( _pocetChyb > 3 );
}

int zistiHodnotenie () {
        return ( viacAkoTriChyby () ? 2 : 1 );
}
```


```
int zistiHodnotenie () {
        return ( _pocetChyb > 3 ? 2 : 1 );
}
```

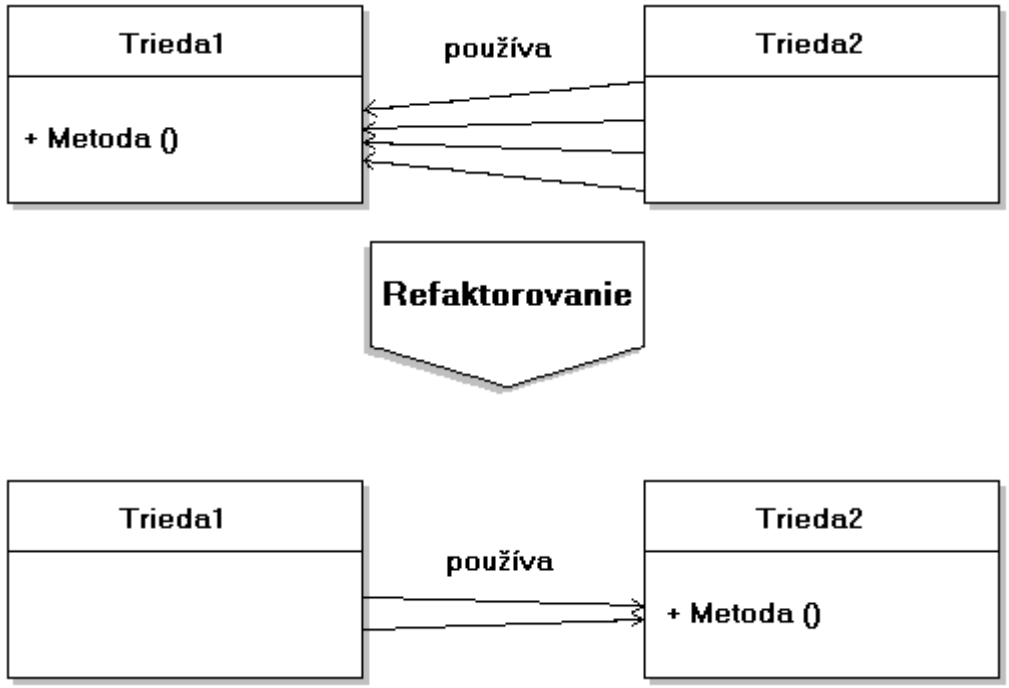
Pri aplikovaní tohto refaktorovania si treba dať pozor aj na pomenovanie lokálnych premenných, ktoré sa nachádzajú v metóde, ktorú ideme zrušiť. Ak by sa lokálna premenná z tejto metódy volala rovnako, ako nejaká premenná v cieľovej metóde, tak môže prepísať jej obsah a tým zmeniť správanie programu.

Teraz sa pozrime na dve refaktorovania, ktoré presúvajú funkčnosť v hierarchií tried.

Presunutie premennej alebo metódy (Move field, method)

Premennú alebo metódu by sme mali presunúť na iné miesto, ak je využívaná inými triedami viacej ako tou, v ktorej sa nachádza. Aplikovaním tohto refaktorovania znížime vzájomnú previazanosť medzi triedami.

Príklad 5 : UML schéma refaktorovania presunutia metódy medzi dvoma triedami. Pred refaktorovaním druhá trieda volala metódu z prvej štyri krát. Po refaktorovaní sa prvá trieda odvoláva na presunutú metódu len dva krát. Tým sa znížila vzájomná previazanosť týchto dvoch tried.



Nasleduje príklad refaktorovania, ktorý zjednodušuje podmieňovaciu logiku programu.

Odstránenie riadiacej premennej (Remove control flag)

Pri vykonávaní postupnosti podmieňovacích výrazov sa často objavuje premenná, ktorá riadi vykonávanie nejakej lokálnej časti programu. Používanie

týchto premenných pochádza ešte z čias štruktúrovaného programovania, keď mali metódy a bloky kódu jeden vstupný a jeden výstupný bod. Moderné programovacie jazyky obsahujú možnosti (return, break, continue), ktoré umožňujú vytvoriť viac výstupných bodov. Tým sa podmienky, ktoré používajú riadiace premenné stávajú zbytočnými a môžeme sa ich zbaviť.

Príklad 6 : Refaktorovanie programu odstránením riadiacej premennej najdeny z metódy kontrolaUzivatela.

```
void kontrolaUzivatela (String[] zoznam, String uzivatel) {
        boolean najdeny = false;
        for (int i = 0; i < zoznam.length; i++) {
                if (! najdeny) {
                        if (zoznam[i].equals (uzivatel)) {
                                upozorniAdministratora();
                                najdeny = true;
                        }
                }
        }
}
```


```
void kontrolaUzivatela (String[] zoznam, String uzivatel) {
        for (int i = 0; i < zoznam.length; i++) {
                if (zoznam[i].equals (uzivatel)) {
                        upozorniAdministratora();
                        break;
                }
        }
}
```

Na záver sa pozrime na refaktorovanie, ktoré využíva vlastnosť polymorfizmu, poskytovanú objektovo-orientovanými programovacími jazykmi.

Nahradenie podmienky polymorfizmom (Replace conditional with polymorhism)

Dobrou vlastnosťou polymorfizmu je, že nám umožňuje vyhnúť sa písaniu presných podmienok pri objektoch, ktorých správanie závisí od ich typov. Nasledujúce refaktorovanie využíva túto vlastnosť a dáva nám tú výhodu, že robí podmieňovaciu logiku nezávislú od počtu typov objektov. Ak budeme chcieť neskôr pridať nový objekt s novým správaním, nebudeme už musieť upravovať všetky výskyty tejto podmienky v celom programe.

Samotné refaktorovanie aplikujeme tak, že pre každú vetvu podmienky vytvoríme novú podtriedu a pôvodnú metódu spravíme abstraktnou. V každej novej podtriede potom zadefinujeme metódu, ktorou prekryjeme tú v pôvodnej triede.

Príklad 7 : Refaktorovanie programu nahradením podmienky polymorfizmom.

```
class zamestnanec { ...
        double vypocitajPlat () {
                switch (_typ) {
                        case MANAZER:
                                return 4 * zakladnyPlat () + premie ();
                        case URADNIK:
                                return zakladnyPlat () + premie ();
                }
        }
}
```

```
class zamestnanec { ...
        abstract double vypocitajPlat ();
…

class manazer extends zamestnanec { ...
        double vypocitajPlat () {
                return 4 * zakladnyPlat () + premie ();
        }
…

class uradnik extends zamestnanec { ...
        double vypocitajPlat () {
                return zakladnyPlat () + premie ();
        }
```

2.5 Výhody refaktorovania

Refaktorovanie nie je liek na všetky problémy súvisiace s vývojom softvéru. Môže sa však stať prostriedkom, ktorý nám umožní rýchlejší vývoj a lacnejšiu údržbu softvérových systémov. Hlavné výhody, ktoré refaktorovanie prináša do procesu vývoja softvéru sú:

1. Refaktorovanie pomáha udržiavať dobrý dizajn kódu. Bez refaktorovania sa dizajn zdrojového kódu postupne kazí. Ľudia väčšinou menia program bez toho, aby úplne rozumeli zámerom existujúceho dizajnu, ktorý je skrytý v implementácii programu. Pri takýchto zmenách, ktoré sledujú len krátkodobé ciele, sa stráca štruktúra pôvodného dizajnu, až nakoniec vznikne v programe zmätok.

2. Refaktorovanie zlepšuje zrozumiteľnosť zdrojového kódu programu. Zrozumiteľnosť kódu je vlastnosť, ktorá sa nedá merať presne, pretože

vždy závisí na konkrétnom človeku, ako dokáže program pochopiť. Program, ktorý je dôsledkom refaktorovania lepšie štruktúrovaný, je pre človeka ľahšie pochopiteľný. Refaktorovanie je nástroj, ktorý nám pomáha robiť programy pochopiteľné aj pre iných ľudí, a teda z nás robí lepších programátorov.

3. Refaktorovanie pomáha pri hľadaní chýb v programe. Ak chceme nájsť chybu v kóde programu, musíme ho najskôr pochopiť. Refaktorovaním získame program, ktorý je ľahšie pochopiteľný.

4. Refaktorovanie pomáha rýchlejšie programovať. Toto tvrdenie môže znieť nezmyselne, keďže refaktorovanie je činnosť, ktorú musí programátor vykonávať navyše. Ako sme však spomínali, tak refaktorovaním zlepšujeme a udržujeme zdrojový kód programu. A cieľom udržovania dobre štruktúrovaného kódu je umožnenie rýchlejšieho vývoja softvéru.

2.6 Nevýhody refaktorovania

Proces refaktorovania prináša aj nevýhody, ktoré sú však vo väčšine prípadov vykompenzované jeho výhodami alebo možnosťou ich odstránenia. Hlavnými nevýhodami sú:

1. Refaktorovanie nepridáva do programu žiadne nové funkcie a programátori sú platení za programovanie nových funkcií. Táto nevýhoda je vykompenzovaná rýchlejším vývojom a ľahšou udržovateľnosťou softvéru s lepším dizajnom.

2. Pri refaktorovaní sa môžu vyskytnúť chyby a pokazí sa tým už existujúci, funkčný program. V takomto prípade treba refaktorovanie rozdeliť do menších krokov a po každom kroku otestovať funkčnosť programu.

3. Vyhľadanie „páchnucich“ častí zdrojového kódu je nedeterministický proces a záleží od skúseností a intuície programátora. Keďže nie je ani presne určené, čo je dobrý a čo zlý dizajn, môžu rôzni ľudia zhodnotiť ten istý kód programu úplne inak.

4. Pri refaktorovaní dochádza k zmene zdrojového kódu programu a tým obvykle aj k zmene v jeho výkonnosti. Vo väčšine prípadov platí nepriama úmernosť: Čím je program efektívnejší, tým je ťažšie pochopiteľný pre človeka.

3 Problémy zdrojového kódu

V tejto časti diplomovej práce sa pozrieme na problémy zdrojového kódu, ktoré súvisia s kvalitou jeho dizajnu, a ktoré sa dajú refaktorovaním z programu odstrániť. Ako sme už spomínali, problematické časti programu sa často nazývajú páchnucimi časťami zdrojového kódu programu.

Páchnuce časti kódu hovoria programátorovi, kedy je potrebné softvér refaktorovať. Sú to indikátory problémov zdrojového kódu programu. Takisto ich môžeme považovať aj za mieru, ktorá určuje udržovateľnosť softvéru. Čiže, čím je ich je v zdrojovom kóde viac, tým viac problémov bude s údržbou programu.

Páchnuce časti zdrojového kódu môžeme rozdeliť podľa typu programu, v ktorom sa nachádzajú na dve skupiny. Na tie, ktoré môžu byť vo výslednom programe a na tie, ktoré môžu byť v testovacích programoch. Výsledný program je produkt, ktorý sa odovzdá zákazníkovi na používanie. Naopak, testovacie programy sú len pomocné programy, ktoré pomáhajú overiť správnosť fungovania výsledného programu voči špecifikácií zadanej zákazníkom. Páchnuce časti v každej z týchto dvoch skupín môžeme ďalej rozdeliť na podskupiny podľa spoločných vlastností, ktoré zdieľajú problémy v rámci jednej podskupiny.

3.1 Výsledný program

Problémy sa častejšie vyskytujú vo výslednom programe ako v testovacích programoch. Spôsobené je to neustálym prispôsobovaním výsledného softvéru novým požiadavkám zákazníka.

Pozrime sa teraz na niektoré často sa objavujúce páchnuce časti zdrojového kódu, vyskytujúce sa vo výslednom programe. Pre lepšie pochopenie ich podstaty, ich rozdelíme do podskupín podľa M. Mäntylä [7].

3.1.1 Problémy s veľkosťou

Už pri prvých programovacích jazykoch si ľudia uvedomili, že dlhé kusy zdrojového kódu sú zle udržovateľné a ťažko pochopiteľné. Preto začali programy rozdeľovať do menších celkov. Páchnuce časti kódu v tejto podskupine reprezentujú niečo, čo sa stalo takým veľkým, že už to nedokážeme efektívne spravovať.

Dlhý zoznam parametrov (Long parameter list)
Dlhé zoznamy parametrov v metódach je náročné a ťažké udržiavať konzistentné počas zmien. Pri objektovo-orientovanom programovaní sa nemusia metóde objektu posielať všetky dáta, ktoré potrebuje. Namiesto toho stačí poslať len toľko informácií, aby metóda dokázala ďalšie potrebné údaje získať sama od iných objektov.

Príliš dlhá metóda (Long method)
Nahradením dlhšej časti kódu, volaním niekoľkých metód, môžeme kód programu zjednodušiť. Pri odstraňovaní tohto problému sa musíme hlavne zamerať na výber správnych mien pre nové metódy. Ako sme už hovorili, kľúčom k ľahkému pochopenie programu je dobré pomenovanie symbolov.

Veľká trieda (Large class)
Príliš veľa premenných a metód v jednej triede je živnou pôdou pre vznik duplicitného kódu. Jedna trieda by mala realizovať jeden dobre navrhnutý koncept. Funkcie nad rámec tohto konceptu by mali byť vykonávané na inom mieste. Dodržiavaním tejto zásady sa vyhneme vzniku veľkých a zložitých tried.

Posadnutosť primitívnymi dátovými typmi (Primitive obsession)
Programátori začínajúci s objektovo-orientovaným programovaním sa obvykle snažia používať najmä primitívne dátové typy, prípadne jednoduché dátové štruktúry. Niekedy je však výhodnejšie zmeniť primitívny dátový typ na jednoduchú triedu. Oplatí sa to, ak potrebujeme ozrejmiť, na čo sú dáta používané a aké operácie sú na nich povolené.

Príkladom môže byť použitie jednoduchej triedy na prácu s dátumom, ktorou nahradíme skupinu číselných premenných, v ktorých sme mali dátum uložený.

Zhluky dát (Data clumps)
Niekedy si všimneme rovnaké dátové položky, ktoré sa objavujú spolu, na rôznych miestach v programe. Môže ísť napríklad o skupinu premenných, parametrov pre metódu alebo skupinu lokálnych dát. Takéto dáta by sme mali logicky zoskupiť do samostatnej triedy.

3.1.2 Postrádateľný kód

Spoločnou vlastnosťou pre tieto páchnuce časti kódu je, že nie sú v programe potrebné. Ich odstránením z programu ušetríme čas, ktorý by sme inak potrebovali na ich údržbu.

Duplicitný kód (Duplicated code)
Ak sa v programe nachádzajú rovnaké alebo podobné programové štruktúry na viacerých miestach, mali by sme sa pokúsiť zjednotiť ich. Duplicitný kód zväčšuje veľkosť programu a tým sťažuje jeho udržovateľnosť. Navyše, ak upravujeme duplicitný kód, môže sa stať, že ho na niektorom mieste zabudneme zmeniť a spôsobíme tým chybu v programe.

Špekulatívne zovšeobecňovanie (Speculative generality)
Zákazník často definuje požiadavku na funkciu, o ktorej si myslí, že niekedy v budúcnosti by sa mu mohla hodiť. Na jej implementáciu často potom treba

vytvárať nové metódy alebo aj triedy. V horšom prípade treba upravovať aj existujúci program a to kvôli niečomu, čo možno ani nebude potrebné. Špekulatívne zovšeobecňovanie môže byť odhalené, ak jediným užívateľom triedy alebo metódy je testovací program.

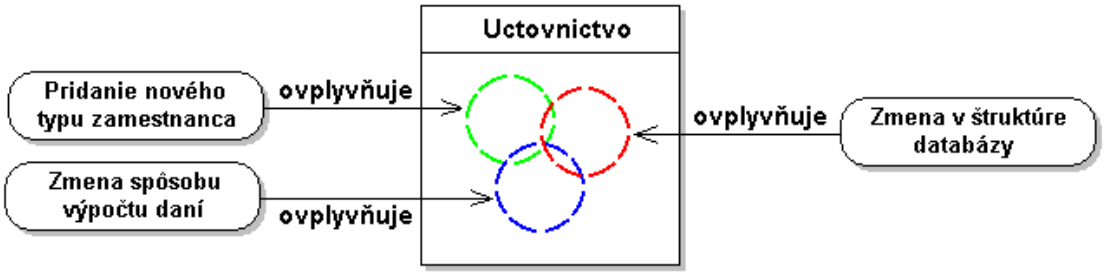
3.1.3 Ťažkopádnosť zmien

Problémy z tejto podskupiny bránia v ľahkej implementácií nových funkcií do programu. Ak chceme program vyvíjať rýchlejšie, mali by sme takéto páchnuce časti refaktorovaním odstrániť čo najskôr.

Rozbiehavá modifikácia (Divergent change)

Softvér by mal byť navrhnutý tak, aby sa dal ľahko zmeniť. Keď modifikujeme program, prejdeme na miesto, ktoré ideme upraviť, a potom vykonáme samotnú zmenu. Problém je, ak jednu triedu potrebujeme meniť rôznymi spôsobmi kvôli rôznym príčinám. Čiže, kvôli rôznym úpravám potrebujeme, vždy meniť tú istú triedu. Potom by sme mali mať radšej viacej tried.

Obrázok 3 : Príklad problému rozbiehavej modifikácie. Trieda Uctovnictvo, zaoberajúca sa podnikovým účtovníctvom, musí byť pri každej zo zmien, upravená iným spôsobom. Teda každý druh zmeny ovplyvňuje inú množinu metód a premenných.

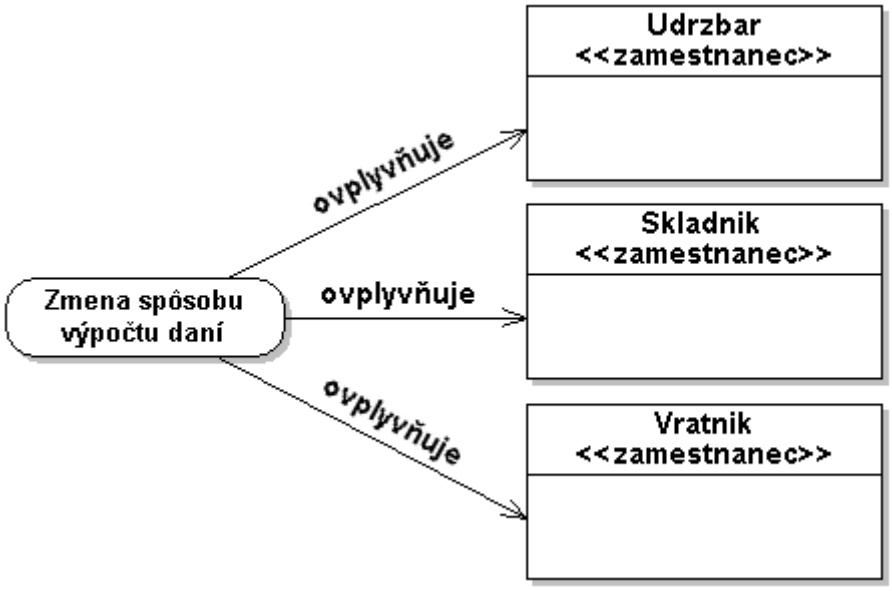


Operácia s brokovnicou (Shotgun surgery)

Tento problém je opakom predchádzajúceho. Vzniká, ak nejaký typ zmeny programu vyžaduje veľa malých zmien na rôznych miestach v rôznych

triedach. Môže byť ťažké nájsť všetky miesta, ktoré treba zmeniť a ľahko sa potom môže zabudnúť na nejakú dôležitú úpravu. V tomto prípade je najlepšie pokúsiť sa presunúť menené časti kódu na jedno miesto alebo do jednej triedy.

Obrázok 4 : Príklad problému operácie s brokovnicou. Jeden druh zmeny ovplyvňuje zdrojový kód viacerých tried.



3.1.4 Previazanosť tried

Problémy v tejto podskupine súvisia s dátovou komunikáciou a zapuzdrením tried, ktoré spôsobujú ich priveľkú previazanosť a malú flexibilitu v kóde programu.

Sprostredkovateľ (Middle man)
Objekt sa stáva sprostredkovateľom, ak väčšinu svojich metód realizuje volaním metód druhého objektu. Hoci tento prístup patrí k štandardným riešeniam v objektovo-orientovanom programovaní [8], treba ho používať veľmi opatrne. Sprostredkovateľská trieda zneprehladňuje program a navyše aj spomaľuje jeho beh.

Znak závisti (Feature envy)

Typickou chybou je, ak metóda viacej používa dáta alebo metódy inej triedy ako tej, v ktorej sa momentálne nachádza. Riešením je presunúť metódu na správne miesto. Ak sa metóda odkazuje na viacej iných tried, tak ju môžeme presunúť do triedy, ktorú využíva najviac.

3.1.5 Zlé používanie tried

Tieto problémy vznikajú, ak programátor nedokáže úplne využiť možnosti, ktoré ponúka objektovo-orientované programovanie. Výsledkom je potom program, ktorý neefektívne využíva systémové zdroje.

Dátová trieda (Data class)

Dátové triedy obsahujú len premenné a metódy potrebné k ich čítaniu a zapisovaniu. Ich jediným cieľom je držať potrebné dáta. Dátovú triedu by sme sa mali snažiť rozšíriť o nové metódy tak, že do nej presunieme metódy, ktoré jej dáta používajú.

Pomocná premenná (Temporary field)

Niekedy majú triedy zadefinované premenné, ktoré sú použité len v istých prípadoch ako pomocné premenné ne uchovanie medzivýsledku. Zdrojový kód takejto triedy je metúci, lebo každý intuitívne predpokladá, že objekt potrebuje všetky svoje premenné.

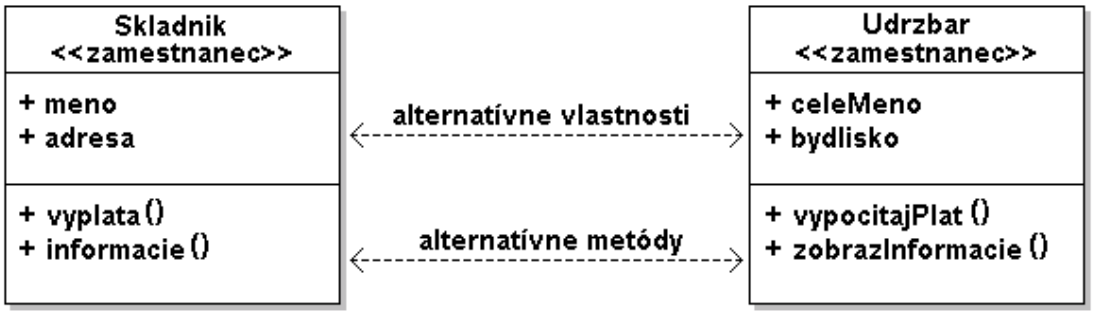
Logické „switch“ štruktúry (Switch statements)

Logické „switch“ (alebo „case“) štruktúry často spôsobujú duplicitný kód v programe a bývajú roztrúsené po celom programe. Ak do nich chceme pridať novú podmienku, musíme prehľadať celý program. Namiesto takýchto logických štruktúr je lepšie využiť polymorfizmus, ktorý nám poskytuje objektovo-orientované programovanie.

Alternatívne triedy z rôznymi rozhraniami (Alternative classes with different interfaces)

Metódy, ktoré vracajú alebo vykonávajú rovnaké úlohy v rôznych triedach by mali byť pomenované rovnako. Jednou z možností, ako to zabezpečiť, je oddediť tieto triedy od spoločnej rodičovskej triedy.

Obrázok 5 : Príklad alternatívnych tried s rôznymi rozhraniami. Alternatívne triedy Sofer a Udrzbar, určené pre dva druhy zamestnancov, majú rôzne názvy pre vlastnosti a metódy, ktoré vykonávajú rovnaké funkcie.



Odmietnuté dedičstvo (Refused bequests)

Podtriedy dedia vlastnosti a metódy od svojich rodičov aj v prípadoch keď ich nepoužívajú. Táto situácia nastáva, ak je zle navrhnutá hierarchia tried a treba ju reorganizovať.

3.1.6 Ostatné

V tejto podskupine sú dva zvyšné problémy, ktoré nezapadajú do žiadnej z predchádzajúcich podskupín.

Komentáre (Comments)

Neprehľadný a zložitý kód programu vedie k nárastu množstva komentárov. Programátori sa boja meniť takýto kód a namiesto toho, aby ho opravili, radšej ho okomentujú. Komentáre zdrojového kódu samé o sebe nie sú

páchnucimi časťami kódu programu. Často sa však používajú na opísanie ich funkčnosti, a preto môžu byť dobrou pomôckou pri ich vyhľadávaní.

Nekompletná knižnica (Incomplete library)
Použitie cudzej knižnice býva niekedy o veľa problematickejšie ako vytvorenie novej vlastnej. Dôvodom býva to, že úprava cudzej knižnice často krát nie je možná a navyše len málokedy nájdeme knižnicu, ktorá presne vyhovuje našim požiadavkám.

3.2 Testovacie programy

Refaktorovanie a testovacie programy sú kľúčové aspekty metódy softvérového inžinierstva nazvanej extrémne programovanie [6]. Písanie a udržovanie testovacích programov je pri tejto metóde súčasťou procesu programovania, a preto sú aj testy podrobované revíznym kontrolám.

Pri použití metódy extrémneho programovania sa odporučuje napísanie testovacej triedy pre každú triedu v systéme. Ideálne je, ak medzi výsledným a testovacími programami dosiahneme pomer 1:1 [6]. Nie je teda prekvapujúce, že pri takomto množstve sa musíme starať aj o testovacie programy.

Pri refaktorovaní testovacích programov sa zistilo, že množina problémov, ktoré sa môžu vyskytnúť v ich zdrojovom kóde, je iná ako pre výsledný program. Preto boli pre testovacie programy nájdené iné refaktorovania [10].

Páchnuce časti, objavujúce sa v testovacích programoch, môžeme rozdeliť do štyroch podskupín. Zamerať by sme sa mali najmä na problémy v prvých dvoch podskupinách, pretože tie môžu spôsobiť, že testovací program vráti nesprávny výsledok.

3.1.1 Podmienky testovania

Ak testovací program nemá zabezpečené vhodné podmienky, jeho výsledky sa môžu javiť ako náhodné. Aby sme sa vyhli takýmto problémom, mali by sme na začiatku každého programu skontrolovať dostupnosť testovaných dát a potrebných systémových zdrojov.

Optimistický predpoklad zdrojov (Resource optimism)
Testovací program, ktorý robí optimistické alebo pesimistické predpoklady o zdrojoch (existencia alebo absencia adresára, či tabuľky v databáze), môže vracať na výstup informácie nezávisle od zadávaných vstupov.

Vojna testovacích programov (Test run war)
Takáto vojna nastane, ak testy bežia dobre pokiaľ sú spúšťané sekvenčne, ale zlyhávajú, ak sú spúšťané paralelne. Spôsobené je to väčšinou obsadením systémových zdrojov jedným z testovacích programov.

3.1.2 Chybné testovanie

Problémy v tejto podskupine spôsobujú, že testovací program môže vrátiť nesprávny alebo nejednoznačný výsledok.

Citlivá rovnosť (Sensitive equality)
Problémom sa často stáva testovanie na rovnosť konštanty s vrátenou hodnotou funkcie testovaného objektu. Typickým použitím je vypočítanie hodnoty, potom jej skonvertovanie na reťazec a porovnanie s konštantou. Niekedy sa stáva, že sa zmení funkcia, ktorá konvertuje hodnotu na reťazec a testy potom vracajú zlé výsledky.

Ruleta tvrdení (Assertion roulette)
Moderné programovacie jazyky umožňujú používať v programe konštrukciu nazvanú tvrdenie (assertion). Ide o príkaz, resp. funkciu, ktorej vstupným

parametrom je výrok. V prípade, že je zadaný výrok vyhodnotený ako nepravdivý, tak je o tom upozornený užívateľ. Spôsob upozornenia je závislý od použitého programovacieho jazyka.

V testovacom programe by sme mali ku každému tvrdeniu pridať aj jeho adekvátny opis. Ak bude nejaký test obsahovať veľa tvrdení a nejaké z nich zlyhá, ľahko potom zistíme, ktoré to bolo.

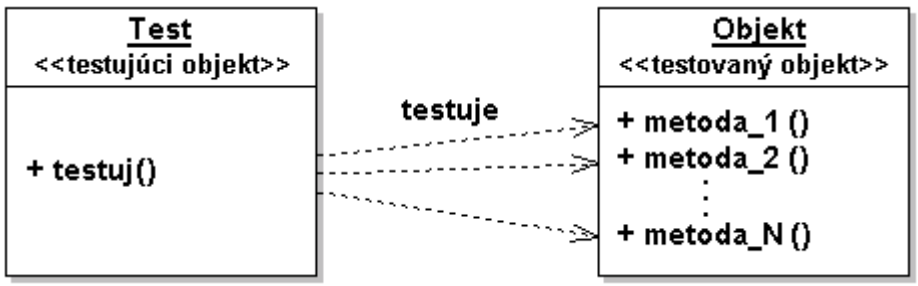
3.1.3 Rozsah testovania

Testovací program by mal byť zameraný na jednu ucelenú oblasť funkčnosti programu alebo triedy. Vyhýbať by sa mal testovaniu problémov, ktoré sú mimo rozsah stanovenej funkčnosti.

Nepriame testovanie (Indirect testing)
Test by nemal kontrolovať objekty, ktoré mu neboli zadané na vstupe. Napríklad by sme nemali testovať objekty, ktorých referencie sme získali počas behu testu. Kontrola týchto objektov by mala byť robená samostatne.

Snaživý test (Eager test)
Keď sa snaží jedna testovacia metóda skontrolovať veľa metód testovaného objektu, test sa stáva ťažšie čitateľný a udržovateľný. Potom je lepšie rozdeliť testovaciu metódu testu do viacerých testov.

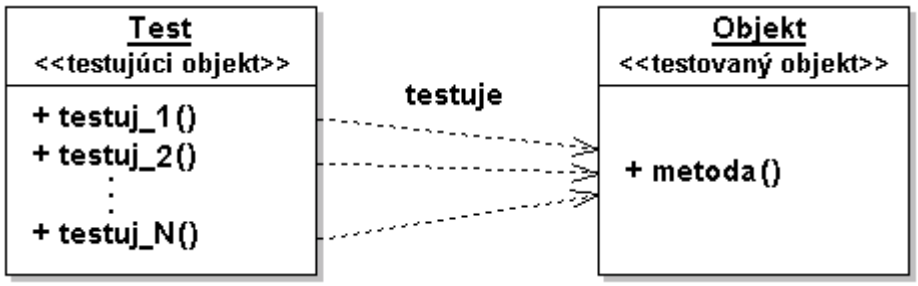
Obrázok 6 : Schéma snaživého testu. Jedna metóda testovacieho objektu kontroluje funkčnosť niekoľkých metód testovaného objektu.



Lenivý test (Lazy test)

Lenivý test je opakom predchádzajúceho problému. Nastane, ak niekoľko metód jedného testu kontroluje tú istú metódu testovaného objektu, pričom využívajú jeho rôzne inštancie. Priamočiarym riešením tohto problému je zlúčenie testov do jedného.

Obrázok 7 : Schéma lenivého testu. Niekoľko metód testovacieho objektu kontroluje funkčnosť jednej metódy testovaného objektu.



3.1.4 Zlý dizajn testu

Problémy v tejto podskupine sú spôsobené zlou implementáciou testu, čoho dôsledkom býva pomalé vykonávanie a nízka udržovateľnosť testovacieho programu.

Všeobecné uchytenie (General fixture)

Mnoho testovacích rozhraní poskytuje možnosť nadefinovať metódu, ktorá bude spustená na začiatku každého testu. Problém nastáva, keď je táto metóda príliš všeobecná a nie každý testovací program potrebuje využiť všetky veci, ktoré robí. Metóda sa stáva potom ťažko pochopiteľná a môže zapríčiniť spomalenie niektorých testov.

Len na testovanie (For testing only)

Metódy a triedy vo výslednom programe, ktoré sú používané len pri testovaní, by mali byť presunuté do testovacieho programu.

4 Otázky výskumu refaktorovania

V tejto kapitole zhrnieme problémy, ktoré nie sú zatiaľ úplne doriešené. K niektorým problémom uvedieme navrhované metódy a techniky na ich riešenie. Samotné problémy sa zvyknú rozdeľovať na základné (určené cieľmi výskumu) a praktické (učené vývojom softvérových nástrojov) [12], [13].

4.1 Základné otázky výskumu

Napriek množstvu práce, ktoré už bolo v oblasti výskumu refaktorovania, vykonanej, stále ostáva ešte veľa otvorených otázok a problémov. Pozrime sa teraz na základné otázky výskumu, ktoré sa týkajú hlavných vlastností refaktorovania.

4.1.1 Čo je to správanie programu a ako môže byť zachované pri refaktorovaní?

Podľa definície sa po vykonaní refaktorovania má zachovať vonkajšie správanie programu. Presná definícia tohto pojmu sa však uvádza len zriedka a intuitívna definícia rovnosti správania programov, získanej porovnávaním vstupov a výstupov, je v mnohých prípadoch nedostatočná. Otázka: Čo je to správanie programu a ako môže byť zachované pri refaktorovaní? je jedným z hlavných problémov výskumu refaktorovania.

Napríklad pre „programy bežiace v reálnom čase“ (realtime systems) je najdôležitejším faktorom čas vykonávania určitých úsekov programu. Pri refaktorovaní však dochádza k zmene zdrojového kódu a teda aj k zmene času vykonávania programu. Analogicky pri „kritických systémoch“ (safety-critical systems) sú definované podmienky, ktoré musia byť po refaktorovaní zachované.

V ideálnom prípade by sa refaktorovaním zachovali všetky vlastnosti pôvodného programu. Z praktického hľadiska to však nie je vždy možné. Preto sa pred refaktorovaním definujú dve skupiny vlastností a to tie, ktoré potrebujeme zachovať a tie, na ktorých nám nezáleží. Potom môžeme použiť nejaké z refaktorovaní, ktoré nám garantuje zachovanie požadovaných vlastností programu. Pozrime sa teraz na dve základné techniky, ktoré sa používajú na zachovanie správania programu z pohľadu užívateľa.

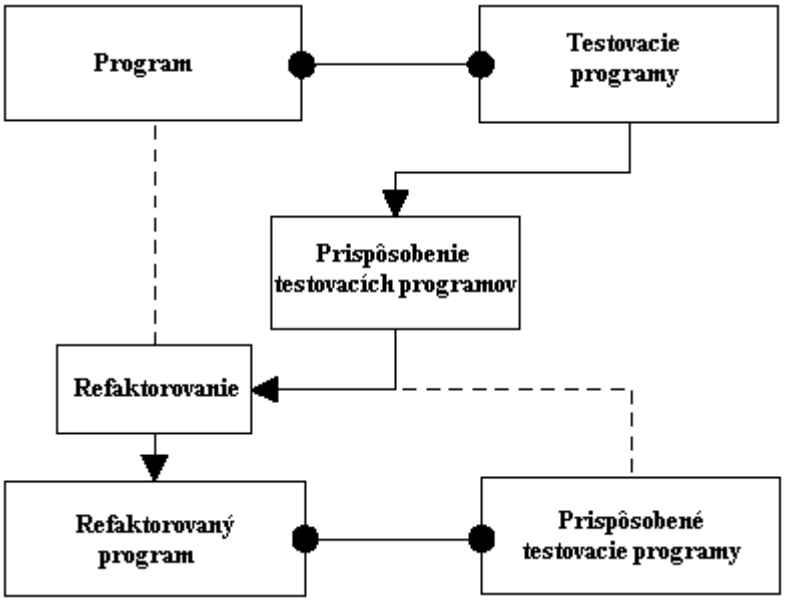
Testovanie softvéru

Jednou z možností ako vyriešiť problém zachovania správania programu je testovanie. Pre program alebo časti programu sa vytvoria testovacie programy, ktoré sú spúšťané po každom refaktorovaní. Ak máme vytvorenú množinu testov, pokrývajúcich všetky prípady, tak po ich vykonaní vieme, či bolo správanie softvéru zachované. Problém je, ako vytvoriť množinu testov pokrývajúcu všetky prípady? To našťastie nie je potrebné. V skutočnosti stačí rozdeliť všetky prípady do tried ekvivalencie a z každej triedy vybrať niekoľko zástupcov.

Nanešťastie existujú aj refaktorovania, ktoré nezmenia správanie programu a testovacie programy napriek tomu vyhlásia chybu. Príčinou tohto efektu je, že testy môžu závisieť od vnútornej štruktúry programu, ktorá bola refaktorovaním zmenená. V praxi to môže viesť k nepríjemným situáciám, keď kód aplikácie a testovacie programy nie sú zosynchronizované a hoci aplikácia funguje správne, testy hlásia chybu.

Na vyriešenie tohto problému môžeme použiť postup [15], pri ktorom sa najskôr upravia testovacie programy podľa plánovaného refaktorovania a až potom sa refaktoruje program.

Obrázok 8 : Náčrt postupu, pri ktorom sa najskôr upravia testovacie programy podľa refaktorovania



Formálne postupy

Ďalším spôsobom ako ukázať, že refaktorovanie zachováva správanie programu, je formálny dôkaz. Pre jazyky s jednoduchou a formálne definovanou sémantikou ako napríklad logický programovací jazyk Prolog, je dôkaz zachovania správania jednoduchý. Pre väčšinu komplexných jazykov, ako C++ alebo Java, kde je zložité formálne definovať sémantiku, musíme použiť obmedzenia na jazykové konštrukcie alebo obmedziť používané refaktorovania [16]. Napriek použitiu obmedzení býva dôkaz, že refaktorovanie zachováva správanie, ťažký, a preto sa formálne dôkazy používajú v praxi len zriedkavo.

4.1.2 Ako môžeme analyzovať závislosti medzi jednotlivými refaktorovaniami?

Ďalším nevyriešeným problémom je určovanie závislosti medzi refaktorovaniami. Ak chceme vytvárať zložené refaktorovania, musíme najskôr určiť, ktoré refaktorovania sú navzájom nezávislé. Takéto refaktorovania môžu byť vykonávané paralelne na urýchlenie celkového procesu. Z tohto pohľadu sú refaktorovania veľmi podobné databázovým transakciám. Podobne, ako pri

transakciách, aj tu sa snažíme vymyslieť techniku na zisťovanie sériovateľnosti refaktorovaní a predchádzať tak možným konfliktom.

4.1.3 Ako zapadá refaktorovanie do procesu vývoja softvéru?

Refaktorovanie mení program potom, ako bol napísaný. Preto je refaktorovanie podporované viacej neformálnymi metódami vývoja ako je napríklad extrémne programovanie. Z toho istého dôvodu refaktorovanie veľmi nezapadá do formálnych postupov ako je vodopádový model vývoja softvéru. Ďalšou otázkou je, ako môžeme refaktorovanie zahrnúť a využiť aj v klasických modeloch vývoja softvéru.

4.1.4 Aké formalizmy a techniky sú najlepšie pre daný problém?

Pre proces refaktorovania bolo vytvorených niekoľko formalizmov a techník. Všetky z nich majú svoje prednosti aj slabiny. Problém je, ako ich môžeme porovnávať, aby sme vedeli určiť ich klady a zápory a ako vybrať tie najvhodnejšie pre riešenie zadaného problému.

4.2 Praktické otázky výskumu

Vytvorenie dostatočnej podpory refaktorovania softvérovými nástrojmi patrí k dôležitým cieľom výskumu, a preto sa značné úsilie venuje aj praktickým otázkam, ktoré sa týkajú problémov vývoja a implementácie takýchto nástrojov.

4.2.1 Ako môžeme určiť, kde a prečo použiť refaktorovanie?

Jedným z cieľov je vytvorenie automatického nástroja na refaktorovanie. Takýto nástroj by sa mal sám rozhodnúť, kde a ako refaktorovať program. Väčšina súčasných softvérových nástrojov poskytuje len podporu pre vykonávanie refaktorovaní.

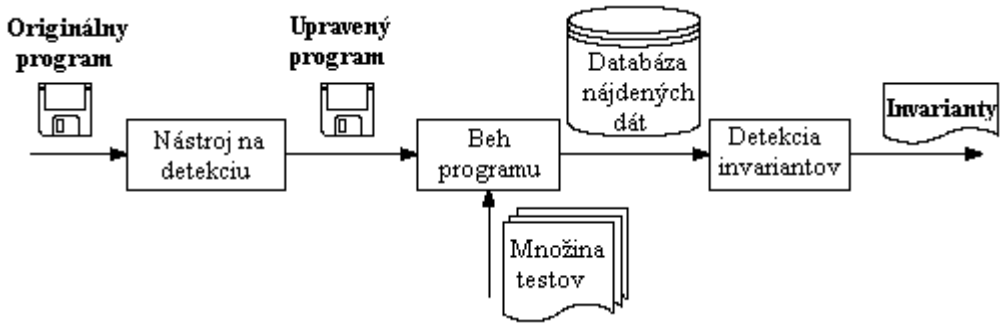
Podstatným faktorom, ktorý treba brať do úvahy pri hľadaní zlého dizajnu programu, je, že štruktúra programu je závislá od konkrétnej aplikačnej domény, pre ktorú je určený. Ak sa napríklad obmedzíme len na oblasť webových aplikácií, tak sa zameriame na chyby, ktoré sa vyskytujú najmä u tohto druhu programov.

Pred refaktorovaním sa ďalej musíme rozhodnúť, na akej úrovni abstrakcie chceme softvér transformovať. A to, či máme zmeniť len samotný program, teda zdrojový kód, alebo aj nejaký iný softvérový artefakt. My sa ďalej obmedzíme len na zdrojový kód a pozrieme sa na tri techniky, ktoré sa dajú použiť pri vyhľadávaní páchnucich častí kódu.

Invarianty programu

Tento postup navrhuje Y. Kataoka [17] a používa invarianty programu na určenie páchnucich častí v zdrojového kódu. Keďže určovanie invariantov je zložitý problém, tak pre ich vyhľadávanie v programe vytvoril jazykovo nezávislý softvérový nástroj Daikon, ktorý momentálne podporuje programovacie jazyky C, Java a Lisp. Nevýhodou tohto nástroja je, že potrebuje analyzovať program dynamicky, t.j. aplikácia musí byť počas analýzy spustená.

Obrázok 9 : Postup pri detekcií invariantov programu implementovaný v nástroji Daikon



Daikon vykonáva zadanú množinu testov a pre každý bod programu si pamätá hodnoty premenných, ktoré tam nadobúdali. Ich analýzou potom zistí invarianty a odporučí zmeny, ktoré by mali byť v programe spravené.

Keďže nie je možné zaručiť, že budú preskúmané všetky možné behy programu, tak nemusia byť vždy nájdené všetky invarianty. Napriek tejto nevýhode sú v praxi dosahované veľmi dobré výsledky. Tento postup je doplnkom k ostatným, ktoré sú založené na statickej analýze programu.

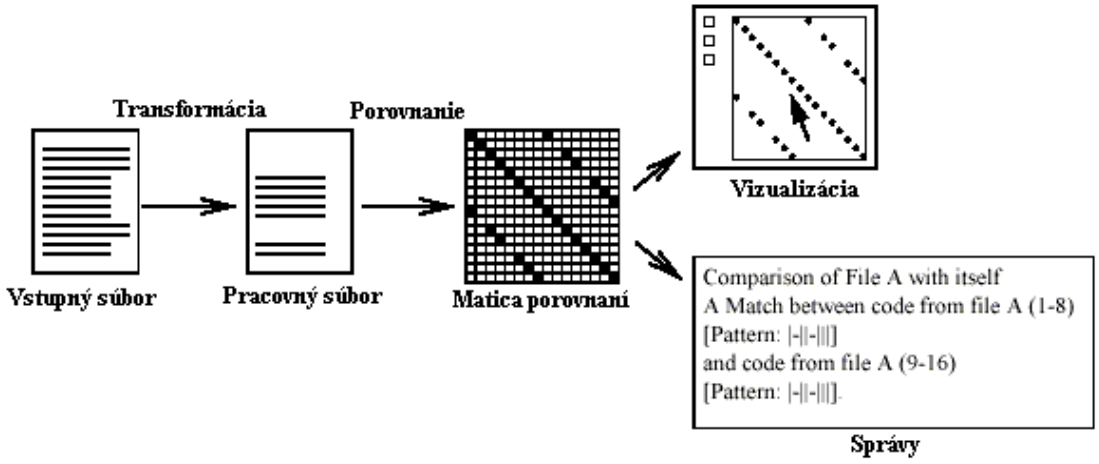
Duplicitné časti programu

Duplicita kódu je jeden z faktorov, ktorý vážne komplikuje správu a vývoj veľkých softvérových systémov. Príčiny, prečo programátori duplikujú kód, spočívajú najmä v ľahšej úprave už existujúceho kódu ako písaní nového a v meraní výkonnosti programátora podľa počtu napísaných riadkov programu. Kopírovaním sa navyše zanášajú do systému skryté chyby a zbytočne sa zväčšuje veľkosť programu.

Väčšina techník na odhaľovanie duplicitných častí kódu je založená na parsovaní programu a teda potrebujú správny parser pre daný programovací jazyk. My si teraz predstavíme metódu, ktorú vytvorili M. Rieger a S. Ducasse [18]. Ich metóda je jazykovo nezávislá a založená je na jednoduchom porovnávaní riadkov programu, na vizuálnej reprezentácií duplicitných častí a na podrobných textových správach. Metóda bola implementovaná ako softvérový nástroj nazvaný Duploc.

Duploc dostane na vstup zdrojový kód programu a porovnáva všetky jeho riadky navzájom. Tým vznikne dvojrozmerná matica, v ktorej každá hodnota je výsledkom porovnania dvoch riadkov. Nakoniec je matica porovnaní analyzovaná. Výsledok je potom zobrazený užívateľovi ako matica so zvýraznenými zónami, ktoré obsahujú duplicitné riadky.

Obrázok 10 : Náčrt postupu jazykovo nezávislej techniky na odhaľovanie duplicitných častí programu



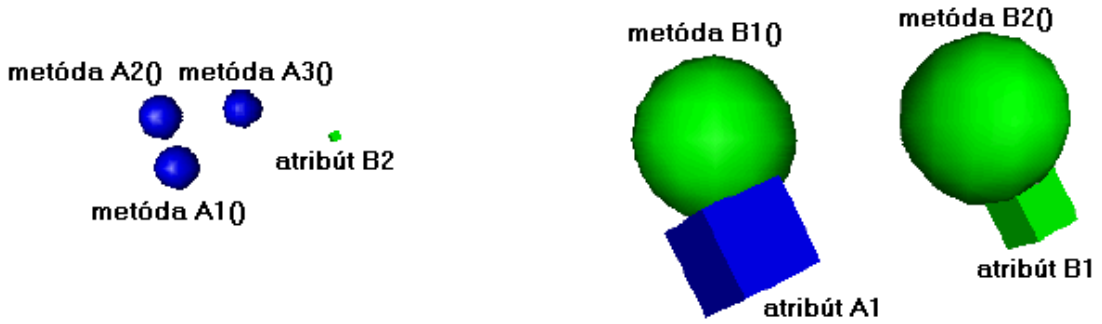
Jadrom tejto metódy je práve spôsob, akým sú dva riadky analyzované. Preto je cieľom do budúcnosti implementácia efektívnejších algoritmov na vyhľadávanie zhodných riadkov programu, a tak skrátiť čas behu analýzy aj pre rozsiahle softvérové systémy.

Softvérové metriky

Softvérové metriky mapujú vnútorné vlastnosti programu na čísla, pomocou ktorých môžeme určovať kvalitu softvéru. Ohodnotené vlastnosti nám potom pomáhajú určiť problémové oblasti programu ako veľkosť, zložitosť alebo kohéznosť. Špeciálnym prípadom softvérových metrík sú objektovo-orientované metriky, ktoré sú zamerané na hodnotenie vlastností tried. Zvláštna pozornosť pri skúmaní softvérových metrík je venovaná hľadaniu častí kódu, ktoré patria významovo k sebe.

F. Simonom vyvinul techniku [19], ktorá je založená na meraní kohéznosti tried, t.j. meria stupeň vzdialenosti entít (atribútov a metód), ktoré sú navzájom závislé. Vzdialenosť dvoch entít je vypočítaná s ich spoločných vlastností. Čím je vzdialenosť entít väčšia, tým menšia je ich kohéznosť. Funkcia na výpočet vzdialenosti medzi metódami a atribútmi vytvára euklidovský priestor, ktorý možno ľahko vizualizovať.

Obrázok 11 : Príklad vizualizácie metód a atribútov dvoch tried A (modrá) a B (zelená). Refaktorované by mali byť atribúty A1 a B2, lebo sú príliš vzdialené od svojich tried.



Tento prístup bol zatiaľ preskúmaný len na malej množine refaktorovaní s obmedzením tried na použitie dedičnosti. Cieľom ďalšieho výskumu sú práve refaktorovania týkajúce sa dedičnosti a spôsob ich vizualizácie.

4.2.2 Ako môžeme definovať refaktorovania formálne, čiže nezávisle od programovacieho jazyka?

Problém určenia, kde a ako refaktorovať program, nás privádza k ďalšiemu problému, ktorý sa týka definície refaktorovania. Často sa refaktorovania opisujú len neformálne a ich interpretácia je otvorená. Čiže správnosť ich použitia závisí od správnosti ich pochopenia programátorom. Softvérové nástroje však potrebujú presnú špecifikáciu pre zvládnutie detailov konkrétneho programovacieho jazyka. Formálny model by mal byť teda dostatočne abstraktný, aby umožňoval pridávať ľubovolné želané jazykové konštrukcie. Povedzme si teraz o dvoch metódach, ktoré sú založené na transformáciách grafu reprezentujúceho zdrojový kód programu.

Refaktorovanie býva často špecifikované ako parametrizovaná transformácia programu so vstupnými a výstupnými podmienkami, ktoré, ak sú splnené, tak je zabezpečené zachovanie správania programu. Zdrojový kód programu potom môžeme reprezentovať pomocou grafu a refaktorovanie ako pravidlá

na jeho zmenu. Aplikovanie refaktorovania potom korešponduje transformácii grafu.

Väčšina softvérových nástrojov používa na reprezentovanie zdrojového kódu graf nazvaný abstraktný syntaktický strom (abstract syntax tree) [20]. Takáto reprezentácia obsahuje detaily o vykonávacej logike programu a je závislá od jazyka, v ktorom je program napísaný. To je v rozpore so snahou definovať refaktorovanie ako transformáciu nezávislú na programovacom jazyku.

Lepšou možnosťou, ako reprezentovať zdrojový kód programu, je použiť tzv. ľahký graf (lightweight graph) [21], ktorý je jazykovo nezávislý a zameriava sa len na kľúčové koncepty objektovo-orientovaného programovania ako sú triedy, metódy a premenné.

4.2.3 Aký je vplyv refaktorovania na kvalitu softvéru?

Pre ľubovolnú časť softvéru môžeme špecifikovať jej externé atribúty kvality ako napríklad robustnosť, výkonnosť, rozšíritelnosť alebo znovupoužiteľnosť. Refaktorovania môžu byť klasifikované podľa toho, ktoré z týchto atribútov kvality ovplyvňujú. To nám dovoľuje zvyšovať kvalitu softvéru aplikovaním relevantných refaktorovaní na správnych miestach. Na dosiahnutie tohto cieľa musíme analyzovať refaktorovania podľa ich účelu a efektu. Niektoré refaktorovania odstraňujú duplicitu kódu, ďalšie zlepšujú modularitu, iné zvyšujú úroveň abstrakcie a tak ďalej. Tento efekt môže byť odhadnutý na spoľahlivej úrovni podľa vnútorných atribútov kvality (zložitosť, previazanosť, kohéznosť), ktoré sú priamo ovplyvňované refaktorovaním.

Na meranie a odhadovanie dopadu refaktorovaní na kvalitu môže byť použitých mnoho techník [22], [23], ktoré bývajú založené na softvérových metrikách, štatistických postupoch alebo na empirických meraniach. Pozrime sa teraz, aký vplyv má refaktorovanie na spravovateľnosť a výkonnosť programu.

Spravovatelnosť verzus výkonnosť

Dôležitým atribútom kvality, ktorý je ovplyvňovaný väčšinou refaktorovaní, je výkonnosť programu. Existuje zaužívaný zlý názor, že zlepšenie štruktúry a tým aj spravovateľnosti programu má vždy negatívny vplyv na jeho výkonnosť. Napríklad v kontexte logických a funkcionálnych programov je typickým cieľom refaktorovania práve zlepšenie výkonnosti pri zachovaní sémantiky programu.

Nanešťastie vo väčšine prípadov má refaktorovanie naozaj negatívny dopad na výkonnosť programu. Väčšina refaktorovaní prináša do programov zložitejšie volanie metód, čo spôsobí vykonanie niekoľkých inštrukcií procesora navyše. Programátori teda argumentujú tým, že si nemôžu dovoliť refaktorovať, kvôli strate výkonnosti programu. Protiargumentom je tzv. pravidlo 80/20 [24], ktoré hovorí o tom, že 80% zdrojov využívaných programom, využíva len 20% zdrojového kódu programu. Teda, ak máme dobre štruktúrovaný program, je ľahké nájsť tých kritických 20% programu a zamerať sa na optimalizáciu ich výkonnosti. Toto pravidlo nemôžeme aplikovať na zariadenia, ktoré majú obmedzené energetické zdroje, lebo u takýchto zariadení treba ušetriť každú zbytočnú operáciu.

V niektorých prípadoch však dochádza k opačnej situácii a to, že refaktorovaný program je rýchlejší ako pôvodný. Z výskumu, ktorý spravil S. Demeyer [24], vyplýva, že nahradenie podmieňovacej logiky polymorfizmom, môže viesť k miernemu nárastu výkonu. Tento efekt je spôsobený efektívnejšími kompilátormi a novými technológiami v procesoroch.

4.2.4 Ako môžeme zachovávať konzistentnosť medzi jednotlivými softvérovými artefaktmi na rôznych úrovniach?

Refaktorovanie mení zdrojoví kód softvéru. Ale softvér je zložený aj z mnoho iných druhov artefaktov (analýza, architektúra, dizajn a ďalšie). Tvorcovia softvéru by sa mali snažiť medzi nimi udržovať konzistentnosť. Otázkou teda

je, akými metódami alebo technikami sa dá udržovať konzistentnosť medzi všetkými artefaktmi softvéru?

4.2.5 Ako môžeme refaktorovania kombinovať?

Vývoj softvérových nástrojov sa zaoberá aj možnosťami spájania refaktorovaní. Zložené refaktorovania poskytujú niekoľko výhod. Po prvé, dokážu lepšie zachytiť celkový zámer refaktorovania. Za ďalšie, poskytujú lepšiu výkonnosť, keďže stačí vstupné a výstupné podmienky skontrolovať len raz a netreba ich kontrolovať neustále ako pri postupnosti jednoduchých transformácií. A navyše sa môžu skladať aj zo zmien, ktoré nezachovávajú správanie programu, lebo stačí, ak ho zachová celé zložené refaktorovanie. Aby sme mohli vyvíjať nástroje pre zložené refaktorovania, je potrebné preskúmať, ako sa dajú spájať jednoduché refaktorovania.

5 Výskumné projekty

Výskum v oblasti refaktorovania v súčasnosti prebieha veľmi aktívne. Aj keď sa softvérové nástroje pre podporu refaktorovania vyvíjajú už dlhšiu dobu, stále ostáva veľa problémov, ktoré treba vyriešiť. V tejto kapitole uvedieme niektoré z aktuálne prebiehajúcich projektov zameraných na výskum reštrukturalizácie a refaktorovania.

5.1 Projekt refaktorovania jazyka C

(C Refactory Project)

Výskum refaktorovania na Univerzite v Illinois v USA (University of Illinois at Urbana-Champaign) prebieha už dlhú dobu [25]. W. Opdyke v čase, keď ako prvý zadefinoval pojem refaktorovania vo svojej PhD. práci, pracoval práve na tejto univerzite. Jeho výsledky potom použili D. Roberts a J. Brant, ktorí tiež pôsobili na tejto univerzite, na vytvorenie prvého nástroja na refaktorovanie nazvaného Refactoring Browser.

Ľudia

Výskum vedie doktor R. Johnson a pracuje na ňom jeho študentka A. Garrido. Celý projekt prebieha v rámci výskumnej skupiny doktora Johnsona nazvanej Software Architecture Group (SAG).

Projekt refaktorovania jazyka C			
Funkcia	**Osoba**	**Univerzita**	**Oddelenie**
Vedúci	Dr. Ralph Johnson	University of Illinois	Department of Computer Science
Výskumníci	Alejandra Garrido		

Motivácia

V súčasnosti už existujú nástroje na refaktorovanie objektovo-orientovaných jazykov ako sú Smalltalk alebo Java. Predpokladá sa však, že viac softvéru je napísaného v programovacom jazyku C, resp. C++. Napriek tejto skutočnosti

neexistuje nástroj na refaktorovanie jazyka C, ktorý by úplne podporoval aj direktívy pre preprocesor kompilátora.

Direktívy preprocesora sú konštrukty, ktoré umožňujú vkladanie a podmienené kompilovanie zdrojového kódu. Ďalej nám dovoľujú definovať makrá, ktoré sú preprocesorom pred skompilovaním nahradené definovaným kusom kódu jazyka C. Napriek tomu, že direktívy preprocesora môžu byť súčasťou zdrojového kódu jazyka C, tak nie sú jeho platným zdrojovým kódom. Pri kompilácii kódu C programov, najskôr preprocesor spracuje všetky direktívy na kód jazyka C, a potom takto upravený zdrojový kód odovzdá kompilátoru.

Súčasné nástroje na refaktorovanie C programov dokážu pracovať len s preprocesorom spracovaným zdrojovým kódom, t.j. kódom bez direktív. To prináša niekoľko nevýhod. Refaktorovania, ktoré ponúkajú musia byť značne obmedzené, aby sa zachovalo správanie programu. Navyše, keďže pracujú so zmeneným kódom programu, tak nemôžu byť súčasťou integrovaného vývojového prostredia (Integrated Development Environment - IDE). Toto spôsobuje aj problémy s vizualicáciou, lebo čistý kód C programu sa môže veľmi líšiť od pôvodného zdrojového kódu.

Ciele

R. Johnoson a A. Garrido tvrdia, že direktívy preprocesora sú veľmi často používané v programoch v jazyku C, a preto nástroj, ktorý by ich pri refaktorovaní plne podporoval, je veľmi dôležitý. Navyše pre podporu direktív preprocesora je podľa nich potrebné vytvoriť nové refaktorovania.

Tento projekt je zameraný na vytvorenie softvérového nástroja s úplnou podporou refaktorovania jazyka C. To vyžaduje vytvorenie nového spôsobu parsovania jazyka C bez použitia preprocesora. Týmto spôsobom bude možné reprezentovať a transformovať C program, ktorý obsahuje aj s direktívy preprocesora.

5.2 Projekt refaktorovania

(Refactoring Project)

Jeden z aktuálne prebiehajúcich projektov [26], vznikol koncom roka 2002 na Univerzite v Antwerpách (University of Antwerp) a na Slobodnej Univerzite v Bruseli (Vrije University of Brussel) v Belgicku.

Ľudia

Tím pracujúci na projekte je zložený z výskumníkov a promotérov projektu, z ktorých väčšina pôsobí na Univerzite v Antverpách.

Projekt refaktorovania			
Funkcia	**Osoba**	**Univerzita**	**Oddelenie**
Promotéri	prof. dr. Dirk Janssens	University of Antwerp	DMCS (FOST)
	prof. Serge Demeyer		DMCS (LORE)
	postdoct. študent Tom Mens	Vrije University of Brussel	Programming Technology Lab
Výskumníci	PhD. študent Bart Du Bois	University of Antwerp	DMCS (LORE)
	PhD. študent Hans Stenten		DMCS (LORE)
	PhD. študent Niels Van Eetvelde		DMCS (FOST)
	PhD. študent Pieter Van Gorp		DMCS (FOST)
	PhD. študent Alan Amsel		DMCS (FOST)

DMCS - Department of Mathematics and Computer Science

LORE - The Lab on Reengineering

FOST - Formal Techniques in Software Engineering

Motivácia

Motiváciou ľudí pracujúcich na tomto projekte je nejednotnosť a rozdrobenosť vo využívaní dosiahnutých výsledkov v oblasti refaktorovania. Ako uvádzajú v opise svojho projektu, tak napriek pokroku vo výskume a veľkému rozvoju nástrojov na refaktorovanie, používa väčšina dnes dostupných riešení len obmedzenú množinu refaktorovaní a je určená pre jeden programovací jazyk. Je viac-menej jasné, že nájdené riešenia by mohli byť použité aj v kontexte iných refaktorovaní, problémov alebo jazykov. Formálnejším prístupom je

možné presne opísať nevyriešené sporné body a získať jasnejší obraz o možnostiach a obmedzeniach refaktorovania.

Ciele

Projekt sa snaží poskytnúť ucelený podklad pre vývoj vhodného formálneho modelu pre refaktorovanie softvéru. Zameraný je na skúmanie zjednodušeného modelu, umožňujúceho štúdium základných vlastností refaktorovania, ako aj dizajn nástrojov podporujúcich proces refaktorovania. Ako základ pre takýto model sa skúma technika založená na prepisovaní (transformáciách) grafov. Existuje niekoľko príčin, kvôli ktorým sa predpokladá, že táto technika bude dobrým základom pre vybudovanie vhodného modelu. Po prvé, použitie grafov dáva možnosť pre vyjadrenie mnoho druhov schematických reprezentácií prirodzeným spôsobom. Po druhé, existujúce výsledky týkajúce sa reprezentácie a analýzy procesu prepisovania grafov poskytujú dobré východisko pre opis a manipuláciu zložených refaktorovaní. To môže viesť napríklad k metódam pre detekciu konfliktov alebo pre optimalizáciu refaktorovaní. Po tretie, otázka, či refaktorovanie zachováva správanie programu, je ekvivalentná otázke, či prislúchajúca transformácia grafu, zachováva potrebné vlastnosti grafu. Analogicky môžeme skúmať aj zložitosť refaktorovaní, ako množstvo krokov potrebných pre pripisovanie grafu.

Cieľom projektu je postupne zodpovedať nevyriešené problémy výskumu, vytvoriť formálne základy pre refaktorovanie a zlepšiť podporu refaktorovania softvérovými nástrojmi, teda prispieť k jeho štandardizácii v softvérovom inžinierstve.

5.3 Jazykovo-parametrická reštrukturalizácia programov

(Language-Parametric Program Restructuring)

Projekt [27] bol spustený v júly roku 2003 Holandskou Organizáciou pre Vedecký Výskum (The Netherlands Organisation for Scientific Research)

a prebieha na Slobodnej Univerzite v Amsterdame (Vrije University of Amsterdam) a v Centre pre Matematiku a Informatiku (Centrum voor Wiskunde en Informatica - CWI) v Holandsku.

Ľudia

Výskumný tím je rozdelený do dvoch skupín podľa miesta pôsobenia. V každej je vedúcim tímu významný profesor. Prvá skupina, vedená profesorom P. Klintom, sa špecializuje na interaktívny vývoj a renováciu softvéru. Druhú skupinu vedie profesor C. Verhoef a je zameraná na softvérové inžinierstvo a správu softvéru. Vedúci výskumu celého projektu je profesor C. Verhoef.

Jazykovo-parametrická reštrukturalizácia programov			
Funkcia	**Osoba**	**Univerzita**	**Oddelenie**
Vedúci skupiny	prof. dr. Paul Klint	CWI	Department of Software Engineering
Výskumníci	dr. Jan Heering		
	dr. Mark van den Brand		
Vedúci skupiny	prof. dr. Chris Verhoef	University of Amsterdam	Faculty of Sciences - Department of Computer Science
Výskumníci	dr. Ralf Lämmel		
	PhD. študent Vadim V. Zaytsev		

Motivácia

Motiváciou ľudí pracujúcich na tomto projekte, je len pomaly sa zlepšujúca situácia v oblasti reštrukturalizácie programov. Napriek neustále prebiehajúcemu výskumu je situácia len o niečo lepšia ako pred pár rokmi. Máme vedomosti o základných stavebných blokoch pre softvérové prostredia umožňujúce refaktorovania. Tie sú však väčšinou navrhnuté pre špecifický prípad a programovací jazyk. To je v kontraste so želanou situáciou, v ktorej by takéto prostredia boli poskladané zo znovupoužiteľných blokov, ktoré by vždy stačilo len použiť v správnom tvare v zmysle modularity programu.

Ciele

Cieľom projektu je identifikácia a zorganizovanie pojmov reštrukturalizácie, ktoré boli používané v rôznych scenároch a jazykoch. Kým minulé projekty boli zamerané na konkrétne problémy a programovacie jazyky, tento projekt sa snaží o všeobecný prístup, ktorý od nich abstrahuje a pokúsi sa zjednotiť kľúčové pojmy.

Z pohľadu výskumu nie je nič horšie ako teória, ktorá nie je používaná, hoci potenciálne možnosti jej použitia sú už známe. Analogicky, z pohľadu praxe je softvérové inžinierstvo ako aplikovaná disciplína len málo ovplyvňované výskumom. Tento projekt je zameraný na zmenšovanie rozdielov medzi teóriou a praxou.

5.4 Dolovanie aspektov a refaktorovanie

(Aspect Mining and Refactoring)

Projekt [28] prebieha na Delfskej Univerzite Technológií (Delft University of Technology) v Holandsku v Laboratóriu pre Výskum Softvérovej Evolúcie (Software Evolution Research Laboratory - SWERL) a spustený bol 15. septembra 2003.

Ľudia

Projekt vedie profesor A. van Deursen a na projekte úzko spolupracujú s Centrom pre Matematiku a Informatiku (CWI) v Holandsku.

Dolovanie aspektov a refaktorovanie			
Funkcia	**Osoba**	**Univerzita**	**Oddelenie**
Vedúci	prof. Arie van Deursen	Delft University of Technology	Software Evolution Research Laboratory
Výskumníci	asist. prof. Leon Moonen		
	postdoc. študent Hylke van Dijk		
	PhD. študent Marius Marin		
	PhD. študent Bas Graaf		
	PhD. študent Marco Lormans		

Motivácia

Dekompozícia veľkého celku na menšie časti je v softvérovom inžinierstve hlavný spôsob, ako zvládnuť vývoj a správu zložitých systémov. Výsledkom dekompozície je rozdelenie systému na oddelené celky, uľahčenie tímovej práce, lokálnych zmien, testovania a plánovania práce. Existujú však aj funkcie systému, ktoré sú ťažko oddeliteľné do samostatných celkov. Sú to funkcie ako napríklad ošetrovanie a zaznamenávanie chýb. Práve takéto funkcie sa označujú ako aspekty programu alebo ako „presahujúce časti“ programu (cross cutting concerns).

Aspektovo-orientované programovanie vzniklo ako paradigma, ktorá sa zameriava na presné definovanie takýchto aspektov a použitie techník generovania kódu na zabudovanie aspektov späť do aplikačnej logiky. Toto môže vyriešiť problémy vnútornej rozdrobenosti a spletitosti kódu.

Výsledky tohto projektu môžu byť použité na podporu vývoja softvéru, priebežnú analýzu systémov počas ich budovania a na odhaľovanie aspektov, ktoré je potrebné refaktorovať.

Ciele

Zámerom ľudí pracujúcich na tomto projekte je vytvoriť koncepty, metódy a nástroje pre vyhľadávanie aspektov v existujúcich objektovo-orientovaných systémoch. Výskum má ďalej zahŕňať vytvorenie techniky na automatickú analýzu objektovo-orientovaných systémov a overenie dosiahnutých výsledkov na skutočných softvérových systémoch. Vo vzťahu k refaktorovaniu bude treba zodpovedať najmä otázku využitia identifikovaných aspektov na určenie potrebných refaktorovaní.

5.5 Refaktorovanie zabudovaných softvérových kompo – nentov vo všadeprítomných prostrediach

(Refactoring Embedded Software Components for Ubiquitous Environments)

Projekt [29] prebieha na Univerzite vo Viktórií (University of Victoria) v Kanade na oddelení počítačovej vedy. Skrátene sa označuje ako RESCUE projekt a je súčasťou Konzorcia pre Výskum Softvérového Inžinierstva (Consortium for Software Engineering Research). Ďalším výskumným partnerom na tomto projekte je Herzbergov Inštitút Astrofyziky (Herzberg Institute for Astrophysics), ktorý poskytuje priemyselné prípadové štúdie zabudovaných systémov použitých vo vysoko výkonných rádioteleskopoch.

Ľudia

Výskum vedie profesor P. de Souza a spolupracuje na ňom aj s organizáciou Softvérové Laboratórium IBM v Ottawe (The IBM Ottawa Software Lab) a spoločnosťou Intec Innoventures Inc.

Refaktorovanie zabudovaných softvérových komponentov vo všadeprítomných prostrediach			
Funkcia	**Osoba**	**Univerzita**	**Oddelenie**
Vedúci	prof. Phillip de Souza	University of Victoria	Department of Computer Science
Výskumníci	prof. dr. Jens H. Jahnke		
	prof. Marc d'Entremont		

Motivácia

Veľká časť používaného softvéru je zabudovaná v elektronických a mechanických prístrojoch. Spoločnosť sa postupne stáva závislou od takýchto zabudovaných softvérových systémov. Často totiž vykonávajú kriticky dôležité funkcie a životy ľudí môžu závisieť od ich správneho fungovania a spoľahlivosti. Pokračujúci proces miniaturizácie hardvéru dáva nové možnosti pre sofistikovanejšie a zložitejšie softvérové aplikácie, ktoré môžu byť doňho zabudované.

Refaktorovanie zabudovaného softvéru je ťažšie ako refaktorovanie vo všeobecnosti a to kvôli možným závislostiam od prostredia, v ktorom sa softvér nachádza. Niekedy môže byť o veľa ekonomickejšie nahradiť niektoré časti novými softvérovými komponentmi a iné, špecifické časti refaktorovať, aby sa dali použiť aj v novom systéme.

V súčasnosti už existuje veľké množstvo zabudovaných aplikácií a mnohé z nich pracujú na zastaraných hardvérových a softvérových platformách. Rastúci tlak na ich modernizáciu zväčšuje prostriedky, ktoré sú investované do výskumu zabudovaných softvérových komponentov. Vznikajú najmä požiadavky na možnosti spolupráce aplikácií v počítačových sietiach a na väčšiu modularitu softvéru.

Ciele

RESCUE projekt skúma praktické metódy, procesy a nástroje pre transformovanie zastaralých zabudovaných softvérových systémov na moderné komponentové modely. Takáto technológia umožní spoločnostiam zmodernizovať a používať ďalej existujúci softvér, zvýšiť produktivitu a kvalitu vývoja zabudovaných aplikácií.

6 Podpora refaktorovania softvérovými nástrojmi

Manuálne refaktorovanie programu je náchylné k chybám a pomerne nešikovné. Pri rozsiahlych programoch býva často aj nemožné. Riešením tohto problému je vytvorenie nástroja, ktorý by pomáhal programátorom refaktorovať programy. Takýto softvérový nástroj by skontroloval nutné podmienky pre použitie refaktorovania, analyzoval by syntax a sémantiku programu a prezentoval by výsledok náležitým spôsobom. Potom by sa programátor rozhodol, či refaktorovať program alebo nie. Nakoniec, po refaktorovaní programu, by sa automaticky spustili testy, ktoré by overili, či sa sémantika programu nezmenila.

Softvérové nástroje umožňujúce refaktorovanie toto všetko zatiaľ nedokážu a sú len pomocníkmi, ktorí však výrazne redukujú čas a úsilie potrebné na vykonanie refaktorovania. V súčasnosti je hlavným cieľom nástrojov na refaktorovanie umožniť programátorovi refaktorovať kód programu efektívnym spôsobom bez obáv, že by poškodil existujúci program. Vo väčšine dostupných softvérových nástrojov je implementovaná len limitovaná množina refaktorovaní pre nejaký konkrétny jazyk.

V tejto kapitole si povieme o kritériách, ktoré by mali spĺňať softvérové nástroje určené na vykonávanie refaktorovaní a potom si porovnáme možnosti vybraných súčasných softvérových nástrojov.

6.1 Kritéria pre softvérové nástroje

Aby bol nástroj podporujúci refaktorovanie úspešný, mal by byť v prvom rade ľahko používateľný. Keďže manuálne refaktorovanie programu je časovo náročný proces, tak jedným z hlavných kritérií je to, aby vykonanie refaktorovania pomocou softvérového nástroja urýchlilo celý proces refaktorovania.

Výhodou je aj, ak je nástroj integrovaný do vývojového prostredia (IDE) alebo editora zdrojového kódu. Softvérový nástroj by mal ďalej obsahovať aj systém pre správu verzií súborov vyvíjaného softvéru (Concurrent Versioning System - CVS), ktorý je potrebný, ak chceme zrušiť vykonané úpravy. V praxi sa totižto často stáva, že refaktorovanie, ktoré sme spravili pred pár dňami, už dnes nie je dobré a potrebujeme ho zrušiť. Práve CVS nám umožňuje vrátiť zmenené súbory do pôvodného stavu.

K najdôležitejším vlastnostiam nástroja na refaktorovanie patrí jeho schopnosť zabezpečiť zachovanie požadovanej vlastnosti správania programu. Ako sme už spomínali, toto nie je možné zabezpečiť napríklad pri časovo kritických (time critical) aplikáciách

Softvérový nástroj na refaktorovanie má ďalej obsahovať databázu všetkých programových elementov (premenných, metód a tried) a umožňovať pre ľubovolný z nich zistiť jeho previazanosť s ostatnými. Samozrejme, že po každej zmene v zdrojovom kóde, sa musí táto databáza automaticky aktualizovať.

6.2 Porovnanie softvérových nástrojov

V tejto časti porovnáme päť softvérových nástrojov, ktoré umožňujú refaktorovať zdrojový kód. Pri porovnávaní sa zameriame hlavne na použiteľnosť nástrojov z pohľadu užívateľa. Nebudeme sa zaoberať implementačnými a technickými detailmi. Porovnávať budeme nasledovné nástroje: VisualWorks, Eclipse, IntelliJ Idea, SlickEdit a ReSharper. V nasledujúcich častiach si opíšeme ich hlavné črty a funkcie. Kompletný zoznam refaktorovaní, implementovaných jednotlivými nástrojmi, sa nachádza v prílohe.

6.2.1 VisualWorks 7.3 [31]

VisualWorks je integrované vývojové prostredie vytvorené spoločnosťou Cincom pre programovací jazyk Smalltalk, ktoré má integrovanú kompletnú sadu nástrojov na refaktorovanie zo známeho softvérového nástroja Refactoring Browser [32]. VisualWorks sa zameriava len na proces písania kódu programu, a preto je jeho užívateľom zvyčajne programátor.

Ďalej VisualWorks obsahuje nástroj na grafické zobrazenie hierarchie tried, ktoré je možné použiť pri vyhľadávaní problémov v zdrojovom kóde. Nemá vlastné CVS, ale ponúka možnosť zrušenia zmeny (undo) do užívateľom nastavenej hĺbky alebo ho je možné použiť spolu s nejakým externým CVS.

VisualWorks obsahuje dynamickú kontrolu vstupných a výstupných podmienok, ktoré zaručujú zachovanie správania vyvíjaného programu. Jediné, čo musí pri refaktorovaní spraviť užívateľ, je zadanie parametrov refaktorovania.

Refaktorovania, ktoré ponúka VisualWorks sú zamerané najmä na pridávanie, mazanie a modifikáciu programových elementov. Aby sa zachovala sémantika programu, tak sú tieto refaktorovania často používané spolu ako zložené transformácie. Rozhodnutie, kde a ako použiť refaktorovanie je na užívateľovi. Kontrola vstupných a výstupných podmienok a samotné refaktorovanie je potom spravené automaticky.

VisualWorks je implementovaný a pracuje v prostredí programovacieho jazyka Smalltalk, ktoré využíva dynamickú kompiláciu. Keďže refaktorovanie nemení správanie a počas vývoja by sa nemala odoberať funkčnosť, ale len pridávať, tak systém môže byť menený aj počas behu. Tento nástroj je preto odporúčaný pre projekty, ktoré sa potrebujú vyvíjať dynamicky.

Autori tohto nástroja vytvorili skelet (framework), ktorý povoľuje dodefinovanie nových refaktorovaní, ktoré sú založené na existujúcich. Ako však sami uvádzajú je príliš zložitý a vyžaduje si rozsiahle znalosti o refaktorovaní. Preto si za cieľ do budúcnosti dávajú vytvorenie grafického rozhrania, ktoré umožní definíciu nových refaktorovaní.

6.2.2 Eclipse 3.0 [33]

Cieľom projektu Eclipse je vytvoriť mulitplatformové integrované vývojové prostredie, ktoré môže byť použité na vytváranie rôznych druhov aplikácii (internetové stránky, Java programy, C++ programy, ...). Eclipse má otvorenú architektúru, založenú na moduloch, ktoré sa dajú doňho ľahko pridávať. Vďaka tejto architektúre je ľahko rozšíriteľný o podporu pre nové programovacie jazyky a rôzne nástroje. My sa ďalej zameriame na modul pre programovací jazyk Java (Java Development Tool - JDT), ktorý je do Eclipsu štandardne vložený.

Eclipse je tiež zameraný len na jeden softvérový artefakt, a tým je opäť zdrojový kód. Užívateľom je zvyčajne programátor alebo tester. Eclipse obsahuje nástroj na distribuovanú správu verzií súborov. Projekty, súbory a adresáre sú ukladané v jednom alebo viacerých skladoch (repository), ktoré sa môžu nachádzať aj na iných počítačoch. Užívateľ môže pracovať z viacerými skladmi naraz, čo umožňuje paralelnú prácu na projektoch.

Programátori pracujúci s JDT môžu meniť rozsiahle systémy bez potreby prekompilovania celého systému odznova. Umožňuje to nástroj nazvaný Java Model, ktorý obsahuje programové elementy uložené v stromovej štruktúre. Pri vytváraní novej verzie systému sú potom prekompilované len zmenené súbory a súbory od nich závislé. Tento postup sa zvykne označovať ako inkrementálne vytváranie projektu (incremental project build).

Eclipse prezentuje refaktorovania tak, že programátor nepotrebuje veľké znalosti o refaktorovaní. Programátor musí len označiť časť kódu programu a vybrať refaktorovanie, ktoré sa má vykonať. Pri refaktorovaní programu môžeme v Eclipse používať aj zabudovaný CVS a knižnice na testovanie, ktoré obsahuje. Na rýchlejšie nasadzovanie refaktorovaných programov do praxe môžeme využívať spomínaný nástroj Java Model.

6.2.3 IntelliJ Idea [34]

Idea je integrované vývojové prostredie vyvíjané spoločnosťou JetBrains pre programovací jazyk JAVA, ktorý okrem nástroja na refaktorovanie obsahuje viac ako 20 ďalších nástrojov. Architektúra celého nástroja je otvorená a umožňuje ľahké pridávanie nových nástrojov. Z hľadiska množstva funkcií, ktoré tento softvér ponúka, patrí určite k najlepším. My si teraz povieme o tých, ktoré sa dajú využiť v procese refaktorovania.

Idea má množstvo funkcií, ktoré umožňujú rýchle programovanie softvéru. Napriek tomu nie je výhradne určená len pre programátorov. Využiť sa dá aj pri revízií zdrojového kódu, pre ktorú sú určené nástroje na inšpekciu a analýzu kódu. V nich je zabudovaná aj funkcia na vyhľadávanie duplicitného kódu, ktorá hľadá rovnaké alebo podobné riadky v zdrojovom kóde programu. Táto funkcia spolu s nástrojom na analýzu závislostí súborov, knižníc a tried sa dajú veľmi dobre využiť pri hľadaní páchnucich častí zdrojového kódu.

Ďalším z nástrojov, ktoré IDEA obsahuje, je prepracovaný CVS systém, ktorý umožňuje vizuálne porovnávanie a integráciu súborov. Na kontrolu a testovanie naprogramovaného kódu má IDEA v sebe zabudovanú knižnicu JUNIT.

Idea obsahuje až 34 druhov refaktorovaní zdrojového kódu. K netradičným refaktorovaniam, ktoré Idea ponúka, patrí napríklad nahradenie duplicitného kódu metódou.

6.2.4 SlickEdit v10 [35]

SlickEdit je tiež multiplatformový softvérový nástroj. Vyvíjaný je však ako editor, ktorý je možné použiť v iných vývojových prostrediach, ako je napríklad Eclipse. Keďže to nie je vývojové prostredie, ale len editor, tak jeho funkčnosť je zameraná len na prácu so zdrojovým kódom, a preto neobsahuje nástroje ako kompilátor, či debugger. Celkovo SlickEdit podporuje až 46 programovacích jazykov

Napriek tomu, že SlickEdit je len editor, poskytuje dostatok funkcií, ktoré sa dajú využiť pri refaktorovaní. Zabudovaný má vlastný CVS, funkciu na automatické formátovanie zdrojového kódu a knižnicu na testovanie programov JUNIT.

Okrem prepracovaného a flexibilného užívateľského rozhrania, obsahuje aj 18 refaktorovaní pre programovací jazyk C++. Nevýhodou je, že refaktorovania nepodporujú direktívy kompilátora. Čiže pred refaktorovaním musí byť zdrojový kód spracovaný preprocesorom jazyka C++.

6.2.5 ReSharper 1.5 [36]

Resharper je modul do vývojového prostredia VisualStudio .NET, ktorý umožňuje refaktorovať zdrojový kód jazyka C# (C sharp). Napriek tomu, že Resharper nie je samostatný softvér, ponúka dostatok zaujímavých funkcií. K hlavným funkciám, okrem refaktorovaní, patrí zvýrazňovanie syntaktických chýb v kóde bez nutnosti jeho kompilácie a prehľadná navigácia v zdrojovom kóde programu.

ReSharper poskytuje 17 refaktorovaní. Okrem tých často používaných refaktorovaní, obsahuje aj niektoré zaujímavé, ako je napríklad zmena abstraktnej triedy na rozhranie a naopak zmena rozhrania na abstraktnú triedu.

6.2.6 Zhrnutie

Opísané nástroje boli vybrané kvôli zrejmým rozdielom v cieľoch, na ktoré sú určené. Napriek týmto rozdielom môžeme povedať, že obsahujú mnoho rovnakých funkcií a smer ich vývoja je veľmi podobný. V opise funkcií a nástrojov, ktoré poskytujú sme sa zamerali najme na tie, ktoré sa dajú využiť pri procese refaktorovania (CVS, testovanie, ...).

Z pohľadu užívateľa sa najmä komerčné produkty snažia o jednoduché užívateľské rozhranie a intuitívne ovládanie. Štandardom sa postupne stávajú funkcie na automatické formátovanie a dopĺňanie zdrojového kódu a na odhaľovanie syntaktických chýb v zdrojovom kóde už počas jeho písania.

Porovnávané softvérové nástroje sú zamerané hlavne na fázu písania zdrojového kódu. Ich užívateľmi sú preto najmä programátori. Výnimkou je len Idea, ktorá poskytuje navyše aj funkcie na analýzu a revíziu kódu. Nástroje Eclipse, Idea a SlickEdit sa dajú využiť aj počas testovania funkčnosti programu.

Z hľadiska spravovania verzií zdrojového kódu neobsahujú vlastný CVS systém len nástroje VisualWorks a ReSharper. VisualWorks však môže byť použitý s externým CVS systémom a pre ReSharper je uvažovať o CVS irelevantné, keďže je to len modul do vývojového prostredia.

Vybrané softvérové nástroje sú z pohľadu aplikovania refaktorovania považované za poloautomatické. To znamená, že užívateľ označí kus zdrojového kódu, vyberie refaktorovanie a zadá požadované parametre. Vykonanie refaktorovania už potom zabezpečí softvérový nástroj. Každý z porovnávaných nástrojov má implementovaný dostatok rôznych refaktorovaní. Všetky majú implementované základné refaktorovania na úpravu metód, tried a premenných.

Porovnanie vlastností a funkcií					
Softvérový nástroj	**VisualWorks 7.3**	**Eclipse 3.0 (JDT)**	**IntelliJ Idea 4.5**	**Slick Edit 10.0**	**ReSharper 1.5**
Výrobca	Cincom	Eclipse.org	JetBrains	SlickEdit Inc.	JetBrains
Licencia	Komerčný produkt	Zdarma	Komerčný produkt	Komerčný produkt	Komerčný produkt
Typ aplikácie	IDE	IDE	IDE	Editor alebo modul do Eclipse	Modul do VisualStudio .NET
Podporované Platformy*	Windows, Mac OS, Intel Linux, UNIX	Windows, Linux, Solaris, AIX, HP-UX, Mac OS	Windows, Linux, Generic Unix, Mac OS	Windows, Linux, AIX, HP-UX, IRIX, Solaris, Mac OS	Windows
Podporované programovacie jazyky	Smalltalk	možnosť pridávania	Java	46 programovacích jazykov	C#
Programovacie jazyky s podporou refaktorovania	Smalltalk	Java	Java	C++	C#
Užívateľ	programátor	programátor, tester	programátor, dizajnér, tester	programátor, tester	programátor
CVS	externý	áno	áno	áno	irelevantné
Potreba zastavenia pri zmene verzie	nie	áno	áno	áno	áno
Podpora testovania	nie	áno	áno	áno	nie
Stupeň automatizácie refaktorovania	polo-automatické	polo-automatické	polo-automatické	polo-automatické	polo-automatické
Počet podporovaných refaktorovania	29	22	34	18	17
Meranie vplyvu refaktorovania	nie	nie	čiastočne	nie	nie

Pri nasadzovaní zmeneného programu do praxe jedine VisualWorks nemusí vyžadovať zastavenie vyvíjanej aplikácie. Spôsobené to však je prostredím Smalltalku a nie samotným nástrojom. U ostatných nástrojov je potrebné vyvíjanú aplikáciu pri zmene verzie zastaviť.

Ani jeden z porovnávaných nástrojov neposkytuje spôsob, ako zistiť vplyv refaktorovania na zdrojový kód. Jedine Idea umožňuje zistiť aspoň previazanosť tried a metód. Vo všeobecnosti však treba použiť na tento účel nejaký externý nástroj.

Na záver môžeme povedať, že všetky opísané nástroje sú zamerané najmä na aplikovanie refaktorovania v kóde programu. Chýba im však podpora pre ďalšie činnosti súvisiace s refaktorovaním, ako je napríklad vyhľadávanie páchnucich častí kódu alebo analýza vplyvu refaktorovania na dizajn a štruktúru zdrojového kódu.

7 Záver

Cieľom tejto diplomovej práce bolo vytvoriť monografiu o refaktorovaní, čiže zhrnúť teoretické a praktické výsledky výskumu refaktorovania. V práci som sa snažil o čitateľnosť a ľahkú pochopiteľnosť aj bez hlbších znalostí softvérového inžinierstva. Od čitateľa sa však požadovali aspoň základné vedomosti o objektovo-orientovanom programovaní a UML diagramoch.

V diplomovej práci som snažil pozrieť sa na problematiku refaktorovania z rôznych uhľov. V prvých kapitolách práce som sa zameral hlavne na vysvetlenie podstaty refaktorovania a problémov, ktoré sa snaží riešiť. V ďalšej kapitole som sa sústredil na výskum refaktorovania, na otvorené otázky a problémy. Na túto časť nadväzuje kapitola o aktuálnych výskumných projektoch, v ktorej som sa snažil priblížiť motiváciu ich snaženia a objasniť ciele ich výskumu. V poslednej kapitole som sa snažil čitateľovi ukázať dosiahnuté praktické výsledky výskumu refaktorovania. Na tento účel som vybral niekoľko softvérových nástrojov určených na refaktorovanie a porovnal ich funkčnosť z pohľadu užívateľa.

Keďže hlavne v období posledných rokov sa výskum refaktorovania zintenzívňuje, nebolo v tejto práci možné spomenúť všetky metódy, techniky ani nástroje určené pre refaktorovanie. Do tejto práce som sa snažil vybrať tie, ktoré som pokladal za najviac preskúmané a rozpracované.

Literatúra

1. Martin Fowler. Refactoring: Improving the Design of Existing Programs. Addison-Wesley Professional, 1999.

2. Tom Mens. A Survey of Software Refactoring. IEEE Transactions On Software Engineering, vol. 30, no. 2, 2004

3. Elliot J. Chikofsky, James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.

4. Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann and DPunkt, 2002.

5. William Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992. ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z

6. Kent Beck, Cynthia Andres. Extreme Programming Explained : Embrace Change. Addison-Wesley Professional, 2004.

7. Mika Mäntylä. Bad Smells in Software – a Taxonomy and an Empirical Study. Master's Thesis, Helsinki University Of Technology, 2003.

8. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley Professional, 1995.

9. John Brant, Don Roberts, Ralph Johnson. Refactoring Browser Tool. http://st-www.cs.uiuc.edu/users/brant/Refactory/

10. Arie van Deursen, Leon Moonen, Alex van den Bergh, Gerard Kok. Refactoring Test Code. The second International Conference on eXtreme Programming and Flexible Processes in Software Engineering, 2001.

11. Ward Cunningham. Wiki Pages About Refactoring. http://c2.com/cgi/wiki?WikiPagesAboutRefactoring

12. Bart Du Bois, Pieter Van Gorp, Alon Amsel, Niels Van Eetvelde, Hans Stenten, Serge Demeyer, Tom Mens. A Discussion of Refactoring in Research and Practice. Technical report, University of Antwerp, 2003

13. Tom Mens, Serge Demeyer, Bart Du Bois, Pieter Van Gorp, Hans Stenten. Refactoring: Current Research and Future Trends. Third Workshop on Language Descriptions, Tools and Applications, 2003.

14. Martin Fowler. UML Distilled. Addison-Wesley Professional, 3rd Edition, 2003.

15. Jens U. Pipka. Refactoring in a 'Test First'-World. Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng., 2002.

16. Lance Tokuda, Don S. Batory. Evolving Object-Oriented Designs with Refactorings. Automated Software Eng., vol. 8, no. 1, pp. 89-120, 2001.

17. Yoshio Kataoka, Michael D. Ernst, William G. Griswold, David Notkin. Automated Support for Program Refactoring Using Invariants. Proc. Int'l Conf. Software Maintenance, pp. 736-743, 2001.

18. Matthias Rieger, Stéphane Ducasse. Visual Detection of Duplicated Code. Software Composition Group, University of Berne, 1997.

19. Frank Simon, Frank Steinbrückner, Claus Lewerentz. Metrics Based Refactoring. Proc. European Conf. Software Maintenance and Reeng., pp. 30-38, 2001.

20. P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated Distributed Diagram Transformation for Software Evolution. Electronic Notes in Theoretical Computer Science, vol. 72, no. 4, 2002.

21. Tom Mens, Serge Demeyer and D. Janssens. Formalising Behavior Preserving Program Transformations. Graph Transformation, 2002.

22. Yoshio Kataoka, T. Imai, H. Andou, T. Fukaya. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. Proc. Int'l Conf. Software Maintenance, pp. 576-585, Oct. 2002.

23. Ladan Tahvildari, Kostas Kontogiannis. A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph. Proc. Working Conf. Reverse Eng., pp. 77-86, Oct. 2002.

24. Serge Demeyer. Maintainability versus Performance: What's the Effect of Introducing Polymorphism? Technical report, Lab. On Reeng., Universiteit Antwerpen, Belgium, 2002.

25. C Refactory Project
https://netfiles.uiuc.edu/garrido/www/CRefactory.html

26. Refactoring Project.
http://www.win.ua.ac.be/~lore/refactoringProject/

27. Language-Parametric Program Restructuring.
http://www.cs.vu.nl/lppr/

28. Aspect Mining and Refactoring.
http://swerl.tudelft.nl/bin/view/AMR/WebHome

29. Refactoring Embedded Software Components for Ubiquitous Environments.
http://strongbow.cs.uvic.ca/Projects/rescue/

30. Tom Mens, Jocelyn Simmonds, E. des M. de Nantes. A Comparison of Software Refactoring Tools. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel, 2002.

31. Cincom: Smalltalk VisualWorks.
http://smalltalk.cincom.com

32. Refactory Inc.: Refactory Browser.
http://www.refactory.com/RefactoringBrowser/

33. Eclipse.org: Eclipse.
http://www.eclipse.org

34. JetBrains: IntelliJ Idea.
http://www.jetbrains.com/idea/

35. SlickEdit Inc.: SlickEdit.
http://www.slickedit.com/

36. JetBrains: ReSharper.
http://www.jetbrains.com/resharper/

Zoznam refaktorovaní

V tejto prílohe je zoznam refaktorovaní, ktoré sú implementované v softvérových nástrojoch porovnávaných v 6. kapitole.

VisualWorks 7.3 [31]

1. Abstract Instance / Class Variable
2. Add Instance / Class Variable
3. Add Parameter
4. Convert To Instance Variable
5. Convert To Sibling Class
6. Convert To ValueHolder
7. Create Instance / Class Variable Accessors
8. Create Subclass
9. Extract Method
10. Extract To Temporary
11. Inline All Self Sends
12. Inline Message
13. Inline Parameter
14. Inline Temporary Variable
15. Move To Component
16. Move To Inner Scope
17. Protect / Concrete Instance Variable
18. Pull Up Instance / Class Variable
19. Push Down Instance / Class Variable
20. Push Down Method
21. Push Up Method
22. Remove Instance / Class variable
23. Remove Parameter
24. Rename Class
25. Rename Instance / Class variable
26. Rename Method
27. Rename Temporary
28. Safe Remove Class
29. Safe Remove Method

Eclipse 3.0 (JDT) [33]

1. Change method signature
2. Convert Anonymous Class To Nested
3. Convert Local Variable To Field
4. Encapsulate Field
5. Extract Constant
6. Extract Interface

7. Extract Local Variable
8. Extract Method
9. Generalize Type
10. Inline Field
11. Inline Method
12. Introduce Factory
13. Introduce Parameter
14. Move Class
15. Move Member Type To New File
16. Move Method
17. Push Down Subclass
18. Push Up Superclass
19. Rename Class
20. Rename Field
21. Rename Method
22. Use SuperType where Possible

IntelliJ Idea 4.5 [34]

1. Convert Anonymous Class To Inner
2. Convert To Instance Method
3. Copy / Clone Class
4. Encapsulate Fields
5. Extract Interface
6. Extract Method
7. Extract Subclass
8. Extract Superclass
9. Change method signature
10. Inline Constant
11. Inline Local Variable
12. Inline Method
13. Introduce Constant
14. Introduce Field
15. Introduce Parameter
16. Introduce Variable
17. Make Method Static
18. Move Class
19. Move Inner Class To Upper Level
20. Move Package
21. Move Static Member
22. Pull Members Up
23. Push Members Down
24. Rename Class
25. Rename Field
26. Rename Local Variable
27. Rename Method
28. Rename Method Parameters
29. Rename Package

30. Replace Constructor With Factory Method
31. Replace Inheritance with Delegation
32. Replace Method Code Duplicates
33. Replace Temp With Query
34. Use Interface Where Possible

SlickEdit 10.0 [35]

1. Convert Global To Static Field
2. Convert Local To Field
3. Convert Static Method To Instance Method
4. Create Standard Methods
5. Encapsulate Field
6. Extract Class
7. Extract Method
8. Extract Super Class
9. Modify Parameter List
10. Move Method
11. Move Static Field
12. Pull Up To Super Class
13. Push Down To Derived Class
14. Quick Rename
15. Rename Class
16. Rename Method
17. Rename Variable
18. Replace Literal With Constant

ReSharper 1.5 [36]

1. Convert Abstract Class To Interface
2. Convert Interface To Abstract Class
3. Convert Method To Property
4. Convert Property To Method
5. Copy Type
6. Encapsulate Field
7. Extract Interface
8. Extract Method
9. Extract Superclass
10. Extract Type To A New File
11. Change Method Signature
12. Inline Variable
13. Introduce Field
14. Introduce Parameter
15. Introduce Variable
16. Move Type
17. Rename Symbol