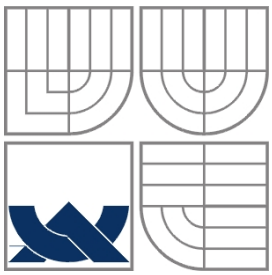
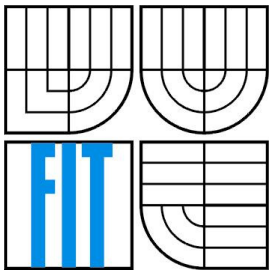


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REFAKTORING PŘI VÝVOJI SOFTWARE

REFACTORING IN SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Ján Dilík

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. Jitka Kreslíková, CSc.

BRNO 2008

Abstrakt

Táto práca sa zaoberá problematikou využitia refaktoringu, pri vývoji objektovo orientovaných aplikácií. Osvetľuje problematiku refaktoringu a jeho vlastnosti ovplyvňujúce vývoj softwarového projektu. Pred samotným refaktoringom je potrebné pokryť refaktorovaný systém testami. Samotný refaktoring je možné aplikovať v procese vývoja pri oprave chýb, kontrole kódu ale aj ako celkový refaktoring systému. V tejto práci sú uvedené dopady refaktorovania produktu OKbase firmy Oksystem s.r.o. a ich zhodnotenie vzhľadom k refaktorovaniu objektovo orientovaných aplikácií.

Kľúčové slová

refaktoring, testovanie, integračné testovanie, vývoj software, kontrola kódu, Java, OKbase, VVAF

Abstract

This paper deals with the subject of refactoring usage in the development of object-oriented applications. It outlines the refactoring theory and the properties that influence the software project development. Prior to refactorization, the system has to be covered with tests. The actual refactoring can be applied during the development process as bug fixing, code review or the refactoring of the whole system. Presented in this work are the impacts of refactorization of Oksystem's company OKbase product and the evaluation with respect to the refactoring concepts of object-oriented applications.

Keywords

refactoring, testing, integration testing, software development, code review, Java, OKbase, VVAF

Citácia

Ján Dilík: Refaktoring pri vývoji software, diplomová práca, Brno, FIT VUT v Brne, 2008

Refaktoring pri vývoji software

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením RNDr. Jitky Kreslíkovej, CSc. Ďalšie informácie mi poskytol Ing. Karel Miarka z firmy Oksystem s.r.o.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Ján Dilík
19.5.2008

Poďakovanie

Ďakujem vedúcej mojej diplomovej práce RNDr. Jitke Kreslíkovej, CSc., za vedenie a odbornú pomoc pri riešení problémov. Taktiež externému konzultantovi Ing. Karlovi Miarkovi, ktorý mi poskytol usmernenie hlavne v technických záležitostiach.

V neposlednom rade ďakujem mojím rodičom za podporu počas celého štúdia.

Obsah

1 Úvod

Vývoj software sa skladá z viacerých fáz životného cyklu softwarového projektu. Začína analýzou požiadaviek a od návrhu sa prechádza k jeho samotnej implementácii. Nasleduje testovanie software ako celku a jeho nasadenie u zákazníka. Týmto sa však vo väčšine prípadov softwarový projekt nekončí. Počas jeho života sa objavujú rôzne chyby, ktoré je potrebné opravovať. Mení sa jeho funkcionalita podľa aktuálnych požiadaviek klienta a dodáva nová. Týmito úpravami sa mení štruktúra pôvodného návrhu a projekt sa začína stávať neprehľadný, aj keď bol navrhnutý a implementovaný dobre. Týmto problémom sa zaoberá refaktoring, ktorého úlohou, je znova dostať takýto kód do čitateľnej podoby.

Refaktoring je cielený postup, pomocou ktorého sa softwarový produkt dostáva na vyššiu úroveň. Program, ktorého zdrojový kód je čitateľný, obsahuje menej chýb. Lepšie sa s ním pracuje, opravujú chyby a pridáva nová funkcionalita. Prehľadný a dobre štruktúrovaný kód sa lepšie optimalizuje a napomáha k vyššej produktivite programátorov. Účelom refaktoringu, vysvetlením základných princípov a popisom príznakov popisujúcich náznaky zlého kódu sa zaoberá druhá kapitola.

Úlohou refaktoringu je meniť iba jeho vnútornú štruktúru, t. j. zdrojový kód. Nesmie ovplyvniť funkčnosť systému navonok. Preto je pred samotným refaktoringom potrebné vytvoriť sadu testov popisujúcich refaktorovaný kód. Tie majú za úlohu zaručiť, že proces refaktorizácie kódu neovplyvnil správanie sa aplikácie z pohľadu jej funkčnosti. Pre podporu refaktoringu sa dnes používajú testovacie nástroje, ktoré tento proces uľahčujú. Týmito technológiami, testovacími návykmi ale aj technológiami podporujúcich hľadanie nečistôt sa zaoberá tretia kapitola.

Pred samotným vykonávaním refaktoringu je potrebné pochopiť refaktorovaný systém z hľadiska používania užívateľmi, ale aj z hľadiska vnútornej štruktúry kódu a použitých nástrojov. Refaktoring som aplikoval na systém OKbase firmy Oksystem s.r.o., ktorého funkčnosť a použité technológie sú popísané v štvrtej kapitole. Analýza problematiky sa zaoberá aj určením možných refaktorovaní a očakávaných problémov.

Piata kapitola sa zaoberá popisom tvorby testov pre zachytenie funkčnosti systému pred refaktoringom. Nasledujú špecifické refaktorovania pre systém OKbase pri oprave chyby, kontrole kódu alebo postup refaktorovania zle štruktúrovanej časti kódu pri celkovom refaktoringu systému.

Výsledkami aplikácie refaktoringu sa zaoberá šiesta kapitola, na ktorú nadväzuje zhodnotenie týchto výsledkov z hľadiska refaktorovania objektovo orientovaných aplikácií v siedmej kapitole.

Táto diplomová práca vychádza so semestrálneho projektu, v ktorom bola rozobraná problematika refaktoringu a technológií pre testovanie Java aplikácií. Získané poznatky sú v tejto práci rozšírené, aplikované a vyhodnotené na systéme OKbase.

2 Refaktoring a refaktorovanie

Prvé zmienky o refaktorovaní sa spájajú s programovacím jazykom Smalltalk, ktorý sa dá považovať za prvý OO jazyk. Najprv sa využíval pri vývoji programových knižníc, ale vlastnosti, ktoré tento spôsob programovania priniesol sa rozšírili aj do komerčnej sféry. Je to ustálený postup pri vývoji software, ktorý má zabezpečiť lepšiu čitateľnosť kódu programu.
Definícia refaktoringu podľa [1, str. 15]:

> *Refaktorovanie je proces vykonávania zmien v softwarovom systéme takým spôsobom, že nemajú vplyv na vonkajšie chovanie kódu, ale vylepšujú jeho vnútornú štruktúru. Je to disciplinovaný spôsob prečisťovania kódu s minimálnym rizikom vnášania chýb.*

V skratke: „Pri refaktorovaní zlepšujete návrh kódu po tom, čo bol napísaný.“ (viď [1, str. 15]).

2.1 Účel refaktoringu

Podľa definície vieme, že pri refaktorovaní ide o skvalitnenie kódu softwarového produktu. Ale aký to má význam pri vývoji software, keď sa nerozvíja žiadna nová funkcionalita? Programátori sú v dnešnej dobe platení za to, čo vyprodukujú. Akú novú funkcionalitu pridajú do vyvíjaného produktu. Takže prečo by sa malo refaktorovať, keď to pre zákazníkov neprinesie nič nové? Na túto otázku nie je ťažké odpovedať. Ťažšie je presvedčiť vedúceho projektu, poprípade ten najvyšší článok vývoja, ktorý riadi celý vývoj a je zaň zodpovedný.
Ak je kód kvalitný, lepšie sa s ním pracuje. Lepšie sa pridáva nová funkcionalita a opravujú chyby, ktoré sa počas používania zistia. Čo je najdôležitejšie, ak je kód kvalitný vyskytuje sa v ňom menej chýb. A to je hlavný dôvod prečo refaktorovať.

Za kvalitným programom stojí kvalitný návrh. Ale je niekoľko dôvodov, prečo sa z dobre navrhnutého produktu stáva „nečistý“ produkt. Spomeniem hlavné dva dôvody:

1. zmena pôvodného návrhu,
2. kvalita programátorov.

Kvalita programátorov

Cena programátora rastie s jeho vedomosťami, skúsenosťami, praxou a na základe iných vlastností. Kvalita programátora sa hodnotí z rôznych hľadísk a nie je možné hodnotiť dvoch programátorov jednou mierou. Hlavné je, že kvalita softwarového produktu závisí na kvalite a zložení tímu

programátorov, ktorí ho vyvíjajú. Pri dnešnej požiadavke trhu sa k vývoju software dostávajú aj menej skúsení, alebo tiež menej kvalitní programátori. To má za následok vnášanie nečistôt do kódu.

Zmena pôvodného návrhu

Po určitej dobe sa do systému pridávajú nové funkcionality, opravujú chyby. To má za následok zmenu pôvodného návrhu. Je veľká pravdepodobnosť, že sa takýmto spôsobom do kódu zanesú nečistoty, a to aj keď je vývojársky tím zložený z kvalitných programátorov.

Refaktorovanie je teda úprava štruktúry kódu bez toho, aby ovplyvnili funkcionality celého systému. To je možné vykonávať v určitých cykloch vývoja, alebo aj v samostatných cykloch programátora pri jeho práci – vývoji nových funkcionalít a pod.

2.2 Princípy refaktorovania

Refaktorovanie nie je len obyčajné čistenie kódu, ale poskytuje postupy ako čistiť kód efektívnejšie a cielene. Udáva presný postup, ako pri refaktorovaní postupovať na základe určených a overených pravidiel.

Dôležité je tiež uvedomiť si, že zmyslom refaktorovania je zlepšiť zrozumiteľnosť a možnosť ďalších zmien softwarového projektu. Ten je možné upraviť rôznymi spôsobmi, tak aby nemenili jeho chovanie. Jedným z nich je optimalizácia výkonu, ktorá tiež nemení funkčnosť programu, ale iba jeho vnútornú štruktúru. Keďže nehľadí na čitateľnosť kódu ale na rýchlosť programu, väčšinou sa kód stane menej čitateľný.

2.2.1 Prečo by sa malo refaktorovať

Keďže pri refaktorovaní len čistíme a sprehľadňujeme kód, prečo by sme to mali robiť? Odpoveď na túto otázku by sa dala zhrnúť do štyroch bodov, z ktorých prvé tri som už naznačil v prvej časti.

Čo prináša refaktorovanie:

1. zlepšuje návrh software,
2. vedie k lepšej zrozumiteľnosti software,
3. pomáha hľadať chyby,
4. pomáha k zrýchleniu práce programátora.

Zlepšuje návrh software

Keď softwarový projekt beží už určitú dobu, jeho návrh začína naberať nedostatky, ktoré sťažujú prácu vývojárom. Je zložité tento kód čítať, pretože v ňom boli vykonané zmeny bez jeho

pochopenia, alebo tieto zmeny slúžili len ku krátkodobým cieľom. Takýto návrh potom stráca svoju štruktúru a je zložitejšie sa v ňom orientovať, pochopiť ho a zachovať.

Zle navrhnutý kód obsahuje duplicity, ktoré sú častou príčinou chýb. Nachádzajú sa v zbytočne dlhých kódoch a čím zložitejší je kód, tým zložitejšie sa upravuje, vylepšuje a opravujú sa v ňom chyby. Taktiež je zložitejší na pochopenie programátorom. Keď sa zmení jedno miesto v kóde, program nevykonáva to čo by mal, pretože sa nezmenil na druhom duplicitnom mieste. Preto sa pri refaktorovaní zameriava na dlhý kód (napr. dlhé metódy a triedy) a odstraňujú sa duplicity, ktoré obsahuje.

Vedie k lepšej zrozumiteľnosti software

Časy, keď jediným programovacím jazykom bol strojový jazyk alebo po čase jazyky jemu blízke sú dávno preč. V dnešnej dobe OO jazykov využívame objekty a pomenovávame ich podľa toho, čo v reálnom svete vyjadrujú. Môžeme písať názvy metód podľa toho čo vykonávajú. Proste môžeme písať kód, ktorý bude čitateľný pre počítač ale aj pre programátorov.

Už som spomínal kvalitu programátora. Tá spočíva aj v tom, aký kód po ňom zostane. V dnešnej dobe je veľmi pravdepodobné, že vývojár na danom projekte nebude pracovať vždy. Buď si nájde lepšie miesto, alebo postúpi v kariérnom raste na inú pozíciu alebo projekt. Preto je dôležité aby po ňom zostal kód, ktorý je dobre čitateľný a vyjadruje presne to čo by mal. Vývojár musí myslieť aj do budúcnosti a písať kód tak, aby bol zrozumiteľný aj pre tých, ktorí prídu po ňom. Potom pri príchode nových členov do tímu bude ich začlenenie a pochopenie kódu jednoduchšie a rýchlejšie, ako keby sa mali brodiť nezrozumiteľným kódom.

Preto napr. po ukončení vývoja novej funkcie je potrebné, aby po sebe programátor refaktoroval kód ktorý vytvoril. Jednoducho povedané, spravil ho čitateľnejším aj pre ostatných programátorov, ale aj pre seba s odstupom času.

Pomáha hľadať chyby

Na to, aby bola opravená chyba v kóde, je potrebné jeho presné pochopenie. Ak je kód napísaný kvalitne a čitateľne, je to jednoduchšie. Ale aby si bol programátor istý, nič mu nebráni napísať si test a refaktorovať daný kód. Možno nájde cestu ako ho lepšie vyjadriť a tým pochopí činnosť kódu dôkladnejšie. Oprava chyby je už potom väčšinou len maličkosť.

Hlavné je pochopiť rozdiel medzi refaktorovaním pred opravou a samotným odstránením chyby. Ak pri refaktorovaní odstránime chybu, nejedná sa o refaktorovanie. To je dôležité si uvedomiť. Refaktoruje sa chybný kód a po refaktorovaní musí obsahovať tú istú chybu, pretože pri refaktoringu sa nemení funkčnosť programu. Alebo najprv sa odstráni chyba a až potom sa vykoná dôkladnejšia refaktorizácia kódu (napr. odstránenie duplicity).

Pomáha k zrýchleniu práce programátora

Dobrý návrh je základom kvalitného vývoja. Refaktorovanie pomáha vyvíjať software rýchlejšie, pretože zamedzí zhoršeniu architektúry systému. Implementácia nových funkcií a oprava chýb je jednoduchšia.

2.2.2 Kedy je potrebné refaktorovať

Nedá sa jednoducho povedať, kedy by sa malo refaktorovať. Refaktoruje sa po malých krokoch. Keď programátor vidí kód, ktorý by bolo lepšie trochu upraviť, tak ho upraví. V ojedinelých prípadoch, keď sa prechádza k novému projektu a je vidieť, že je z väčšej časti „špinavý“, pristúpi sa k celkovému refaktoringu.

Pravidlo „Do tretice“

Pravidlo prevzaté z [1, str. 74]:

Keď niečo robíš prvýkrát, proste to spravíš. Keď niečo podobné robíš druhýkrát, premýšľaš kvôli duplicite, ale nakoniec to aj tak spravíš. Keby si mal niečo podobné robiť tretíkrát, budeš refaktorovať.

Refaktorovanie pred pridaním funkcionality

Sú dva typy refaktorovania kódu pred pridaním novej funkcionality. Prvá je, že pri refaktoringu programátor lepšie pochopí kód, do ktorého danú funkčnosť zasadzuje. Druhou je možnosť jednoduchšej implementácie novej funkčnosti. Je možné, že súčasný návrh nie je dostačujúci, alebo sa jeho refaktorovaním dosiahne jednoduchšia a kvalitnejšia implementácia tejto funkcionality, s ktorou sa možno z počiatku nepočítalo a zákazník s ňou prišiel dodatočne.

Refaktorovanie pri oprave chyby

Častokrát je ohlásená chyba znamením k refaktorovaniu, pretože kód nie je tak zrozumiteľný, aby bola chyba jednoducho odstránená. Pred odstránením je potrebné lepšie pochopenie kódu. Pomocou refaktorovania sa programátor s kódom lepšie zoznámi a uľahčí mu opravu.

Refaktorovanie pri revízii kódu

Revízia kódu je myšlienka pochádzajúca z XP programovania (extrémne programovanie – *Extreme Programming*). Jedná sa o revíziu napísaného kódu skúsenejším programátorom alebo tímovým kolegom. To čo napíše jeden programátor, nemusí byť zrozumiteľné pre ostatných členov tímu. Preto sa po určitých (nepravidelných) cykloch kód reviduje a dostáva do stavu čitateľného pre celý tím. Revízor navrhuje svoje pripomienky a po dohode s autorom sa implementujú zmeny v kóde. Po takejto revízii je kód čitateľnejší a prehľadnejší. Niekedy sa pri samotnej revízii nájdu aj chyby.

2.2.3 Problémy spojené s refaktorizáciou

Jedná sa o problémy, ktoré pri refaktoringu môžu nastať. Je dôležité aby si ich vývojár, ktorý refaktoring aplikuje uvedomil a vedel o nich. Zvlášť človek, ktorý s refaktoringom ešte len začína. Medzi základné problémy patrí:

1. databáza,
2. zmeny rozhrania,
3. rýchlosť programu,
4. kedy sa refaktorovaniu vyhnúť.

Problémy s databázou

Jedná sa o problém tesného spojenia programu so štruktúrou databázy, nad ktorou je program postavený. Pri veľkých dátach je zložitá migrácia dát z jedného návrhu do druhého. Preto je zložité začleniť zmeny vzniknuté refaktorovaním dôležitej časti návrhu, ktorá je s databázou tesne spojená. Dnešné aplikácie sú vyvíjané pomocou oddeľujúcich vrstiev, ktorých úlohou je oddeliť tesnú náväznosť medzi databázou a štruktúrou programu. Jedná sa o vrstvu oddeľujúcu databázový a objektový model, ktorý zaručuje ich nezávislosť a pri zmene jedného sa nemusí meniť druhý.

Zmeny rozhrania (*interface*)

V jednoduchosti môžeme povedať, že rozhranie slúži na zapúzdrenie operácií, ktoré môžu využívať iné objekty. Ak sa refaktorujú objekty, využívajúce rozhranie, nie je tu žiaden problém. Problém nastáva, ak sa mení rozhranie, pretože na ňom sú závislé okolité objekty. Uvažujme so zmenou názvu metódy. Ak je vidieť všetky jej volania, ktoré dané rozhranie využívajú nie je problém. Ten nastáva v prípade, že rozhranie je použité v kóde, ktorý nemôžeme nájsť a zmeniť. Martin Fowler takéto rozhranie nazval publikované rozhranie (*published interface*) (viď [1, str. 81]). V jednoduchosti je možné ako príklad uviesť rôzne knižnice alebo API, ktoré programátor využíva. Tím vyvíjajúci tieto knižnice nemôže meniť volania a rozhrania ako si zmyslí. Verzie využívajúce programátormi (na celom svete) by potom neboli kompatibilné a nikto by ich nepoužíval. Bližší popis a rozdiel medzi verejným a publikovaným rozhraním je popísaný v [2].

Jedným z riešení je nepublikovať rozhranie v rámci vývojového tímu. K tomu je potrebné nastaviť vlastnícke práva tak, aby kód mohli upravovať aj iní programátori a nie len jeho autor. Ďalším riešením je nahradiť funkciu starého rozhrania novým s tým, že staré rozhranie je potrebné zachovať a bude volať nové rozhranie. Veľkou chybou je, keď sa nové rozhranie vytvorí skopírovaním starého. Takto vznikajú duplicity kódu. Staré rozhranie (napr. názov metódy) využívajúce nové, je dobré označiť anotáciou `deprecated`, ktorý hovorí o zavrhnutí tohto riešenia a existujúcom novom riešení.

Rýchlosť programu

Keďže pomocou refaktoringu dostávame kód programu do čitateľnej podoby, často sa kvôli tomu spravia zmeny, ktoré danú aplikáciu spomalia. Nestáva sa to vždy, ale takéto prípady existujú. Pre zákazníka je rýchlosť aplikácie dôležitá, pretože veľmi pomalú aplikáciu nemusí prijať. Aj keď refaktorovanie určitým spôsobom rýchlosť programu spomalí, na druhú stranu dáva oveľa lepšiu možnosť pre ladenie výkonu. Je samozrejmé, že sa jednoduchšie ladí dobre čitateľný a štruktúrovaný kód ako nečitateľný.

Ak sa nejedná o aplikácie pracujúce v reálnom čase, kde je rýchlosť najdôležitejšia, najprv by sa optimalizácia pri programovaní nemala brať v úvahu, pretože sa väčšinou optimalizuje časť kódu, ktorá to vôbec nepotrebuje (podľa [1, str. 86] sa jedná až o 90% optimalizovaného kódu). Mal by sa písať kód dobre čitateľný a optimalizovať až pomocou ladiacich nástrojov (profilerov) na miestach, ktoré to naozaj potrebujú. Martin Fowler v [1] spomína, že refaktorovanie mu pomáha písať rýchly software, pretože počas refaktorovania je pomalší, ale výsledný kód je možné oveľa rýchlejšie optimalizovať. Konečný výsledok je tak omnoho rýchlejší.

Kedy sa refaktorovaniu vyhnúť

Sú prípady kedy sa refaktorovanie nevypláca. Ak máme refaktorovať kód, ktorý obsahuje veľa chýb, nie je to najlepšia cesta. Signálom k refaktoringu je nefunkčný kód, ale ak sa pri vytváraní testov nájde oveľa viac nefunkčných celkov a chýb ako sa predpokladalo, mal by sa kód proste prepísať. Ako bolo spomenuté, refaktorovaním sa nemení funkčnosť systému, ale len jeho vnútorná štruktúra, ktorá sa dáva do čitateľnejšej podoby.

Druhým prípadom kedy sa refaktorovaniu vyhnúť, je podľa [1, str. 83] blízkosť termínu odovzdania projektu. Zvýšená produktivita, ktorú by refaktorovanie prinieslo, by sa dostavila až po termíne odovzdania, teda neskoro. Pretože význam refaktoringu nie je krátkodobý, ale svoje opodstatnenie má v dlhšom horizonte života softwarového projektu.

2.2.4 Problematické časti kódu

Pachy v kóde popisujú najčastejšie situácie, ktoré ukazujú na náznaky nečistého kódu. Jedná sa o príklady nečistôt, ktoré dávajú znamenie, že túto časť kódu je potrebné refaktorovať. Táto kapitola popisuje len základný princíp jednotlivých pachov. Bližší popis je uvedený v [1, Kapitola 3].

Duplicitný kód

Jedná sa o najvýraznejšiu nečistotu. Duplicity v kóde sú častým predpokladom pre vznik chyby. Ak sa na viacerých miestach nájde rovnaký blok kódu, je potrebné ho zjednotiť.

Dlhá metóda

OO jazyky sa vyznačujú implementáciou krátkych metód v objektoch. Metódy by mali byť čo najkratšie aby vyjadrovali len jednu úlohu. Tým je kód prehľadnejší, lepšie štruktúrovaný a v kóde sa jednoduchšie implementujú jeho zmeny.

To či je metóda dlhá, niekedy nie je jednoduché určiť. Jednou možnosťou ako už bolo spomenuté je zistenie, že metóda robí viacero logicky oddeliteľných úsekov. Druhou sú komentáre, ktorými sa vysvetľujú jednotlivé kroky, pričom jednoduchšie by bolo extrahovať komentovaný celok ako samostatnú metódu. Ďalšími znakmi môžu byť podmienky a cykly.

Veľká trieda

Veľká trieda je podobný prípad ako predchádzajúci pach dlhá metóda. Aj triedy s dlhým kódom, zvyknú obsahovať duplicity. To, že sa trieda stará aj o tie úlohy, ktoré by nemala spracovávať, naznačuje aj veľké množstvo inštančných premenných.

Pre zjednodušenie tried a refaktorovanie do jednoduchšej čitateľnej podoby sa používajú refaktorovania ako extrahovanie triedy, podtriedy rozhrania a iné.

Dlhý zoznam parametrov

Metódy v OO jazykoch by nemali mať veľký počet vstupných parametrov. Pri procedurálnom programovaní bolo potrebné do každého podprogramu predávať všetky potrebné parametre, aby sa vyhlo používaniu globálnych premenných. Pri objektoch je jednoduchšie metóde predať objekt, kde si tieto informácie môže nájsť. Taktiež je veľa vecí, ktoré metóda potrebuje pre svoju činnosť v triede, v ktorej sa nachádza.

Dlhý zoznam parametrov môže tiež zbytočne odlákať pozornosť programátora inou cestou. Preto je potrebné udržovať počet parametrov v rozumnom počte.

Protichodné zmeny a rozptýlené úpravy

Dobre štruktúrovaný software je taký, ktorého zmena kódu na jednom mieste neovplyvní chovanie systému na inom mieste. Alebo pri zmene chovania systému, musíme vykonať zmeny na viacerých miestach kódu. Ich úlohou je vytvoriť takú štruktúru programu, aby existovala jednoznačná existencia medzi prevádzanými zmenami a triedami.

O protichodných zmenách hovoríme, ak je bežné meniť jednu triedu odlišnými spôsobmi na základe rôznych dôvodov. Ak sa napr. pri zmene DB menia stále tie isté metódy v jednej triede.

Rozptýlené úpravy predstavujú obrátené protichodné zmeny. Nastávajú, ak je pri jednej zmene potrebné spraviť viacero drobných úprav vo viacerých triedach. Tieto miesta sa obvykle ťažko hľadajú. Takéto zmeny je potrebné umiestniť do jednej triedy.

Chýbajúce schopnosti

Pach chýbajúce schopnosti je možné odhaliť tak, že objekt pre svoju činnosť neustále pristupuje k metódam iných objektov. Myslí sa tu nadmerný prístup, napríklad pri výpočte určitej hodnoty, pre tento objekt dôležitej. Ak je zjavné, že by tieto operácie mali patriť do tohto objektu, je potrebné dané metódy presunúť.

Dátové zhluky

Ak sa na rôznych miestach programu nachádzajú časté zhluky dátových položiek, jedná sa o dátové zhluky. Sú to napríklad polia v triedach alebo parametre do metód. Takéto dáta väčšinou po rozdelení nedávajú zmysel. Z takýchto skupinových dát je dobré vytvoriť samostatný objekt, a poprípade ak sa takto vytvorenej triede zistia chýbajúce schopnosti doplniť ich.

Primitívna obsesia

Tento názov vyjadruje posadnutosť používania primitívnych dátových typov v OO jazykoch. Programátor sa musí vedieť rozhodnúť, kedy použiť jednoduchý dátový typ alebo objekt.

Príkazy switch

Príkazy `switch` často spôsobujú duplicitu kódu. Väčšinou sa na rôznych miestach kódu nachádzajú rovnaké delenia kódu pomocou `switch` príkazu. Keď potrebujeme dodať novú vetvu, je potrebné tieto výskyty vyhľadať a doplniť to do každého výskytu. Preto je lepšie využiť polymorfizmus, ktorý nám poskytuje objektovo orientovaný jazyk (Java).

Paralelná hierarchia dedičnosti

Problém nastáva, keď pri každom vytvorení podtriedy je potrebné vytvoriť tiež podtriedu z inej triedy. Často je tento pach identifikovateľný z toho, že predpony tried z prvej a druhej sú rovnaké.

Lenivá trieda

Trieda ktorá stratila význam pri refaktorovaní a nenesie už význam musí byť odstránená. Taktiež to môže byť trieda, ktorá bola navrhnutá pre ďalšie rozšírenie, ale to už nebolo implementované. Je to každá trieda, ktorá nerobí toľko aby mohla byť považovaná za samostatnú. Metódy ktoré ešte obsahuje sa presunú do iných tried.

Špekulatívna obecnosť

Špekulatívna obecnosť je, keď sa v kóde často vytvárajú zložité postupy, ktoré robia kód zbytočne zložitým s tým, že v budúcnosti sa to môže niekedy hodiť. Takýto kód je neprehľadný a programátor, ktorý sa k nemu dostane musí zbytočne pátrať, prečo je to tak a to ho odtrhuje od skutočnej podstaty

kódu. Patria medzi ne aj abstraktné triedy, ktoré nemajú využitie a sú zbytočné alebo nevyužité inštančné premenné v triedach.

Dočasná položka

Stáva sa, že v niektorých triedach sa nachádzajú inštančné premenné, ktoré nenadobúdajú svoju hodnotu za každých okolností. Sú to dočasné položky, ktoré slúžia vo väčšine prípadov pri výpočte zložitých algoritmov. Tie sú veľmi mätúce a je dobré zmeniť ich použitie.

Zreťazené správy

Keď jeden objekt získava hodnotu cez iné objekty často pomocou niekoľkých prístupových `get` operácií za sebou, jedná sa o zreťazené získavanie správ. Každá menšia zmena sa potom objektu, ktorý získava správy takouto cestou dotkne a je potrebné to upravovať.

Prostredník

V OO programovaní sa často používa zapuzdrenie a prostredník, ktorý poskytuje operácie objektu iným objektom. Je dôležité aby sa tento prístup využíval opatrne. Ak objekt väčšinu svojich operácií realizuje volaním metód druhého objektu, je to potrebné delegáta odstrániť.

Nevhodná dôvernosť

Objekty by na sebe nemali byť závislé a ak používajú nejaké spoločné operácie, tie je potrebné oddeliť na osobitné miesto. Jedným z prípadov je, ak v dedičnosti vie potomok o predkovi viac ako by mal.

Alternatívne triedy s rôznymi rozhraniami

Názvy metód, ktoré sa nachádzajú v rôznych triedach, vykonávajú tú istú funkciu ale ich názvy sa odlišujú je potrebné zjednotiť.

Neúplná knižničná trieda

Tvorba knižníc nie je jednoduchá a častokrát sa jej kvalitný návrh črtá až pri konci jej vývoja. Je jednoduché používať cudzie knižnice, ale ťažšie je vytvoriť si vlastné. Pri vývoji je preto potrebné upraviť si nejaké časti cudzích knižníc, čo je zložité.

Dátová trieda

Sú to triedy, ktoré obsahujú len položky a ich prístupové metódy. Využívajú sa ako úložištia dát. Ak takéto triedy obsahujú verejné inštančné premenné, je potrebné ich zapuzdriť a k ním vytvoriť/zakázať prístupové metódy. Malo by sa zistiť ich využitie v systéme a ak existujú operácie, ktoré sa na nich často vykonávajú a súvisia s nimi, bolo by dobré ich presunúť do tejto triedy.

Odmietnuté dedičstvo

Keď podtriedy nevyužívajú väčšinu metód zdedených z rodičovskej triedy, jedná sa o odmietnuté dedičstvo. Tento prípad nastáva pri zlom návrhu hierarchie tried a je potrebné ho reorganizovať. Ak je to zložité, nie je to až taký problém ako keď sa neimplementujú metódy z rozhrania.

Komentáre

Komentáre často upozorňujú na zle čitateľný kód. Tie by sa mali používať len v prípade doplnenia informácií pri zložitom algoritme, alebo vysvetlenie a poznámky autora. Nemali by dokumentovať kód. Na to je dobré správne pomenovanie metód a ich dobrá štruktúra.

3 Technológie pre podporu refaktoringu

Keďže proces refaktoringu je náročný a zdĺhavý pri refaktorovaní sa musia používať rôzne pomocné nástroje. Refaktoring nesmie zmeniť funkcionalitu produktu, ktorý sa refaktoruje, a preto je potrebné zabezpečiť aby sa počas refaktorovania nemenila funkčnosť produktu. To je možné zabezpečiť testovacími nástrojmi.

Ďalšou súčasťou refaktorovania je využívanie nástrojov, ktoré refaktoring uľahčujú a zrýchlia jeho samotný priebeh. Zároveň sa dá spoľahnúť, že daný proces refaktorizácie prevedú bezpečne.

3.1 Overenie správnosti refaktoringu

To či sa pri refaktoringu kódu nezmenila funkcionalita, teda nevniesli chyby do vyvíjaného software nám zaručí vytvorenie testov pred samotným refaktoringom. Vždy pred začatím refaktorizácie kódu je potrebné kód určený k čisteniu, pokryť sadov testov, ktoré zachytia funkčnosť pred refaktoringom a overia jeho správnosť po ukončení.

Jedná sa o cielené pokrývanie kódu programu sadov testov. Výhodné je, ak boli testy vytvárané už v priebehu vývoja software, pretože písanie testov na už napísaný kód je zložité. Napísanie testu na existujúci kód v sebe zahŕňa viacero problémov. Jeho jedinou výhodou je, že na napísanie kvalitných testov je potrebné jeho úplné pochopenie, čo je pri refaktorovaní aj výhodou.

Testovaním sa zaoberá hlavne vývoj riadený testami (TDD – *Test-Driven Development*), ktorý popisuje metodiky tvorby software pomocou testov. Je to spôsob vývoja, pri ktorom sa ešte pred samotným písaním funkčného kódu píšu testy a až v rámci testov sa tvorí kód programu. Takto vytvorený software obsahuje menej chýb a je tiež jednoduchší. Znaky a tvorby testov je možné využiť aj pri tvorbe sady testov potrebných pri refaktoringu.

3.1.1 Tvorba sady testov

Tvorba testov pri refaktorovaní je veľmi dôležitá a nesmie sa podceňovať. Pri každom kroku si programátor uplatňujúci refaktoring musí byť istý, že každá jeho chyba bude zachytená. Ak sa na testy nekladie patričný dôraz, výsledkom refktoringu môže byť nedokonalosť pokračovať v refaktoringu, kvôli neistote vykonaných refaktorovaní.

Princíp tvorby testu je rovnaký pri použití každého testovacieho framework. V publikácii [3, str. 10] je popis tvorby testov použitý v TDD, ktorý je možné aplikovať aj na testy, vyvíjané pre zachytenie chýb vzniknutých refaktoringom.

Postup tvorby testu je zasadený do troch pravidiel:

1. červená,
2. zelená,
3. refaktorovanie.

Červená

Prvotnou úlohou tvorby testu je aspoň prázdny alebo nefunkčný test. Ten sa potom rozširuje, upravuje a dopĺňa o nové testy, pričom sa stále zoznamujeme s kódom určeným pre refaktorizáciu.

Zelená

Zelená je farba ukazujúca správny priebeh. To isté je aj pri písaní testov. V tomto štádiu sa spriechodňuje napísaný test. Dostáva sa z nefunkčného stavu do stavu funkčného.

Refaktorovanie

Refaktorovaním sa v tomto kroku myslí úprava napísaného testu do čitateľnej podoby, pretože pri prechode z červeného stavu do zeleného, sa nehľadí na čistotu kódu ale iba na spriechodnenie testu a pochopenie funkcionality kódu. Takto funkčný test sa potom upraví do prijateľnej formy.

3.2 Testovanie java aplikácií

Pre programovanie v jazyku java bolo vyvinutých viacero nástrojov na testovanie aplikácií. Zoznam väčšiny voľne šíriteľných testovacích nástrojov je uvedený v [4]. Každý sa venuje inej problematike a testuje aplikácie iným spôsobom.

Typy testovania Java aplikácií:

1. klasické „Unit“ testy,
2. integračné testy,
3. testy pomocou mocking objektov,
4. testy GUI.

3.2.1 Jednotkové (Unit) testy

Jednotkové alebo tiež klasické testy, slúžia pre programátorov na otestovanie jednotkovej funkčnosti v kóde, ale nie sú závislé na celom systéme. Tieto testy sa venujú jednej problematike. Zástupcom takýchto testov je najznámejší nástroj na testovanie java aplikácií JUnit (URL <http://junit.sourceforge.net>). Jednotkou sa myslí trieda, poskytujúca určitú funkciu, ktorú je potrebné otestovať. Testy sa potom vytvárajú na jednotlivé metódy danej triedy.

Vytvorí sa testovacia trieda, dedená od hlavnej triedy `TestCase`, v ktorej sa implementujú metódy vykonávajúce testovanie `pomocou assert` metód. V každej testovacej triede je možné

implementovať metódy, ktoré sa vykonajú pred samotným spustením testovacej metódy a po jej ukončení. Tie zaručujú nezávislý priebeh spúšťania testovacích metód.

3.2.2 Integračné testy

Písanie integračných testov je zložitejšie ako písanie testov jednotkových. Sú to testy, ktoré testujú súčinnosť viacerých objektov (komponent). Ich využite je hlavne pri testovaní Java EE aplikácií, kde je súčinnosť viacerých objektov nevyhnutná. Pomocou týchto testov by sa mal software testovať ako celok (viď [5]). Priebeh testu by sa mal približovať reálnemu behu aplikácie.

Do tejto triedy ja možné zaradiť nástroj DbUnit (URL <http://dbunit.sourceforge.net>), pomocou ktorého je možné vytvárať databázové testy. To nám umožní lepšie simulovať reálne prostredie vykonávania programu.

Nástroj DbUnit je rozšírením JUnit, pre podporu testovania DB aplikácií. Vytvorenie testu je podobné ako pri vytvorení klasického testu. Obsahuje metódy, ktoré sa volajú pred spustením a po ukončení každej testovacej metódy. Rozdiel je v tom, že pri testovaní je umožnené vytvoriť si vlastnú databázu, alebo použiť rovnakú DB, s tým že dáta sa načítajú z XML súboru (je možné použiť aj iné zdroje). Každá operácia vykonaná nad touto databázou na ňu nemá vplyv, pretože po každej testovacej metóde sú tieto zmeny zahodené (je vykonaný tzv. rollback).

3.2.3 Testovanie pomocou mock objektov

Túto techniku testovania je možné zaradiť do stredu medzi klasické jednotkové testy a integračné. Jedná sa o klasické testovanie, pričom ak je k vykonaniu tohto testu potrebné využitie súčinnosti viacerých objektov alebo aj databázy, je možné operácie daného objektu nahradiť pripravenými výsledkami, ktoré potrebujeme. Používa sa hlavne tam, kde je náročné zostavenie prostredia objektov, ktoré sú potrebné pre otestovanie daného objektu. Neoveruje sa tu súčinnosť týchto objektov, ale len samotný testovaný objekt, za využitia zástupných tzv. mocking objektov.

Najznámejším nástrojom pre testovanie pomocou zástupných „mock“ objektov je EasyMock (URL <http://www.easymock.org>). Vytváranie testovacích tried je obdobné ako v pri vytváraní klasických testov, pričom pre testovanie aplikačnej logiky je v testovacích metódach možné používať zástupné objekty. Vytvorí sa objekt a miesto pôvodnej implementácie sa mu nastaví testovacia náhrada, ktorá zaisťuje, že pri zavolaní vráti nami požadované dáta.

3.2.4 Testy GUI

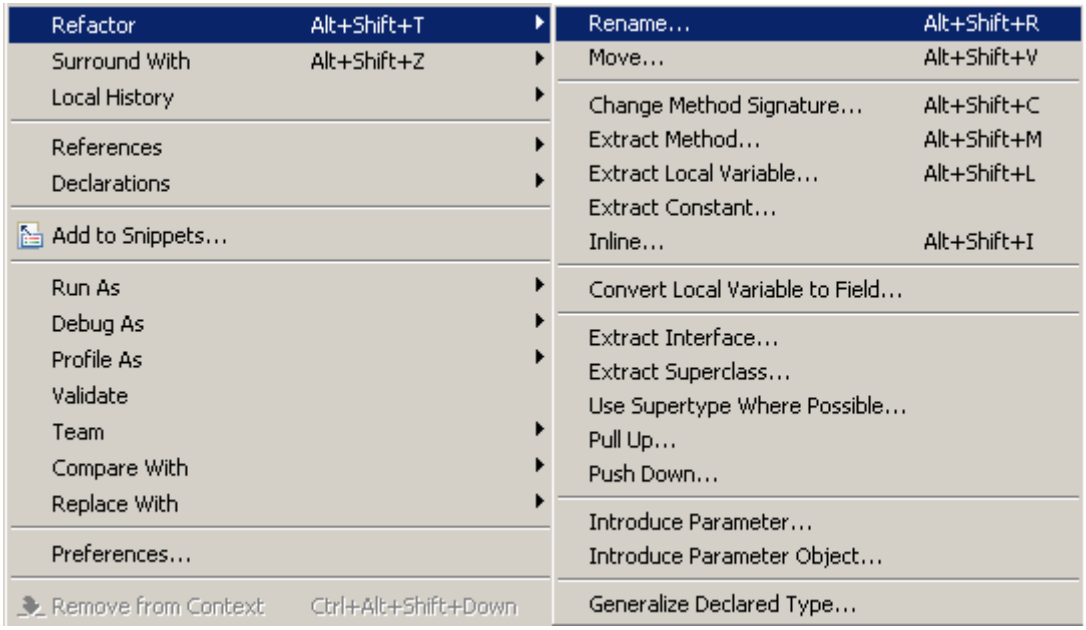
Testovanie grafického užívateľského rozhrania patrí tiež medzi testovacie metodiky. Zameriam sa hlavne na testovanie webového rozhrania. Pri intranetových aplikáciách s prístupom cez webový prehliadač je potrebné, aby nástroje na testovanie neboli závislé na webovom prehliadači. Testuje sa

štruktúra HTML dokumentov a obsah, ktorý obsahujú. Tieto testy musia byť plne automatizované, aby nahradili namáhavú ručnú prácu, ktorá je zdĺhavá.

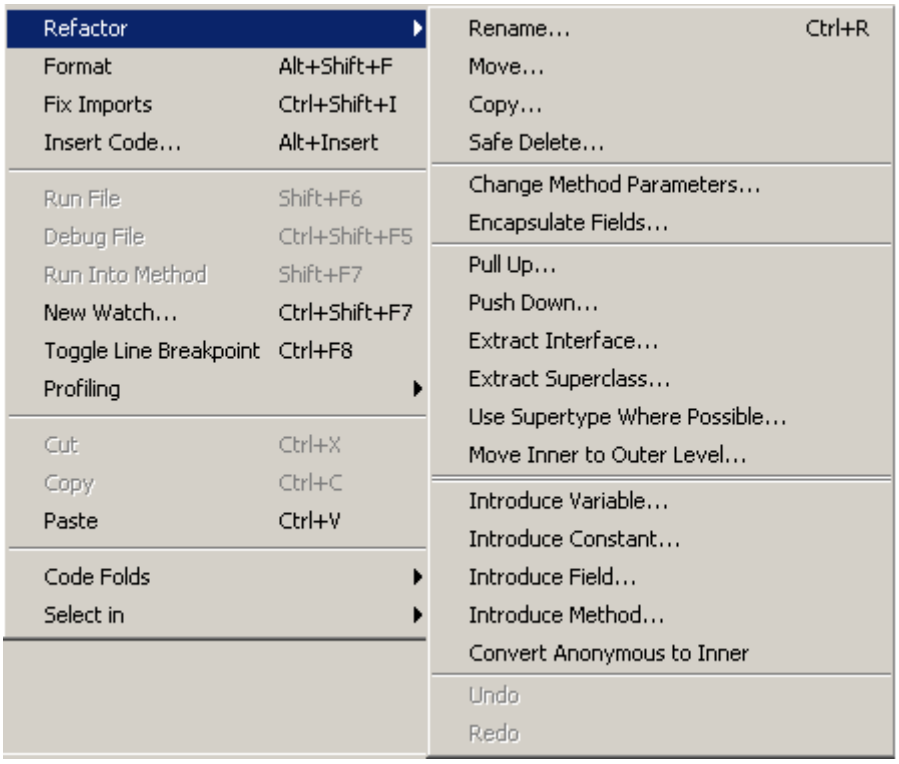
Nástroj Selenium [6] poskytuje možnosti pre testovanie webových aplikácií. Spúšťajú sa priamo vo webovom prehliadači a vykonávajú operácie ako praví užívatelia. Pomocou tohto nástroja je možné otestovať funkcionalitu systému, ale aj jeho správnu funkčnosť pod viacerými webovými prehliadačmi. Podporované platformy a prehliadače sú uvedené v [6]. Funkčnosť testov je realizovaná skriptami napísanými v jazyku JavaScript. Pomocou Selenium IDE, je možné tieto skripty raz nahrať z používania systému užívateľom, a potom automatizovane spúšťať.

3.3 Zjednodušenie refaktorovania

Proces refaktorizácie kódu v dnešnej dobe uľahčujú integrované vývojové prostredia (IDE). Tie poskytujú zaručené a bezpečné prevedenie jednoduchších typov refaktorizácie, napr. premenovanie metódy či premennej, typy extrahovania a presunu položiek a iné. Na Obr.3-1 a Obr.3-2 sú zobrazené rolovacie ponuky obsahujúce rôzne typy refaktorovania kódu. Tieto rozšírenia zjednodušili proces refktorizácie kódu vo veľkej miere najme pri jeho úprave počas pridávania novej funkcionality, alebo oprave chýb, teda počas vývoja softwarového produktu.



Obr. 3-1: Možnosti refaktorizácie v IDE Eclipse.



Obr. 3-2: Možnosti refaktorizácie v IDE NetBeans.

Pri jednoduchších typoch refaktorovania ako napríklad premenovanie, extrakcia, presun položky alebo metódy sa skráti čas vykonania. Pred samotným refaktorovaním nie je potrebná tvorba testov. IDE vyhľadá všetky výskyty a nahradí ich novými. Pred prepísaním ešte ponúkne zoznam nájdených výskytov spolu s tým, ako bude daná časť kódu vyzerať po refaktoringu.

3.4 Ďalšie možnosti

Hľadanie nečistôt v kóde môže byť náročný proces. Častou otázkou je kde začať hľadať, alebo vôbec ako začať. Pre získanie obrazu o stave kódu nám môžu pomôcť reporty, generované zo zdrojových kódov podľa nadefinovaných pravidiel.

3.4.1 PMD reporty

Skratka PMD vlastne neznamená žiadny význam. Autori uvádzajú, že názov tohto nástroja vznikol tým spôsobom, že sa im zapáčil sled týchto troch po sebe idúcich písmen a poskytujú možné vysvetlivky ako napr. „*Prety Much Done*“ - veľmi pekne spravené, alebo „*Project Mess Detector*“ – detektor neporiadku (pachov) v projekte (viď [7]).

Jedná sa o open source projekt, ktorý vyhľadáva možné problémy v zdrojovom kóde napísanom v programovacom jazyku Java. Možné chyby hľadá na základe syntaktickej analýzy kódu a vyhodnocuje ich podľa nastavených pravidiel.

Vyhľadávané problémy:

- možné chyby – napr. prázdne bloky `try`/`catch`/`finally`/`switch`,
- mŕtvy kód – tzv. „*dead code*“, nepoužívané premenné, parametry alebo celé metódy. V kóde môže nájsť časť kódu, ktorá sa nikdy nevykoná,
- nedokonalý kód – (neoptimálny kód), napr. zbytočné použitie `String`/`StringBuffer`,
- prehnane komplikovaný kód - zbytočne komplikovaný kód, ktorý je možné riešiť elegantnejšie (priame spojenie s pachmi). Napr. zbytočné podmienky `if` pre cykly, ktoré môžu byť implementované pomocou príkazu `while`,
- duplikovaný kód – (priame spojenie s pachmi v kóde), natvrdo kopírovaný a na iné miesto vkladaný kód spôsobuje chyby.

Reporty je možné vytvárať samostatne, alebo pomocou IDE. Podporuje integráciu do množstva vývojárskych Java prostredí ako napr. Eclipse, NetBeans, IntelliJ IDEA, JDeveloper a iné.

3.4.2 Reporty programátorských konvencií (*Checkstyle reports*)

Jedná sa o nástroj pre kontrolu štylistiky zdrojových kódov napísaných v jazyku Java. Hlavnou úlohou je pomôcť programátorom dodržiavať sa určitým štandardom pri písaní kódu v Jave. Kontrola sa robí automaticky a nezaťažuje tým programátora. Je to ideálny nástroj pre projekt, ktorý chce dodržiavať určité štandardy.

Tento nástroj je možné použiť aj pri hľadaní problémových časti kódu pred a počas refaktoringu. Je pravdepodobné, že v blokoch kódu, ktorý nedodržiava určité štandardy, sa môže nachádzať pach, ktorý je potreba refaktorovať. Report vytvára mnoho kontrol a jednou z nich je aj napr. duplicita kódu, čo je pach definovaný spomínaný v predchádzajúcich kapitolách, ale aj iné.

Základné typy kontrol (úplný zoznam viď [8]):

- prázdne bloky kódu,
- duplicity kódu,
- prekročenie max. počtu parametrov v metóde,
- nedodržiavanie konvencie pre pomenovávanie premenných/metód/tried,
- používanie magických čísel a iné.

Tento nástroj je taktiež podporovaný viacerými Java IDE, alebo je možné generovať reporty bez ich použitia.

3.4.3 Reporty pokrytia kódu

Tieto reporty slúžia pre prehľad celkového pokrytia zdrojového kódu testami. Tie poskytujú percentuálne zobrazenia pokrytia jednotlivých balíkov, alebo aj jednotlivých tried a ich riadkov. Najznámejším je nástroj Cobertura.

Cobertura

Je to voľne šíriteľný nástroj, ktorý počíta percento kódu použité pri spustených testoch. Využíva sa na zistenie, ktoré časti kódu boli pomocou testu použité, a ktoré test zasiahol. Tieto reporty umožňuje vytvárať vo formáte HTML alebo XML. Podľa [9], výsledky rozdeľuje pre celý projekt, jednotlivé balíky, triedy až samostatné riadky kódu.

Tieto reporty je možné generovať pomocou nástroja na preklad Java zdrojových kódov „ant“ alebo najnovšie „maven“. Bližšie informácie o tomto projekte sú dostupné v [9].

4 Analýza problematiky

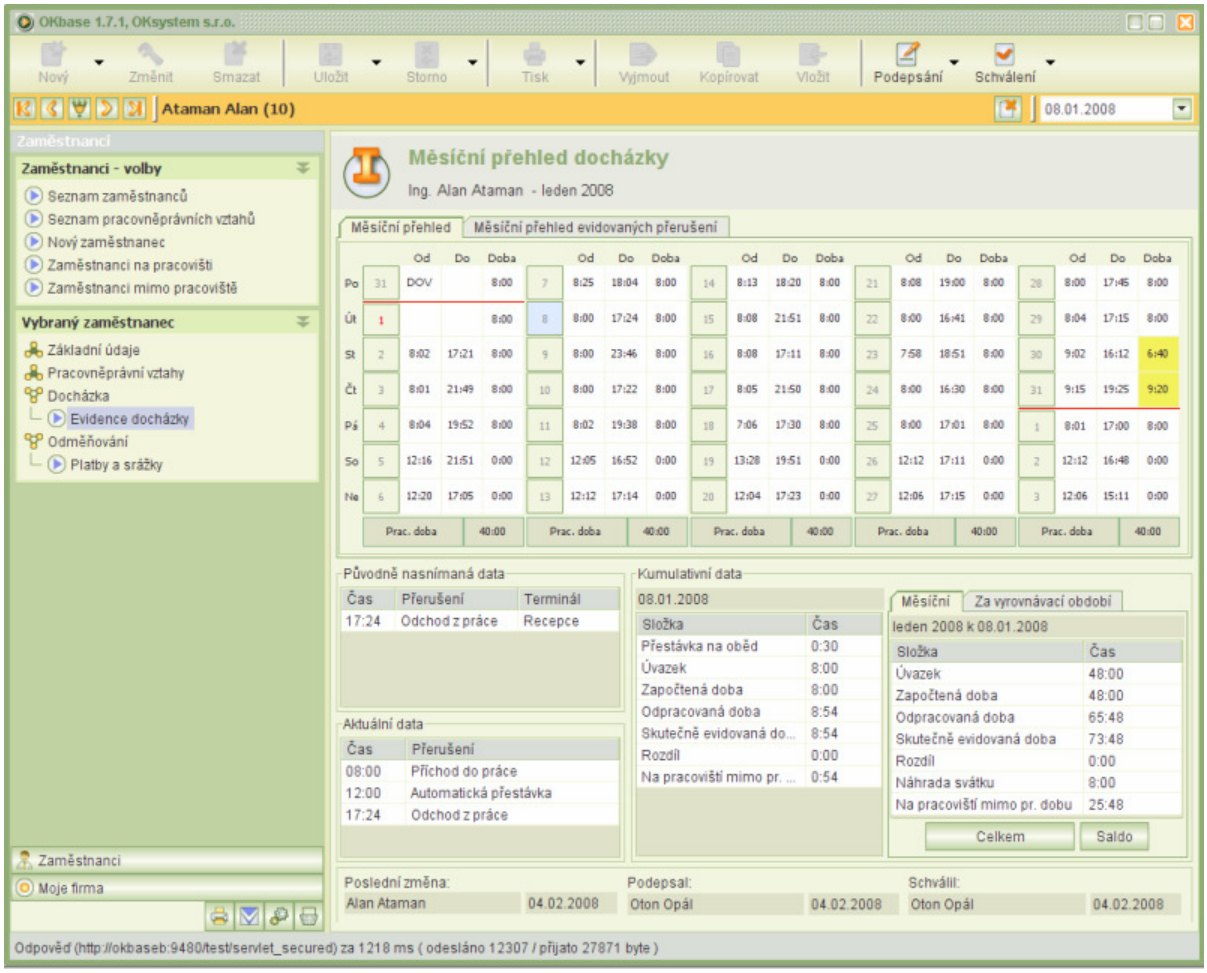
Pred samotným refaktoringom je potrebné zoznámiť sa s produktom, na ktorom sa bude vykonávať. V rámci tejto diplomovej práce bude refaktoring prebiehať na produkte OKbase firmy Oksystem s.r.o. S týmto programom je potrebné sa najprv zoznámiť ako s jeho použitím, tak aj štruktúrou DB, zdrojovými kódmi a technológiami používanými pri vývoji.

4.1 Popis systému OKbase

OKbase je softwarovým produktom firmy Oksystem s.r.o., ktorý je už niekoľko rokov nasadený v praxi, má svojich stálych používateľov a noví stále pribúdajú. Jedná sa o modulárny systém pre komplexné riadene ľudských zdrojov firmy, ich vonkajších vzťahov s partnermi, zákazníkmi a inštitúciami. Modularita umožňuje transparentné rozširovanie funkcií systému kedykoľvek po inštalácii v súvislosti s rastom potrieb firmy, alebo rozširovaním ponuky modulov, pretože sa tento systém stále vyvíja a rozširuje.

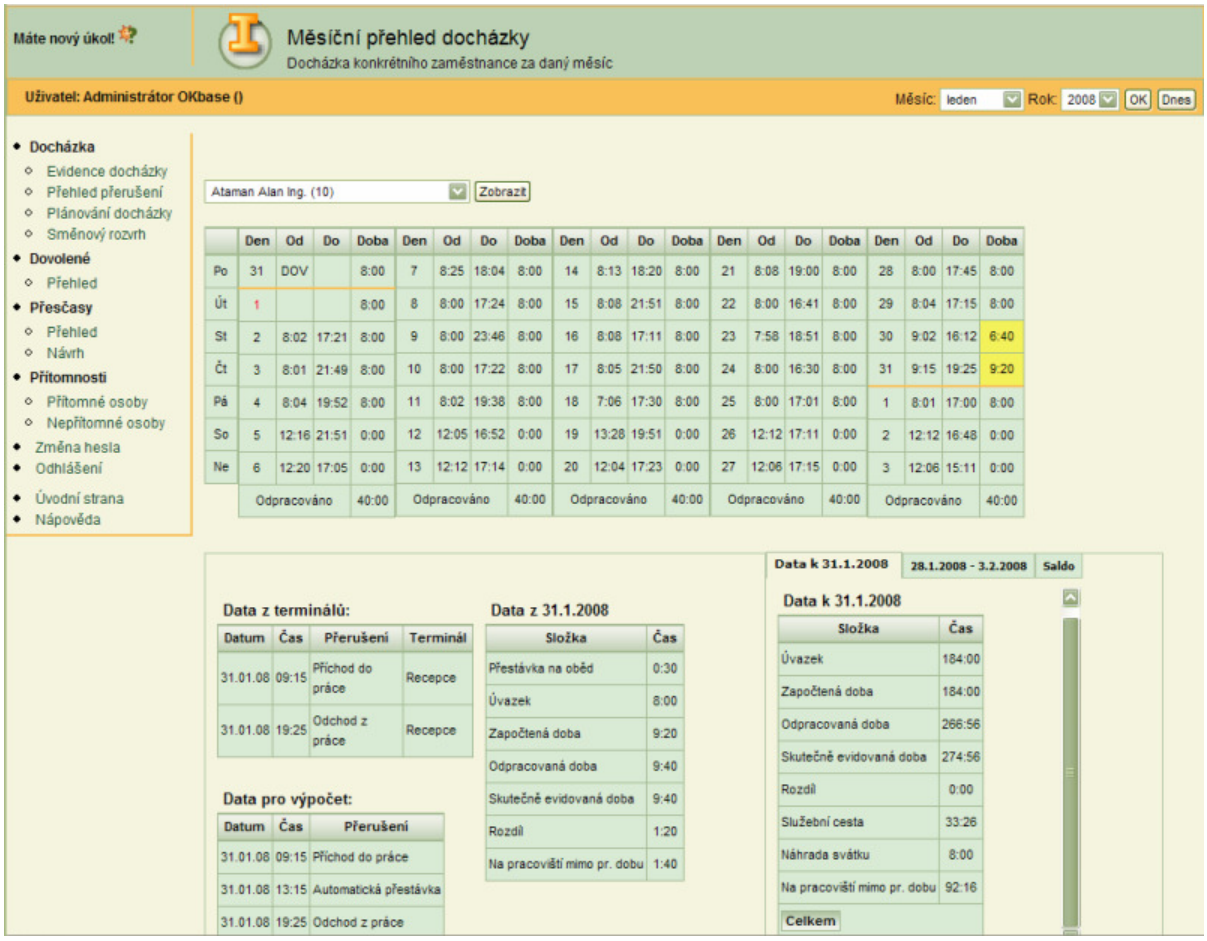
Všetky moduly pracujú nad jednotnou dátovou základňou. Je možné ich používať ako predinštalované, alebo sa môžu voľne konfigurovať podľa potrieb zákazníka. Pre prístup do systému je možné použiť webový, tzv. tenký a bohatý, tzv. plný klient. Ďalšou súčasťou je dochádzkový terminál.

Bohatý klient (viď Obr. 4-1) obsahuje plnú funkcionalitu správy systému. Implementuje správu všetkých modulov. Obrázok tohto klienta znázorňuje prehľad dochádzky za požadovaný mesiac v roku. Používajú ho hlavne vedúci pracovníci a personalisti.



Obr. 4-1: OKbase - bohatý klient.

Webový klient (viď Obr. 4-2) v aktuálnej verzii implementuje iba správu jedného z modulov a to dochádzkový. Obrázok znázorňuje prehľad dochádzky vybraného mesiaca. Slúži najmä na správu dochádzky, žiadosti a schvaľovanie dovoleniek a nadčasov a iné. Využívajú ho nadriadený aj podriadený zamestnanci v nasadenej firme.



Obr. 4-2: OKbase - webový klient.

4.1.1 Moduly OKbase

Ako bolo spomenuté systém OKbase je modulárny a obsahuje nasledujúce moduly:

- Modul H – obsahuje personalistiku, organizáciu a systematizáciu, vzdelávanie.
- Modul S – jedná sa o správu systému.
- Modul D – dochádzkový modul.

V najbližšej dobe sa modul H bude zásadne rozširovať o novú funkcionalitu.

Systémový modul S

Systémový modul je základná súčasť systému OKbase. Poskytuje funkcie pre beh všetkých modulov. Umožňuje konfiguráciu a administráciu celého systému. Poskytuje správu spoločných číselníkov, evidenciu užívateľov a začleňovanie ich do užívateľských skupín a pod. Stará sa o celkovú správu prideľovania práv a rolí užívateľom. Eviduje potrebné informácie pre vyhodnotenie bezpečnosti systému, audit, tlačové zostavy a ďalšie.

Dochádzkový modul D

Dochádzkový modul je nástrojom pre evidenciu pracovnej doby zamestnancov. Podľa platných právnych predpisov eviduje príchody a odchody zamestnancov vo zvolenom rozlíšení. Následne ich umožňuje vyhodnocovať, kontrolovať a schvaľovať. Monitoruje prítomnosť osôb na pracovisku a mimo pracoviska. Jeho súčasťou je terminál, na ktorom sa pomocou čipových kariet snímajú jednotlivé údaje. Tie je možno upravovať/dopĺňať aj pomocou webového či bohatého klienta.

Pomocou webového klienta ešte navyše umožňuje plánovanie, evidovanie a schvaľovanie dovoleniek a nadčasov.

Personálny modul H

Personalistika umožňuje evidovať a spravovať základné personálne údaje o osobách v pracovno-právnych vzťahoch o zamestnancoch alebo osobách v služobnom pomere vrátane bývalých. Ďalej o uchádzačoch o zamestnanie vrátane kvalifikačných predpokladov.

Umožňuje evidenciu základných informácii o danej firme (organizácii) vrátane organizačnej schémy a smenových kalendárov, evidenciu pracovných miest, činností a pracovných pozícii.

Personálny modul systému OKbase sa v najbližšej dobe plánuje rozširovať vo väčšom rozmedzí, a preto je potrebné ho pred týmito úpravami refaktorovať, či už celkovo alebo len postupne. Dôležité pritom je, aby sa stará funkcionalita nezaniesla chybami, pretože je tento produkt nasadený v praxi a používajú ho zákazníci firmy Oksystem s.r.o.

4.2 Implementácia OKbase

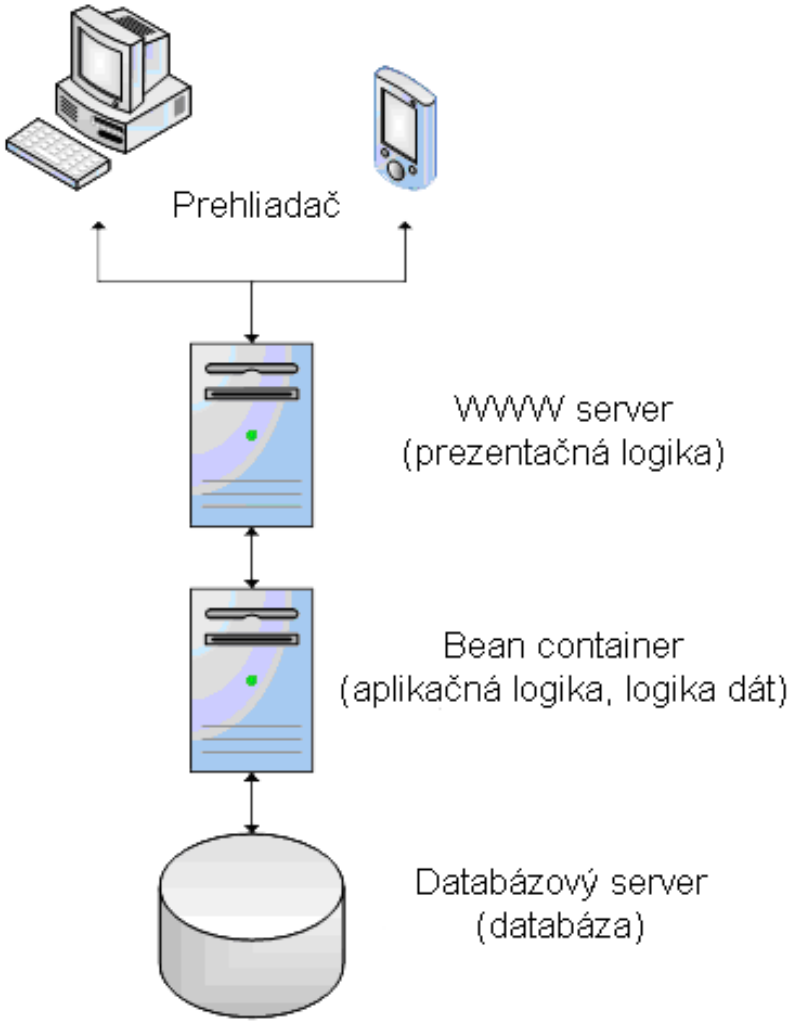
Systém OKbase je implementovaný vo viac-vrstvej architektúre, t.j. oddeľuje dátovú, aplikačnú a prezentačnú logiku. Celý systém je implementovaný v jazyku Java. Bohatý klient je implementovaný pomocou VVA frameworku vyvinutým firmou OKsystem. Webový klient pomocou technológie Struts 2. Tzv. business logika je sústredená na aplikačnom serveri. Využíva technológie J2EE a framework Spring v spojení s Hibernate. Tieto technológie sú použité v najnižšej vrstve a využívajú sa hlavne pri styku s databázou. Pomocou Hibernate sa databázové tabuľky prevádzajú do entít (Java tried) a dotazuje sa pomocou jazyka HQL[1] nad entitami uloženými v session. Do session sa entity načítajú z databázy.

4.2.1 VVA Framework

VVAF (Viac-Vrstvá Architektúra - Framework) je framework ň, vyvinutý firmou Oksystem s.r.o., ktorý sa využíva pri implementácii grafického rozhrania bohatého klienta.

[1] HQL (*H*ibernate Query Language) – dotazovací jazyk nad databázovými údajmi využívajúci entity

Využíva sa 4-vrstva architektúra, ako je vidieť na obrázku Obr.4-3.



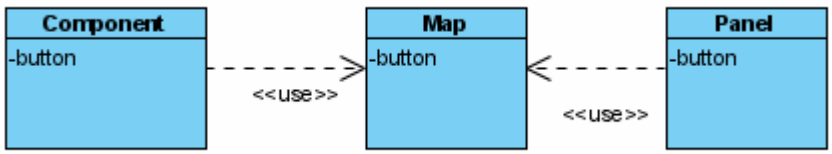
Obr. 4-3: Viacvrstvá architektúra VVAF [10].

V súčasnej verzii systému OKbase sa používa VVA framework 1.0. Jedná sa o knižnicu, ktorá umožňuje programátorovi jednoduchšie oddelenie prezentačnej logiky a prezentácie dát. Stará sa tiež o komunikáciu klienta so serverom.

Najdôležitejšou časťou tejto knižnice je komunikácia klienta so serverom. Pri spustení bohatého klienta pred sebou vidíme celý rad komponent. Nejedná sa iba o viditeľné grafické komponenty, ale tiež o dátové komponenty, ktoré nie je vidieť. Komponenty, ktoré sa za behu môžu meniť, alebo komponenty, ktoré sú pre každého užívateľa rôzne, majú tiež svoj obraz na serveri. Tento obraz je potomkom abstraktnej triedy `AbstractServerComponent`, ktorá pochádza z VVA knižnice.

Triedy nachádzajúce sa na serverovej strane obsahujú koncovku „`Component`“ alebo „`Controller`“. Starajú sa o prezentačnú logiku. Na klientskej strane sa nachádzajú triedy s koncovkou „`Panel`“, ktoré dávajú komponentom serverovej strany vzhľad a umiestnenie. Tieto

dve triedy sa spájajú pomocou triedy s koncovkou „Map“. Pomocou tejto mapovacej triedy je potrebné v serverovej aj klientskej triede zaregistrovať jednotlivé komponenty (viď Obr. 4-4).



Obr. 4-4: VVAF Component-Map-Panel.

Trieda `Panel` obsahuje iba grafické informácie o umiestnení a vzhľade danej komponenty a celého formulára. Trieda `Component` nastavuje a plní komponenty dátami. Stará sa o prezentačnú logiku zobrazovaného formulára, pričom využíva biznis vrstvu. Trieda `Map`, ako už bolo spomenuté slúži len pre spojenie týchto dvoch tried.

4.3 Refaktorovanie OKbase

Aplikácia refaktoringu na systéme OKbase prebieha vo viacerých fázach a to:

- pri pridaní novej funkcionality,
- pri oprave chýb,
- pri „*code review*“,
- celkový refaktoring.

Celkový refaktoring sa vykonáva nad personálnym modulom OKbase H.

4.3.1 OKbase H

Úlohou je popísať refaktoring nad týmto modulom. Keďže sa plánuje veľký zásah – rozšírenie tohto modulu, je potrebné vykonať aj celkový refaktoring.

Modul H sa vyvíjal už dávnejšie a obsahuje nečistoty zanesené neskúsenými programátormi. Pri tvorbe GUI sa pre tvorbu formulárov používal grafický návrhár, čoho následkom je nevyhovujúce pomenovanie inštančných premenných formulárov. Ďalším nedostatkom je nesprávne priradenie logiky k vrstve, t.j. v prezentačnej logike sa vykonávajú operácie patriace k biznis vrstve.

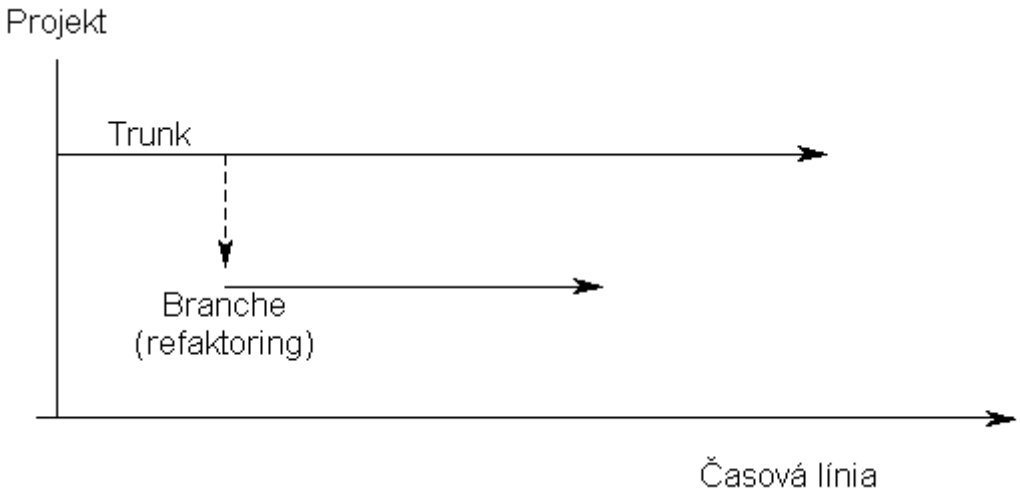
Tieto a ďalšie nedostatky je potrebné pomocou refaktorovania odstrániť. Popri tom je potrebné hľadieť na to, že tento produkt je nasadený v praxi.

4.3.2 Nasadenie v praxi

Pretože je tento systém nasadený v praxi je potrebné zabezpečiť, aby rôzne typy refaktorizácii neovplyvnili funkčnosť systému a nezaniesli do tohto systému nové chyby.

Najprv je potrebné pokryť celú oblasť základnými integračnými testami. Potom sa pri rôznych aplikáciách refaktoringu tieto testy doplnia špecifickými o funkčnosť, nad ktorou sa práve pracuje. Môžu sa tiež doplniť UNIT testami.

Ďalšou možnosťou ako zabrániť vneseniu chýb do ostrej vyvíjanej verzie je vytvoriť si samostatnú vetvu (tzv. *branche*) v SVN [1] repository (viď Obr.4-5) a refaktoring prevádzať nad touto vetvou. Aplikované zmeny v systéme sa otestujú dôkladne, a až potom bude možné tieto zmeny premietnuť do ostrej verzie.



Obr. 4-5: Vytvorenie SVN vetvy (*branche*).

4.3.3 Očakávané problémy

Základnou úlohou personálneho modulu je zobrazovanie a zadávanie personálnych dát. Z toho vyplýva, že sa jedná o modul, ktorý je pevne spojený s databázou aj napriek tomu, že sa používa objektovo relačné mapovanie na entity. To by mohlo spôsobovať problémy pri refaktorizácii. Ďalším problémom je rozhodnutie, kedy je daný kód pre refaktoring už nevyhovujúci a je potrebné cely kód prepísať a nerefaktorovať.

V neposlednom rade je otázka rýchlosti programu. Ak by sa zistilo, že aplikovaním refaktoringu sa systém viditeľne spomalil, je potrebné ho profilovať a na základe týchto údajov optimalizovať. Je to tiež dôležitá časť na základe toho, že je tento produkt nasadený v praxi a užívatelia sú zvyknutí na určitú odozvu programu.

[1] SVN - SubVersion, verzovací systém pre správu zdrojových kódov

5 Aplikácia refaktoringu

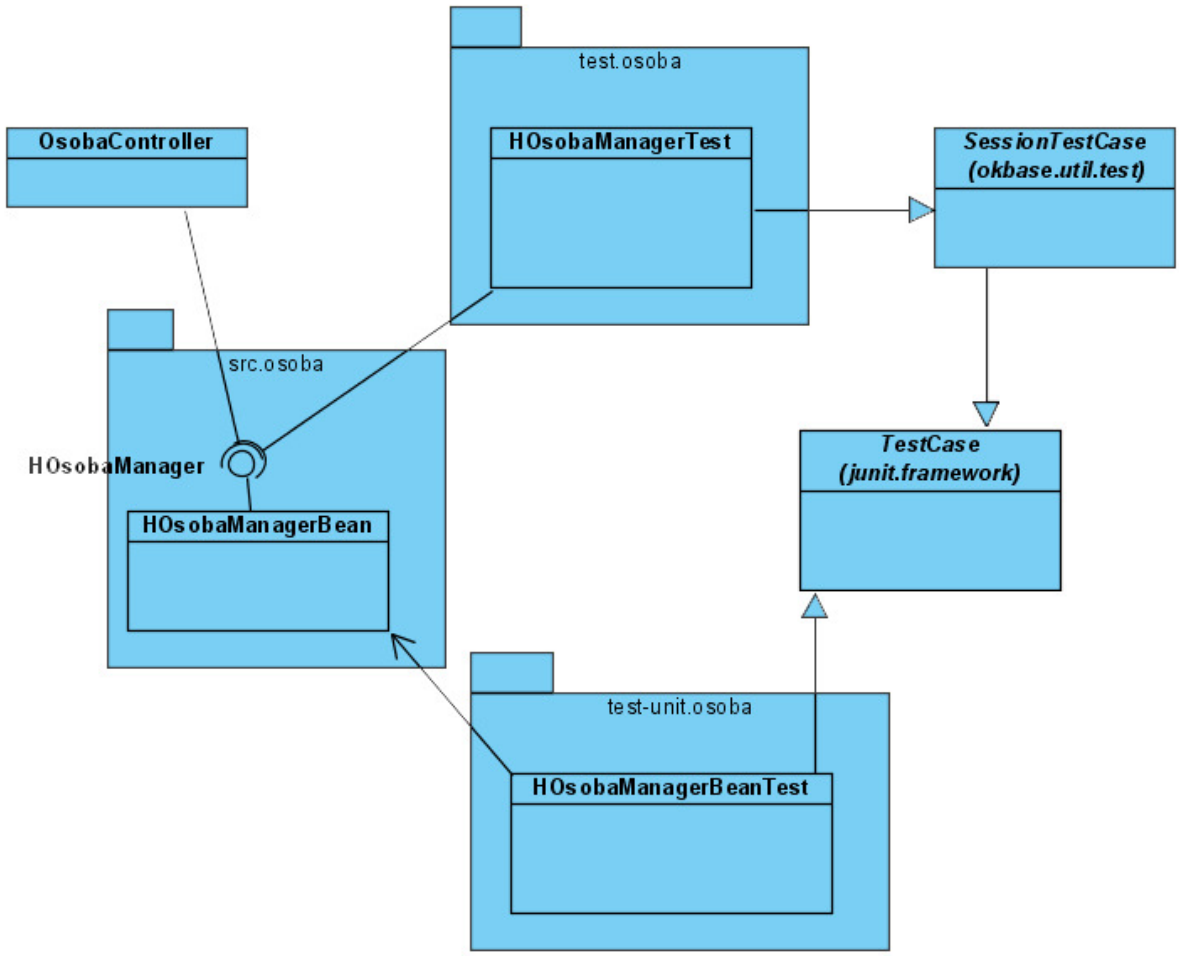
Refaktoring som aplikoval na už spomínaný systém OKbase, ktorý je v praxi používaný užívateľmi. Jednalo sa o celkový refaktoring personálneho modulu OKbase H, taktiež čiastkové refaktoringy boli premietnuté na základe dobrého postupu aj na modul OKbase D (dochádzka).

Kvôli zabezpečeniu nevnášania chýb pri refaktoringu je potrebné celý systém pokryť testami, pomocou ktorých bude podchytená aktuálna funkčnosť systému.

5.1 Tvorba testov

Venovanie sa tejto problematike je potrebné pre zabezpečenie správnosti refaktoringu, pretože sa program v praxi používa. Nie je možné aby sa do systému počas refaktoringu zaniesla chyba, ktorá by nebola podchytená a opravená.

Systém sa začal pokrývať základnými integračnými testami, ktoré zachycovali jeho vnútornú funkčnosť. Tie boli počas refaktorovania rozširované o dôkladnejšie testovacie metódy a dopĺňané jednotkovými testami.



Obr. 5-1: Diagram použitia testovacích tried.

Na obrázku Obr.5-1 je zobrazené použitie testovacích tried. Integračné testy sa nachádzajú v balíku `test`, ktoré pri spustení napodobňujú beh celej aplikácie, t.j. zostaví sa celý back-end (business logika) aplikácie. Takýto test sa napája na prázdnu databázu, kde hodnoty pre aktuálny test sa nachádzajú v popisnom XML ako zobrazuje Príklad 5-2. Pre test je možné načítať jeden z globálnych XML súborov popisujúcich DB, k čomu je možné pripojiť prednastavený XML súbor s rovnakým názvom ako je názov testu viď Príklad 5-1.

```
public class HPracovnikManagerTest extends SessionTestCase {

  private HPracovnikManager pracovnikManager;

  public void setPracovnikManager(HPracovnikManager pracovnikManager) {
    this.pracovnikManager = pracovnikManager;
  }

  @Override
  protected IDataSet getDataSet() throws DataSetException {
    List<String> dataSetNames = new ArrayList<String>();
    dataSetNames.add(XmlDataSetNames.HOSOBA.getLocation());

    return getCompositDataSet(dataSetNames, true);
  }

  public void testNactiPodleId() {
    HPracovnikData pracovnikData = pracovnikManager.nactiPodleId(9001L);

    assertNotNull(pracovnikData);
    assertEquals(9001L, pracovnikData.getId().longValue());
    assertEquals("1", pracovnikData.getOsobniCislo());
    assertEquals(false,
          pracovnikData.getLogickyZrusen().booleanValue());
    assertEquals(0, pracovnikData.getZapocetDoba().intValue());
  }

  public void testUloz() {
    HPracovnikData pracovnikData = new HPracovnikData();
    pracovnikData.setLogickyZrusen(false);
    pracovnikData.setZapocetDoba(0);
    pracovnikData.setOsobniCislo("111");
    pracovnikData.setId(9000L);
    pracovnikManager.vloz(pracovnikData);

    HPracovnikData pracovnik = pracovnikManager.nactiPodleId(9000L);
    assertEquals(pracovnikData.getLogickyZrusen(),
                 pracovnik.getLogickyZrusen());
    assertEquals(pracovnikData.getZapocetDoba(),
                 pracovnik.getZapocetDoba());
    assertEquals(pracovnikData.getOsobniCislo(),
                 pracovnik.getOsobniCislo());
    assertEquals("111", pracovnik.getOsobniCislo());
    assertEquals(pracovnikData.getId(), pracovnik.getId());
  }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
	<!-- ============================================ -->
	<!--  H_PRACOVNIK                                 -->
	<!-- ============================================ -->
	<table name="H_PRACOVNIK">
		<column>H_PRAC_ID</column>
		<column>OSOBNI_CISLO</column>
		<column>ZAPOCET_DOBA</column>
		<column>LOGICKY_ZRUSEN</column>
		<row>
			<value>9001</value>
			<value>1</value>
			<value>0</value>
			<value>0</value>
		</row>
		<row>
			<value>9002</value>
			<null/>
			<value>0</value>
			<value>0</value>
		</row>
	</table>
</dataset>
```

Príklad 5.2: XML súbor popisujúci DB (`HPracovnikManagerTest.xml`)

Integračný test (viď Príklad 5-1) dedí zo `SessionTestCase`, ktorý sa stará o ustanovenie session a spojenie s databázou. Integračné testy sa pripájajú na prázdnu databázu, pričom údaje sa načítavajú z popisných XML súborov ako zobrazuje Príklad 5-2. Tento súbor popisuje dáta pre tabuľku H_PRACOVNIK podľa jej štruktúry. Pri vytváraní viacerých testov sa zistilo, že sa často využívajú tie isté dáta ako napríklad údaje o osobe, bol vytvorený spôsob ako umožniť načítanie týchto dát. To sa deje v preťaženej metóde `getDataSet`, ktorá načíta zoznam globálnych prednastavených XML súborov. Metóda vracia údaj o obsahu databázy, pričom to či chceme použiť aj popisné XML daného testu sa určí `boolean` parametrom v metóde `getCompositDataSet`.

V tomto prípade sa načíta aj popisné XML tohto integračného testu. Príklad 5-2 zobrazuje jeho štruktúru. Na začiatku je hlavička popisujúca typ súboru. Potom sa popisuje `dataset`, ktorý obsahuje jednotlivé tabuľky. Tie najprv popisujú štruktúru tejto tabuľky (tá musí súhlasiť so štruktúrou s prázdnou DB) a následne popis jednotlivých riadkov čiže hodnôt tabuľky. Ak sa v súbore nachádza viac tabuliek, je dôležité ich poradie, pretože sa dodržiavajú databázové väzby (*constraints*).

Ako už bolo spomenuté v popise testov, pred spustením každého testu (metódy v testovacej triede) sa zostavuje databáza znova. Celkové spustenie testov a priebeh integračného testu preto prebieha dlhšiu dobu.

5.2 Pri oprave chyby

Refaktoring pri oprave chyby má za úlohu zjednodušiť programátorovi prácu, ale zároveň ho nesmie zdržiavať. Taktiež musí byť zrejmý rozdiel medzi opravou chyby a refaktorovaním. Preto sa opravovaný kód najprv refaktoruje, potom sa opravuje chyba. Môže sa stať, že je potrebné najprv opraviť chybu, tým znečistiť kód viac, ale následne sa kód sprehľadní. Podobné podmienky platia aj pri refaktoringu pred pridaním novej funkcionality.

Úlohou typu tohto refaktoringu je aby zmeny, ktoré sa majú vykonať boli vykonané správne a bez chyby. To je podmienené dobrou orientáciou sa v danom kóde a následne jeho správnym pochopením.

Tab. 5-1: Možné nečistoty kódu pri oprave chyby.

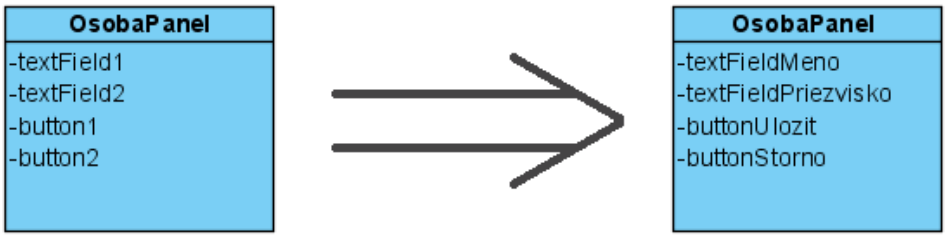
Zápach	Refaktorovanie
Duplicitný kód	Vybrať metódu (121)
Dlhý zoznam parametrov	Nahradiť parameter metódou (272), Zaviesť objekt pre parametre (275)
Dlhá metóda	Vybrať metódu (121), Nahradiť dočasnú premennú dotazom (129), Rozložiť podmienku (227)
Komentáre	Vybrať metódu (121)

Najčastejšie vyhľadávané zápachy kódu znázorňuje Tab.5-1. Pri každej nečistote kódu je uvedený možný typ refaktoringu na základe katalógu refaktorovaní [1].

Na základe použitého VVA frameworku sa v systéme OKbase nachádzajú špecifické nečistoty v jednotlivých vrstvách, t.j. typoch tried.

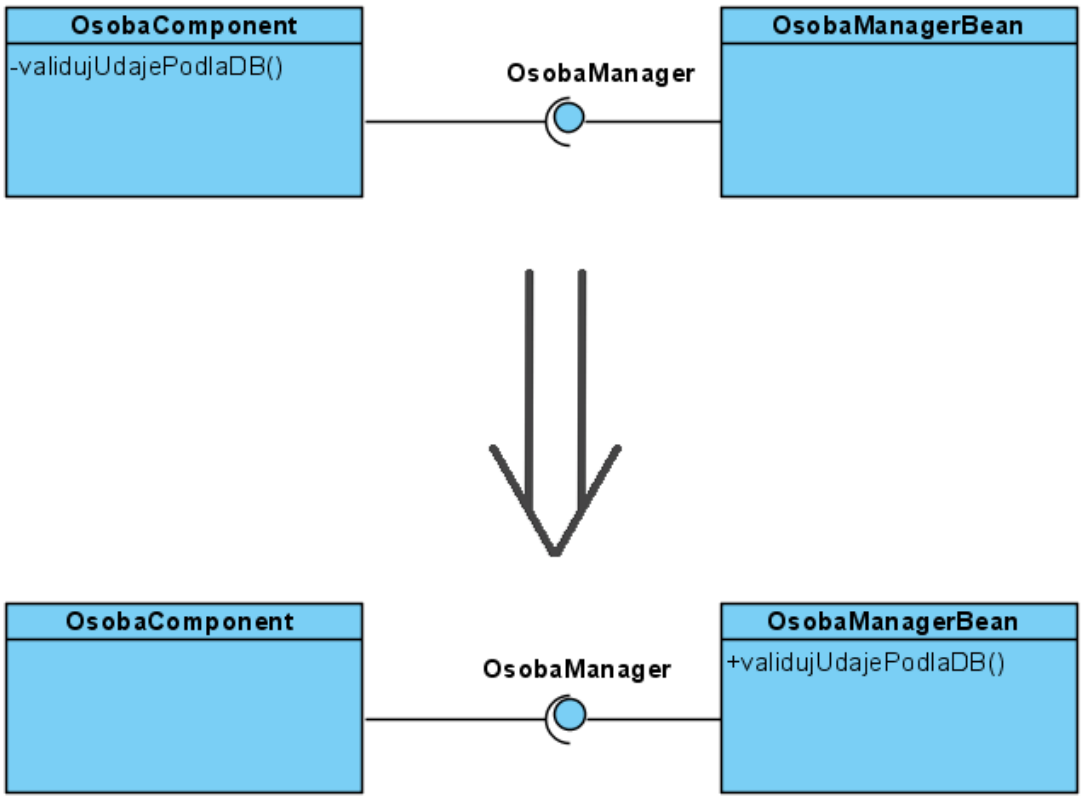
Špecifické problémy na projekte OKbase:

- **Trieda typu „Panel“** – na základe pozostatkov, keď sa formuláre generovali IDE prostredím, zostali niektoré premenné, položky a metódy pomenované na základe typu panelu a číselnej hodnoty. Pri zložitejších formulároch sa v takomto kóde ťažko orientuje. Preto je pre lepšie pochopenie kódu potrebné premenovať jednotlivé položky a metódy na základe toho čo vyjadrujú. Tento refaktoring zjednodušene znázorňuje Obr.5-2.



Obr. 5-2: Zmeny názvov položiek v triedach typu „Panel“.

- **Trieda typu „Component“** – taktiež nevyhovujúce pomenovania položiek, premenných a metód. Navyše nesprávne umiestnenie aplikačnej logiky vo vrstve prezentačnej logiky (triedach typu „Component“). Takéto metódy je potrebné presunúť do nižšej aplikačnej vrstvy (zjednodušene viď Obr.5-3) a pri prípadnej potrebe sprístupniť aj iným triedam pomocou interface.



Obr. 5-3: Presun metódy do aplikačnej logiky.

Tieto metódy refaktorovaní a typy pachov sú podobné s refaktorovaním pri pridávaní novej funkcionality. Pri pridávaní novej funkcionality sa programátor ešte musí zamerať na dopad pridanej funkcionality na časť dopĺňaného kódu.

5.3 Pri kontrole kódu

Kontrola kódu alebo tiež „code review“ je praktika prebraná z agilného prístupu k projektu. Ako bolo spomenuté, jedná sa o kontrolu kódu po napísaní iným programátorom, pred vložením zmeny do repository. Cieľom tejto kontroly je lepšia štruktúra kódu a jeho určitá jednoduchosť, t.j. ak danú zmenu pochopí kontrolór (programátor prizvaný na kontrolu kódu) a nie len implementátor zmeny, je väčšia šanca, že pri zmene alebo oprave chyby tento kód pochopí aj niekto iný.

Pri kontrole sa programátori nesústredia a nedržia pachov, ale kód porovnávajú na základe jeho štruktúry a jednoduchosti. Využívané typy refaktorovaní popisuje Tab.5-2.

Tab. 5-2: Najčastejšie typy refaktorovaní pri „*code review*“.

Refaktorovanie	**Zápach/Problém**
Vybrať metódu (121)	Duplicitný kód, Zložité podmienky
Nahradiť dočasnú premennú dotazom (129)	Dlhá metóda
Rozložiť podmienku (227)	Zložité podmienky, Dlhá metóda
Spojiť podmienené výrazy (229)	Zložité podmienky
Zaviesť vysvetľujúcu premennú (133)	Zložité podmienky

Najčastejšie sa využívajú typy refaktoringu z katalógu na zjednodušenie podmienených výrazov. Tie súvisia tiež s pachom duplicitný kód, pretože niektoré podmienky sa v triede môžu vyskytovať viackrát v rovnakej forme (viď Príklad 5-3). Potom je dobré použiť refaktoring Zaviesť vysvetľujúcu premennú (133) s postupom Vybrať metódu (121).

Príklad 5-3 zobrazuje výhodu extrakcie podmienky do privátnej metódy triedy. Pred code review obsahoval kód 2 duplicity tej istej podmienky. Navyše tretí výskyt bol chybný. Tým že sa podmienka extrahovala do privátnej metódy `maKodParovehoPreruseni(...)` sa odstránila duplicita a zabránilo vneseniu chyby do kódu. To bolo vedľajším účinkom ako code review tak aj daného refaktoringu, pretože k zabráneniu vnášania takýchto chýb by mali slúžiť testy.

```
protected void nactiSeznamPovolenychParovychPreruseni(String
                  typPreruseni){
    if(maKodParovehoPreruseni(typPreruseni)) {
      // nacitaj a spracuj prerusenia
    } else {
      // vytvor nove
    }
}

private void nastavEditovatelnost(boolean edit) {
    // ...
    choiceComponent_kodParPreruseni.setPristupny(edit &&
                  maKodParovehoPreruseni(preruseni.getTyp()));
    // ...
}

public void prirad(ObdCPrerusFullData preruseni) {
    // ...
    nactiSeznamPovolenychParovychPreruseni(preruseni.getTyp());
    if (maKodParovehoPreruseni(preruseni.getTyp())) {
      priradKodParovehoPreruseni();
    }
    // ...
}

private boolean maKodParovehoPreruseni(String typPreruseni) {
    return ObdKonstanty.PRERUSENI_PRICHODOVE.equals(typPreruseni) ||
                  ObdKonstanty.PRERUSENI_ODCHODOVE.equals(typPreruseni);
}
```

Príklad 5-2: Výsledok extrahovania podmienky do metódy.

5.4 Celkový refaktoring

Jednou z výhod zoznamovania sa s aplikáciou pomocou tvorby testov, spojeným s pokrývaním systému testami, je nachádzanie pachov a zle štruktúrovaných časti kódu.

Na nasledujúcom príklade je znázornený postup refaktorizácie tela metódy, ktorej úlohou je kontrolovať určitý vzťah k požadovanému dátumu. Jedná sa o úpravu zle čitateľnej podmienky a odstránenie zbytočných komentárov. V metóde je použitý ternárny operátor pre podmienku, čo v tom to prípade nie je najlepším riešením. Kód je nečitateľný a je potrebné používať dvojnásobnú kontrolu lokálnej premennej `datumDo` na `null`. Kód metódy pred refaktorizáciou znázorňuje Príklad 5-4. Podrobný postup úpravy tela metódy znázorňujú Príklad 5-5 až Príklad 5-7. Časť integračného testu zachycujúci funkčnosť metódy zobrazuje Príloha A.

```
public boolean zkontrolujZmenuPlatnosti(Long idVztahu, Date datumOd) {
    if (idVztahu == null || datumOd == null) {
      throw new IllegalArgumentException("Id vztahu a datum od nesmí být
            null.");
    }

    // zajisti, aby datum ukazovalo na začátek dne
    Date platnostOd = DateHelper.getDatumNaDny(datumOd);

    // projdi seznam zařazení a pokud najdeš potencionální konflikt,
       vrať false
    List<HZarazeniMistoData> list =
            hZarazeniMistoManager.nactiPodleVztahu(idVztahu);
    for (HZarazeniMistoData data : list) {
      Date datumDo = data.getDatumDo() == null ? null :
            DateHelper.getDatumNaDny(data.getDatumDo());
      if (datumDo != null && datumDo.before(platnostOd)) {
        result = false;
        break;
      }
    }
    return result;
}
```

Postup refaktorizácie:

1. Vytvorenie testu pre zachytenie funkcie metódy (viď Príloha A).
2. Premenovanie lokálnej premennej `result` na `vysledek`, pretože je to zaužívané pomenovanie v projekte OKbase.
3. Odstránenie dvojnásobnej kontroly na `null` presunutím podmienky a výpočtu premennej `datumDo`.

```
if (datumDo != null) {
    datumDo = DateHelper.getDatumNaDny(datumDo);
    if (datumDo.before(platnostOd)) {
        vysledek = false;
        break;
    }
```

4. Nahradenie premennej `datumDo` výpočtom (pomocou refaktorovania IDE Eclipse - *inline*).

```
if (DateHelper.getDatumNaDny(datumDo).before(platnostOd)) {
      vysledek = false;
    break;
}
```

5. Použitie konjunkcie podmienok (viď [1, str. 231]) a odstránenie lokálnej premennej `vysledek`, pretože nie je potrebná a zvýši to prehľadnosť v kóde.

```
public boolean zkontrolujZmenuPlatnosti(Long idVztahu, Date datumOd) {
    if (idVztahu == null || datumOd == null) {
      throw new IllegalArgumentException("Id vztahu a datum od nesmí být
            null.");
    }

    // zajisti, aby datum ukazovalo na začátek dne
    Date platnostOd = DateHelper.getDatumNaDny(datumOd);

    for (HZarazeniMistoData data :
            hZarazeniMistoManager.nactiPodleVztahu(idVztahu)) {
       Date datumDo = data.getDatumDo();
       if (datumDo != null &&
         DateHelper.getDatumNaDny(datumDo).before(platnostOd)) {
            return false;
      }
    }
    return true;
  }
```

Príklad 5-6: Výsledok refaktorizácie metódy.

Tento jednoduchý príklad zahŕňa celkový postup, ktorý je potrebný dodržiavať pri refaktoringu. Keďže je systém nasadený v praxi, základom je vytvorenie testu, ktorý zachytáva funkčnosť metódy. Ten je potrebný pre zabránenie vnesenia chyby do programu. Na testy popisujúce funkčnosť sa nesmie zabúdať.

6 Výsledky aplikácie refaktoringu

Základom úspešného refaktoringu je dôkladná analýza refaktorovaného programu. Systém OKbase bol analyzovaný podrobne, pričom boli určené miesta s možnými pachmi kódu. Väčšinou sa jednalo o umiestnenie logiky programu do nesprávnej vrstvy podľa architektúry. Základom refaktoringu pre zabezpečenie funkčnosti bola tvorba testov popisujúcich tento systém, kvôli nasadeniu programu v praxi.

6.1 Testy

Tvorba testov mala hneď niekoľko účelov:

1. zabezpečenie funkčnosti počas refaktoringu,
2. bližšie sa zoznámenie so systémom OKbase a jeho pochopenie,
3. používanie testov aj v budúcnosti.

Vytvorenie testov pred refaktorovaním, je z časového aj psychického hľadiska programátora náročné. Nielenže sa musí zoznamovať s kódom a systémom, ale robí činnosť, pri ktorej nevidí skoro žiadne výsledky a neimplementuje ani žiadnu novú funkcionalitu. Generovanie reportov pomocou nástroja Cobertura trvá tiež určitú dobu a odvádza od pozornosti programátora. Z tohto dôvodu bolo do vývojového prostredia IDE Eclipse nainštalované rozšírenie „EclEmma“ (*Java Code Coverage for Eclipse* URL <www.eclemma.org>), ktoré umožňuje automatické a rýchle generovanie testovacích reportov pokrytia priamo v IDE Eclipse. Dokonca priamo v zdrojovom kóde rozlišuje riadky farebne podľa toho, či cez ne test prešiel alebo nie. Takto má programátor nad pokrytím kódu testami počas vývoja úplnú kontrolu.

Coverage Report - cz.oksystem.okbase.okbaseh.service.osoba

Package	# Classes	Line Coverage	Branch Coverage	Complexity
cz.oksystem.okbase.okbaseh.service.osoba	3	28% 133/479	14% 40/278	0
cz.oksystem.okbase.okbaseh.service.osoba.adresa	2	43% 12/28	50% 2/4	0
cz.oksystem.okbase.okbaseh.service.osoba.bkspoj	2	29% 9/31	17% 1/6	0
cz.oksystem.okbase.okbaseh.service.osoba.duchod	2	58% 22/38	100% 4/4	0
cz.oksystem.okbase.okbaseh.service.osoba.pracovnik	1	72% 36/50	33% 6/18	0
cz.oksystem.okbase.okbaseh.service.osoba.pracovnik.oscislo	2	38% 18/48	14% 2/14	0
cz.oksystem.okbase.okbaseh.service.osoba.prijmeni	2	46% 13/28	100% 2/2	0
cz.oksystem.okbase.okbaseh.service.osoba.prislusnici	3	26% 15/58	17% 1/6	0
cz.oksystem.okbase.okbaseh.service.osoba.prukaz	4	49% 28/57	83% 5/6	0
cz.oksystem.okbase.okbaseh.service.osoba.socpoj	2	41% 12/29	100% 2/2	0
cz.oksystem.okbase.okbaseh.service.osoba.udaj	1	100% 12/12	100% 2/2	0
cz.oksystem.okbase.okbaseh.service.osoba.zamestnani	5	48% 73/151	45% 28/62	0
cz.oksystem.okbase.okbaseh.service.osoba.zdrav	2	43% 12/28	100% 2/2	0
cz.oksystem.okbase.okbaseh.service.osoba.zdrpoj	2	41% 12/29	100% 2/2	0

Obr. 6-1: Report pokrytia kódu testami vytvorený nástrojom Cobertura

Bolo vytvorených 30 integračných testov pre modul H, ktoré sa môžu a už sa aj využívajú popri vývoji. Testy pokrývajú modul z 28%. Obr.6-1 znázorňuje ukážku reportu pokrytia kódu pre balík osoba. To znamená, že pokrytie pred refaktoringom nemalo účel len počas refaktorovania kódu, ale svoj účel zohrávajú naďalej počas vývoja, kde pri zmenách kódu zaručujú jeho funkčnosť aj naďalej. Po určitých cykloch sa tieto testy spúšťajú a uvažuje sa o ich spúšťaní aj pred každým vkladaním väčších zmien v kóde. Jednotkové testy sa spúšťajú vždy pred vkladaním zmien do SVN, pretože tie nie sú na čas náročné.

6.2 Refaktoring

Úlohou refaktorovania systému OKbase bolo zlepšiť jeho štruktúru kódu. Jednalo sa hlavne o personálny modul OKbase H, ktorý sa bude rozširovať. Refaktoring prebiehal vo viacerých fázach a možnostiach refaktoringu.

Nad systémom OKbase boli vykonané tieto možnosti refaktorovaní:

- refaktoring pri oprave chyby,
- refaktoring pri „*code review*“,
- celkový refaktoring.

Refaktorovanie modulu OKbase H, prebiehalo s dobrými výsledkami, a preto sa povolila možnosť skúmania aj refaktoringu pri oprave chyby, alebo možnosť „*code review*“ s refaktoringom pri kontrole kódu, ktoré prebiehali nad modulom OKbase D (dochádzka) a OKbase S (správa systému).

Refaktoring pri oprave chyby

Ak sa nájde chyba v programe je potrebné ju odstrániť čo najrýchlejšie, hlavne ak je systém nasadený v praxi. Zároveň sa oprava musí vykonať poctivo, aby sa v budúcnosti neopakovala a oprava nezaniesla do systému novú chybu.

V [11, str. 587] sa uvádza, že pri oprave jednoduchých chýb má programátor veľkú šancu k tomu aby sa dopustil chyby (až 50%). Ak sa zvyšuje počet riadkov opravy, zvyšuje sa aj pravdepodobnosť novej chyby. Lenže pokiaľ sa zvyšuje počet riadkov ďalej, pravdepodobnosť zanesenia chyby začne výrazne klesať. Jedným z príznakov je pomerne bezstarostný prístup programátorov k jednoduchým zmenám.

V systéme OKbase sa osvedčilo pri oprave chyby refaktorovať nejasné časti kódu, hlavne premenovanie metód či zjednodušenie podmienok, alebo ich extrahovanie. Programátor tak získa väčší prehľad o opravovanej časti kódu a pomáha mu zároveň pri zoznámení sa s kódom. Takto aj pri oprave jednoduchej chyby sa programátor snaží pochopiť celkovú funkčnosť bloku systému. Zabráni sa tým vnášaniu nových chýb pri oprave jednoduchých.

Refaktoring pri kontrole kódu

Je ťažké presvedčiť manažérov, alebo vrcholové predstavenstvo o tom, že po každej zmene by sa mala vykonávať kontrola zmeny kódu iným programátorom. Dá sa povedať, že sa jedná o určitý druh prevencie pre výskyt chyby v programe, ale nie je to jeho hlavnou úlohou. Tou je hlavne zamedzenie vloženia zmeny alebo novej časti kódu, ktorá nie je dobre štruktúrovaná, alebo čitateľná.

Výsledkom nasadenia „*code review*“ a refaktorovania s ním spojeným je zvýšenie kvality kódu, ale aj kvality jednotlivých programátorov. Je to niečo podobné ako párové programovanie, ale nasadené už na hotový výsledok (zmenu). Programátori si tak ujasňujú čo je výsledkom opravy, jednotlivú štruktúru opravy a zmeny, ktoré je možné na danom kóde vykonať. Jednoducho povedané kód sa stane dvakrát tak kvalitným a čitateľnejším pre väčšiu časť programátorov.

Refakotrovanie pri kontrole kódu v systéme OKbase dosahovalo veľmi dobré výsledky. Boli časti kde refaktorovanie nebolo potrebné a kód bol pri kontrole potvrdený. Väčšinou šlo o úpravu podmienok, kontrolu duplicít a dlhých metód.

Následky refaktoringu pri „*code review*“:

- odstránenie duplicít v kóde,
- zjednodušenie podmienok,
- zlepšenie významu metód (dlhé metódy),
- lepšia štruktúra a čitateľnosť kódu.

Vedľajším efektom tohto refaktoringu je zamedzenie vkladania chýb a zvyšovanie kvality programátorov, pretože sa jeden programátor učí praktiky od druhého na praktických ukážkach.

Celkový refaktoring

Pri celkovom refaktoringu je najväčší problém kde začať. K tomu sú nápomocné technológie popísané v kapitole 3 Technológie pre podporu refaktoringu. Osvedčili sa reporty duplicít kódu a ďalších pachov v kóde. Najviac sa osvedčil prístup, keď bolo potrebné pochopiť funkčnosť aplikácie a pokryť systém testami. Tie boli potrebné pre zachovanie funkčnosti systému, keďže je nasadený v praxi.

Pri tvorbe testov boli nájdené nedokonalosti a nečistoty v kóde, ktoré boli zapisované a neskôr refaktorované. Príklady Príklad 5-4 až Príklad 5-7 zobrazujú jednoduchý postup pri refaktorovaní, ktorého dôležitou súčasťou je aj vytvorenie testu zachytávajúci funkciu pred refaktorovaním. Tvorba testov v tomto prípade nesmie byť podceňovaná, pretože testy, ktoré boli počas refaktorovaní vytvorené zachytili zanesené chyby. Tie boli opravené a zároveň sa už používajú pri vývoji a oprave chýb dopĺňaním a úpravou.

Celkový refaktoring prispel k lepšej štruktúre a čitateľnosti modulu OKbase H systému OKbase, pred plánovaným rozširovaním funkčnosti celého systému. Vďaka tomu budú programátori schopní zahrnúť opravy bezpečnejším a predpokladám aj rýchlejším spôsobom.

6.3 Vzniknuté problémy

Pri každom refaktorovaní vznikne určitý problém, s ktorým je potrebné si poradiť. Pri refaktorovaní systému OKbase sa z predpokladaných chýb vyskytla len previazanosť modulu OKbase H s databázou. Znížená rýchlosť programu sa nevyskytla, a preto nebolo potrebné program profilovať a hľadať dôvod jeho spomalenia.

Vzniknuté problémy pri refaktorovaní systému OKbase:

1. **Problém opätovného vracania rovnakej chyby** – pri oprave chyby bol spôsob načítania osôb refaktorovaný do samostatnej metódy. Pri oprave inej chyby (iným programátorom) bol zmenený spôsob načítania osôb na základe určitej podmienky, kde nebola použitá extrahovaná metóda. Takto sa chyba vrátila ešte raz, až sa zistila príčina. Chyba bola odstránená zjednotením načítania osôb do jednej metódy a presunutá do aplikačnej vrstvy, pretože sa nachádzala v logike prezentačnej vrstvy, kde nepatrila.
2. **Problém SVN synchronizácie vetvy pre refaktoring** – jedná sa o problém malých intervalov synchronizácie zmien medzi hlavnou vývojovou vetvou a vetvou vytvorenou pre refaktoring. V hlavnej vývojovej vetve došlo v určitých častiach kódu k veľkým zmenám, čo malo za následok nevyužitia časti refaktoringu vykonanom vo vetve pre refaktoring. Testy bolo možné po miernej zmene synchronizovať.
3. **Refaktorovanie spojené s technológiou Hibernate** – tento problém sa vyskytol pri zmenách entít, ktoré sa tiež používajú v HQL dotazoch, nachádzajúcich sa vo forme typu `String`. Pri použití refaktorovania pomocou IDE Eclipse 3.3, sa zmeny do reťazcov nezahrnuli. Takáto chyba bola zachytená testami. Pre odstránenie bolo potrebné vyhľadať a upraviť všetky výskyty jednotlivých zmien.

Prvá chyba bola spôsobená chybným prístupom pri refaktorovaní, kde nebol zohľadnený celý kód. Svoju zásluhu na tom mal aj programátor, ktorý sa oprave chyby nevenoval dostatočne, čo potvrdilo predpoklad, že aj pri jednouchej oprave chyby je veľké riziko zanesenia novej.

Druhá chyba spočívala v slabej synchronizácii v SVN medzi hlavnou vývojovou vetvou a vetvou vytvorenom pre refaktoring. Táto vetva bola vytvorená kvôli tejto diplomovej práci. V reálnom praktickom nasadení by sa vetva nevytvárala, poprípade by k synchronizácii dochádzalo častejšie na základe potrieb pre refaktoring.

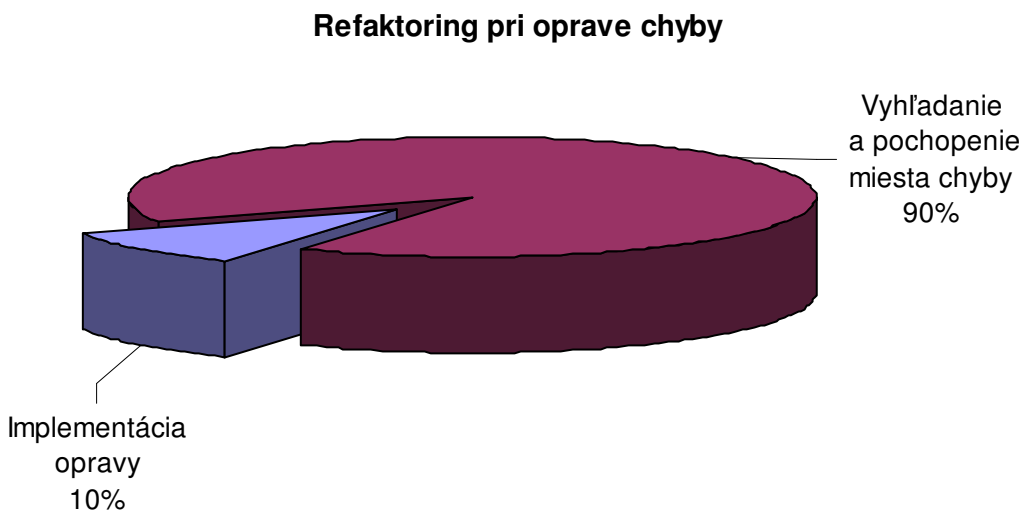
Tretia chyba bola spôsobená spoliehaním sa na nástroj pre refaktorovanie, ktorý ponúka IDE Eclipse. V tomto prípade bolo potrebné refaktoring vykonávať podrobne a poctivo ručne, na základe vyhľadávania výskytov v kóde. Tento problém je napr. podchytený v novej verzii IDE IntelliJ IDEA ver. 7 od firmy JetBrains.

7 Zhodnotenie výsledkov

Nasadením refaktoringu na systém OKbase sa zvýšila kvalita štruktúry jeho kódu. Pomocou refaktorovaní pri oprave chyby a kontroly kódu sa zvýšila aj kvalitatívna stránka produktu z pohľadu zamedzenia výskytov chýb v systéme.

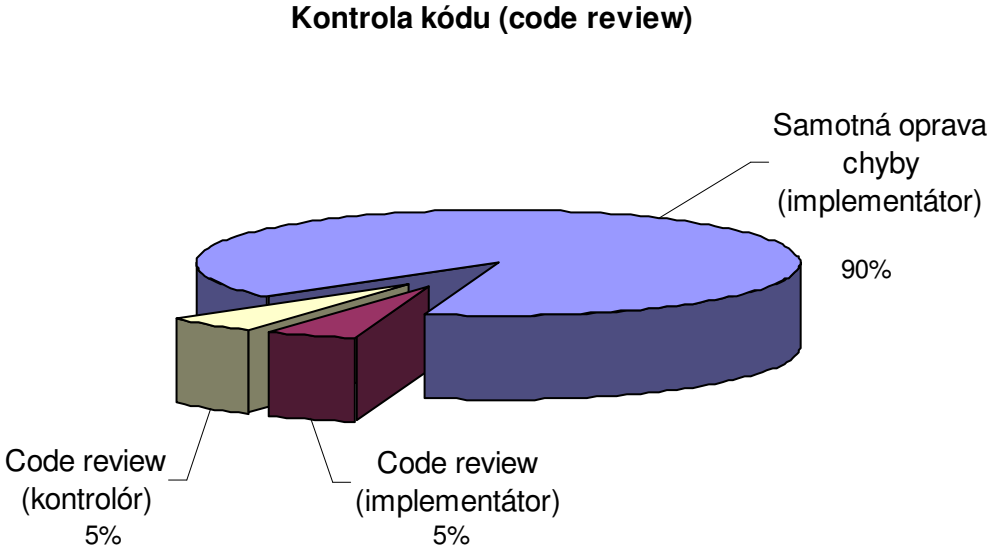
Dôležitou časťou pri oprave ohlásenej chyby v systéme je jej nájdenie a následné pochopenie miesta kódu súvisiaceho s chybou. Až potom nasleduje oprava chyby. Na Obr. 7-1 je znázornené percentuálne rozloženie času opravy chyby. Je zložité uviesť presné hodnoty, pretože časť chýb potrebuje pre opravu menej času, ako vyhľadanie chyby a časť naopak. Napr. v [11, str. 548] sa uvádza, že nájdenie a pochopenie chyby tvorí obvykle 90% celej práce. Preto predpokladajme, že prvý úkon trvá 90% času opravy chyby a samotná oprava 10%. Po zaradení refaktoringu kódu medzi úkony spojené s pochopením chyby sa čas vyhľadania a pochopenia chyby vo väčšine prípadov nezvýši. Väčšia šanca je, že programátor pochopí kód rýchlejšie a lepšie. Popri tom vytvorí sadu testov, ktorá sa využíva aj naďalej.

Preto je dobré zaradenie refaktorovania pri oprave chyby do vývoja softwarového produktu, z hľadiska kvality vyvíjaného produktu.



Obr. 7-1: Graf času stráveného refaktorovaním pri oprave chyby.

Kontrola kódu iným programátorom nie je veľmi známa a používaná. Pritom samotná kontrola spojená s refaktorovaním upravovaného kódu nie je až tak časovo náročná (viď Obr. 7-2) a má viacero kladov, ako bolo spomenuté vo vyhodnotení. Jedná sa hlavne o súčinnosť 2 programátorov a často sa zabráni vloženiu chyby do kódu. Predpokladaná doba trvania kontroly kódu je od 5% do 15% času stráveného na samotnej oprave chyby, alebo pridania novej funkcionality. Na Obr. 7-2 je znázornené celkové časové trvanie opravy spoločne s „*code review*“.



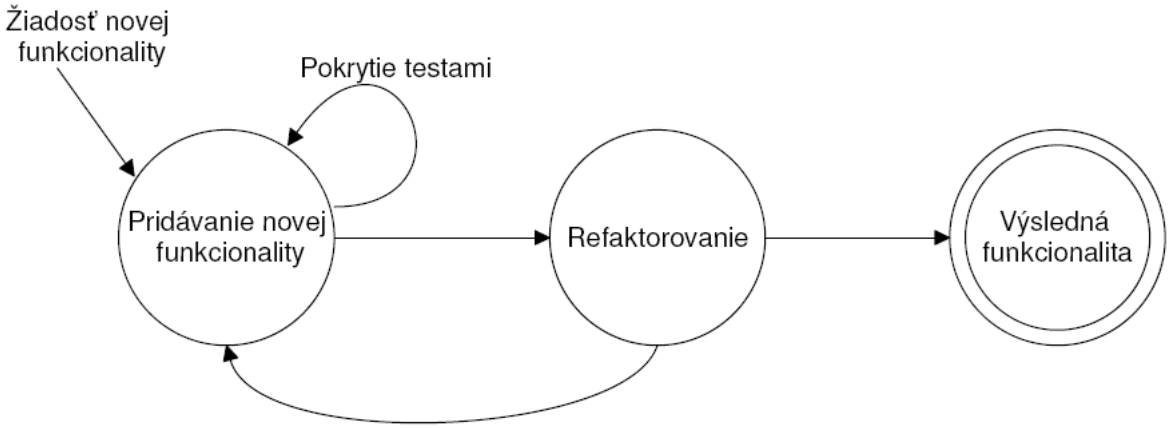
Obr. 7-2: Graf času stráveného refaktorovaním pri "*code review*".

Celkový refaktoring je dôležitý, ak sa v programe nachádza veľa chýb, alebo sa plánuje hromadnejšie rozširovanie systému o novú funkčnosť, ako to bolo v prípade systému OKbase firmy OKsystem s.r.o. V prípade početných chýb môže nastať prípad, keď sa už refaktorovanie danej časti nehodí a je potrebné prepísať ju nanovo. Vtedy je dôležité takýto problém odhaliť včas a následne presadiť v manažmente.

7.1 Súčinnosť programovania a refaktorovania

Programátor by mal pri vývoji software pristupovať k implementácii pomocou dvoch odlišných činností: programovanie, t.j. pridávanie novej funkcionality a refaktorovanie. Pri pridávaní novej funkcionality je možné začleniť tvorbu testov popri vývoji. Existujúci kód by sa nemal meniť, ale rozrastať o novú funkcionalitu. Ak po pridaní novej funkcionality programátor zistí, že napísaný kód je možné prepísať do lepšie štruktúrovanej formy, je čas refaktorovania. Pri tomto refaktorovaní je dôležité, že sa nepridáva žiadna funkcionalita, ale nevytvárajú sa ani testy. Jedine ak nejaký chýba, pretože tvorba testov je zahrnutá v prvej činnosti.

Súvis činností programovania a refaktorovania znázorňuje Obr. 7-3. Programátor po získaní žiadosti o implementáciu novej funkcie začína programovať. Počas jej implementácie pokrýva kód testami, aby sa ubezpečil o jej bezchybnosti. Ak zistí, že potrebuje pridať funkciu, pričom by mu pomohlo štruktúru kódu pozmeniť, začína refaktorovať. Po refaktorovaní pokračuje s pridávaním novej funkcionality. Po ukončení implementácie ešte upraví štruktúru kódu do čitateľnej podoby. Požadovaná funkcionalita je hotová v normálnom čase a bez zbytočného dohadovania s manažmentom, či refaktorovanie použiť alebo nie.



Obr. 7-3: Diagram činností programátora

Tento prístup k programovaniu s využitím refaktorovania sa spomína v [1, str. 71] pod pojmom dva klobúky. Prvý klobúk je pridávanie novej funkcionality a druhý refaktorovanie. Programátor tieto dva klobúky (činnosti) strieda podľa potreby. Využíva ich automaticky, ale stále musí rozlišovať, ktorú činnosť práve vykonáva.

Programátor, ktorý vytvára kvalitný a bezpečný kód týmto spôsobom má v dnešnej dobe veľké možnosti uplatnenia pri tvorbe objektovo orientovaných aplikácií.

7.2 Praktické využitie pri tvorbe OO aplikácií

Celkový prínos refaktoringu je v dnešnej dobe počas vývoja softwarového produktu veľký, pretože je potrebné vytvoriť testy, ktoré zachycujú funkčnosť pred refaktoringom. Tieto testy sa využívajú aj po refaktorovaní. Zlepšuje čitateľnosť kódu a programátorom dáva možnosť lepšieho pochopenia systému z vnútorného pohľadu.

Prínos pre zákazníkov používajúcich takýto produkt je tiež neoceniteľný. Užívatelia síce nevidia kód programu, ale pri používaní sa vyskytuje menej chýb, ktoré je potrebné opravovať.

Z pohľadu refaktorovania objektovo orientovaných aplikácií si myslím, že už v blízkej budúcnosti budú firmy, ktoré nebudú možnosti refaktorovania využívať, pozadu za aktuálnym trendom vývoja kvalitných produktov. Techník a nástrojov, ktoré môžeme v dnešnej dobe používať pre podporu refaktoringu je veľké množstvo, a preto aj doba strávená pri refaktorovaní sa vo veľkej miere skrátila oproti minulým rokom.

Nie je potrebné obávať sa chýb spojených s refaktorizáciou, pretože vývoj je podmienený správou zdrojových kódov pomocou verzovacích systémov (CVS, SVN a iné). Pri ich použití sa zmeny kódu nestrácajú a vždy je možnosť ich vrátenia.

8 Záver

Úlohou tejto práce bolo zoznámenie sa s problematikou refaktorovania softwarových projektov a jej aplikáciou na produkt nasadený v praxi. V práci najprv popisujem podstatu refaktorovania. Rozšíril som si znalosti ohľadom písania čitateľných programov a spôsoby hľadania nečistôt v kóde. Tie sú podstatné pri procese refaktorovania programových aplikácií.

Pretože bol refaktoring vykonávaný nad systémom OKbase, ktorý je nasadený v praxi, bolo potrebné zachytiť jeho funkčnosť pomocou testov. V práci je popísaný spôsob testovania aplikácií prevzatý z vývoja riadeného testami, ktorý je podľa môjho názoru možné aplikovať aj na tvorbu testov pri refaktorizácii. Rozdiel je v tom, že testy sa vytvárajú na už napísaný kód.

Aplikovaniu poznatkov získaných podrobným preštudovaním procesu refaktoringu predchádzala dôsledná analýza systému OKbase a technológií, ktoré sa pri jeho vývoji používajú. Základom samotnej aplikácie bolo vytvorenie sady testov, pri ktorých sa nachádzali nedostatky v kóde, zapisovali a následne refaktorovali. Pred každým refaktoringom som tieto testy rozširoval a dopĺňal o novú refaktorovanú funkcionalitu.

Po aplikácii refaktoringu nad systémom OKbase som pristúpil k diskusii prínosu jednotlivých refaktorovaní. Výsledky som vyhodnotil pozitívne, pretože refaktoring nad systémom OKbase zachytil a určil vnútorné nedostatky a načrtol ich opravu. Zhodnotil som chyby, ktoré vznikli v procese refaktoringu a popísal ich možné riešenia. Vytvorené testy sa už používajú aj v ostrom vývoji tohto systému.

Na základe výsledkov refaktorizácie hodnotím refaktoring ako veľmi silný nástroj pri tvorbe objektovo orientovaných aplikácií. V dnešnej dobe je refaktoring v procese vývoja software dôležitou súčasťou vývoja a udáva nový trend kvality produktu.

Aplikovanie kvalitného refaktoringu je podmienený dobrými programátorskými znalosťami ako napr. znalosť návrhových vzorov. Základom je podrobné pochopenie tejto problematiky a dobré programátorské návyky. Z hľadiska ďalšieho štúdia techník v procese vývoja software ma zaujali praktiky agilného prístupu.

Literatúra

[1] Fowler M. a kol.: Refaktoring - Zlepšení existujícího kódu, Praha: Grada Publishing, prvé vydanie, © 2003, preklad: Vladimír Lahoda, ISBN 80-247-0299-1.

[2] Fowler Martin: Public versus Published Interfaces, IEEE SOFTWARE Marec/Apríl 2002, URL <http://martinfowler.com/ieeeSoftware/published.pdf>.

[3] Beck Kent: Programování řízené testy, Praha, Grada Publishing 2003, ISBN 8024709015.

[4] java-source.net: Open Source Testing Tools in Java, [online], [cit. 27.12.2007], URL <http://java-source.net/open-source/testing-tools>.

[5] Pichlík Roman: Do pranice: Integrační vs. Unit testy, [online], 4.14.2005, [cit. 27.12.2007], URL <http://www.sweb.cz/pichlik/archive/2005_04_10_archive.html>.

[6] OpenQA: Selenium, [online], 2006, [cit. 28.12.2007], URL <http://www.openqa.org/selenium>.

[7] PMD: PMD, [online], 2002-2008, [cit. 20.2.2008], URL <http://pmd.sourceforge.net/>.

[8] Burn Oliver: Checkstyle 4.4, [online], 2001-2007, [cit. 20.2.2008], URL < http://checkstyle.sourceforge.net/>.

[9] Doline Mark: What is Cobertura?, [online], 2005-2006, [cit. 1.3.2008], URL <http://cobertura.sourceforge.net/>.

[10] Oksystem s.r.o.: Vícevrstvá architektura VVAF (VVA framework), interní materiál, [cit. 10.4.2008].

[11] McConnel Steve: Dokonalý kód – Umění programování a techniky tvorby software, Brno: Computer Press, a.s., prvé vydanie, ©2005, preklad: Bohdan Kostka, ISBN 80-251-0849-X.

Zoznam obrázkov

Zoznam príloh

A Test zachycujúci funkčnosť z Príkladu 5-4

```
public class HVztahManagerTest extends SessionTestCase {

  private HVztahManager vztahManager;
  private CiselnikManager ciselnikManager;

  public void setVztahManager(HVztahManager vztahManager) {
    this.vztahManager = vztahManager;
  }

  public void setCiselnikManager(CiselnikManager ciselnikManager) {
    this.ciselnikManager = ciselnikManager;
  }

  @Override
  protected IDataSet getDataSet() throws DataSetException {
    List<String> dataSetNames = new ArrayList<String>();
    dataSetNames.add(XmlDataSetNames.HOSOBA.getLocation());

    return getCompositDataSet(dataSetNames, true);
  }

  public void testZkontrolujZmenuPlatnosti() {
    ObcCalendar cal = new ObcCalendar();
    assertTrue(vztahManager.zkontrolujZmenuPlatnosti(10001L,
            cal.getDateDne(2006, 5, 5)));
    assertTrue(vztahManager.zkontrolujZmenuPlatnosti(10001L,
            cal.getDateDne(2007, 1, 5)));
    assertFalse(vztahManager.zkontrolujZmenuPlatnosti(10001L,
            cal.getDateDne(2007, 3, 5)));
    assertFalse(vztahManager.zkontrolujZmenuPlatnosti(10001L,
            cal.getDateDne(2007, 4, 5)));
    assertFalse(vztahManager.zkontrolujZmenuPlatnosti(10001L,
            cal.getDateDne(2011, 5, 5)));

    try {
      vztahManager.zkontrolujZmenuPlatnosti(null, null);
      fail();
    } catch (IllegalArgumentException e) {
      assertTrue(true);
    }

  }

}
```

Príklad A: Ukážka časti integračného testu zachycujúca funkčnosť refaktorovanej metódy.

B Adresárová štruktúra priloženého CD-ROM

./	- README
./doc	- text diplomovej práce
./src/java	- ukážky zdrojových kódov refaktorizácie
./src/test	- ukážky zdrojových kódov testov