VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

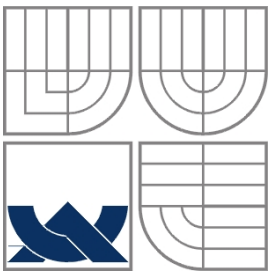
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REFAKTORING OBJEKTOVĚ ORIENTOVANÉ APLIKACE

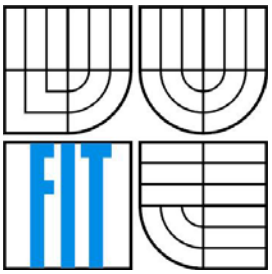
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Martin Solárik
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REFAKTORING OBJEKTOVĚ ORIENTOVANÉ APLIKACE

REFACTORING OF OBJECT ORIENTED APPLICATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Martin Solárik

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. Jitka Kreslíková, CSc.

BRNO 2008

Abstrakt

Tento dokument je magisterskou diplomovou prácou na tému refaktoring objektovo orientovanej aplikácie. Jej cieľom je zoznámiť sa s problematikou refaktoringu, jeho základnými princípmi, výhodami, nevýhodami a základnými používanými vzormi a získané znalosti aplikovať na reálnu aplikáciu. Dokument je rozdelený do niekoľkých častí. V úvode sa venujem definovaniu pojmu refaktoring, v ďalšej jeho základným princípom, výhodám, nevýhodám ako aj dôvodom prečo refaktoring používať. Tretia časť je venovaná vzorom refaktoringu, tvorí akýsi katalóg jednotlivých refaktoringov. Ďalšia časť je venovaná platforme .NET a nástrojom na podporu refaktoringu v tejto platforme. Predposledná kapitola sa venuje predstavením aplikácie a následnej aplikácii refaktoringu. Posledná časť nazvaná záver, je venovaná zhodnoteniu dosiahnutých výsledkov.

Kľúčové slová

Refaktoring, objektovo orientované programovanie, návrh software, softwarové inžinierstvo

Abstract

This document is the master's thesis called refactoring of object oriented application. Goal of this document was to introduce the problem of refactoring and apply gained knowledge on real software project. Document is divided into six chapters. The first chapter is introduction, the term refactoring is defined there with a brief history. Next chapter explains basic principles of refactoring, advantages and disadvantages of using refactoring. The third chapter is called catalog of refactorings and describes common patterns of refactoring. Next two chapters describe .NET platform and tools, which support refactoring on this platform. Sixth chapter is about real application of refactoring. Final chapter is conclusion and summary.

Keywords

Refactoring, object oriented programming, software design, software engineering

Citácia

Solárik Martin: Refaktoring objektovo orientovanej aplikácie. Brno, 2008, diplomová práce, FIT VUT v Brně.

Refaktoring objektově orientované aplikace

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením RNDr. Jitky Kreslíkovej, CSc.

Další informace mi poskytl Ing. Stanislav Mikulecký.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Martin Solárik
15.5.2008

Poďakovanie

Chcel by som sa poďakovať pani RNDr. Jitke Kreslíkovej, CSc. ako aj pánovi Ing. Stanislavovi Mikuleckému za ich odbornú pomoc pri písaní tejto práce. Vždy mi rýchlo poskytli informácie a rady keď som to potreboval.

Obsah

1 Úvod

Tento dokument je magisterskou diplomovou prácou na tému *Refaktoring objektovo orientovanej aplikácie*. Nadväzuje na semestrálny projekt, ktorý bol vypracovaný v zimnom semestri a slúžil ako úvod do problematiky.

Dokument je organizovaný do niekoľkých kapitol. Prvá kapitola slúži ako úvod do problematiky, je v nej definovaný pojem refaktoring a stručná história. Druhá kapitola nazvaná *Princípy refaktoringu* sa venuje refaktoringu ako procesu v teoretickej rovine, popisuje všeobecne princípy a dôvody refaktoringu, jeho výhody a nevýhody. Zaoberá sa otázkami ako sú prečo refaktorovať, kedy refaktorovať a kedy nerefaktorovať. Ďalšia, tretia kapitola popisuje základné vzory refaktoringu, pre každý z nich je tu uvedená motivácia, ilustračný príklad a postup na vykonanie. Refaktoringy sú zoskupené do troch logických skupín pre jednoduchšiu organizáciu. Štvrtá kapitola krátko pojednáva o platforme .NET, jej histórii a kľúčových vlastnostiach. Na túto kapitolu nadväzuje ďalšia kapitola, ktorá sa zaoberá nástrojmi na podporu refaktoringu. Zaoberá sa požiadavkami na takýto nástroj a zároveň popisuje niektoré dostupné a používané nástroje práve v prostredí platformy .NET. Predposledná kapitola je venovaná praktickým ukážkam refaktoringu na reálnej aplikácii. V úvode kapitoly je aplikácia predstavená a určitým spôsobom odmeraný jej stav pred aplikovaním refaktoringu. Počas kapitoly opisujem niekoľko krokov zlepšovania návrhu aplikácie práve s použitím refaktoringu. V závere kapitoly dosiahnuté výsledky zhodnotím a porovnám stav aplikácie po aplikovaní refaktoringu. Poslednou kapitolou je záver, v ktorej zhodnotím to ako som splnil zadanie diplomovej práce.

Úlohou semestrálneho projektu bolo uviesť problematiku, výhody, nevýhody refaktoringu a popísať jeho základne používané vzory. Jeho obsahom sú prvé tri kapitoly, z ktorých tretia bola v rámci diplomovej práce rozšírená o ďalšiu skupinu vzorov. Nasleduje časť riešená v rámci diplomovej práce, ktorej úlohou je preskúmanie nástrojov dostupných pre platformu .NET, praktické využitie refaktoringu na reálnej aplikácii a zhodnotenie dosiahnutých výsledkov.

1.1 Refaktoring

Termín refaktoring má dve definície, ktoré závisia na kontexte, v ktorom sa tento termín používa. Prvá definícia považuje refaktoring za podstatné meno, druhá za sloveso.

Refaktoring (podstatné meno): *zmena vo vnútornej štruktúre software, tak aby bol zrozumiteľnejší, jednoduchší na pochopenie a aby ho bolo jednoduchšie a lacnejšie modifikovať. To všetko bez zmeny vonkajšieho správania sa a funkcionality menej časti software.*[1]

Refaktoring (sloveso): *proces zmeny štruktúry software aplikovaním jednotlivých refaktoringov (viď. vyššie) tak, aby nebolo zmenené vonkajšie správanie a funkcionalita software.*[1]

Pod pojmom refaktoring môžeme teda chápať samostatnú zmenu na kóde vykonanú podľa určitého vzoru za účelom zlepšiť jeho kvalitu, ako aj sériu takýchto zmien vykonaných na celom softwarovom produkte alebo jeho časti.

1.2 História

Refaktoring ako taký nevznikol v nejakú presnú dobu, dobrí programátori vždy trávili nejaký čas aby vyčistili svoj kód. Už dávno prišli na to, že kód, ktorý je ľahký na porozumenie je aj ľahšie udržiavateľný a modifikovateľný. Zároveň bolo jasné, že na prvý krát sa skoro nedá napísať dobrý kód, ktorý sa už nedalo vylepšiť. Tento proces bol však len v povedomí dobrých a skúsených programátorov, nemal žiadnu definíciu, pomenovanie a už vôbec nie vzory, podľa ktorých by sa dalo postupovať.

Jedni z prvých ľudí, ktorí začali chápať dôležitosť refaktoringu boli v osemdesiatych rokoch Ward Cunningham a Kent Beck, ktorí v tom čase pracovali v prostredí Smalltalku. Toto prostredie umožňuje programátorom vo veľmi krátkom čase vyvíjať vysoko funkčný a použiteľný kód a zároveň je objektovo orientované. Keďže títo dvaja páni boli v Smalltalk komunite veľmi vážení, refaktoring sa práve tu začal používať a stal sa z neho veľmi dôležitý nástroj pri vývoji software. Ďalším váženým človekom v Smalltalk komunite je Ralph Johnson, profesor na University of Illionios. Jeho študent, William Opdyke uvidel potenciál a použiteľnosť refaktoringu aj v iných prostrediach ako Smalltalk, konkrétne pre programovanie v jazyku C/C++. V roku 1990 napísal prácu o refaktoringu, ktorá je dodnes jednou z najobsiahlejších na túto tému. Odvtedy môžeme teda datovať pojem refaktoring, aj keď ako proces existoval už dávno predtým.

[1] Fowler, M.: Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.

2 Princípy refaktoringu

Čo to teda refaktoring je? Je to proces, pri ktorom sa čistí už existujúci kód? V podstate áno, ale refaktoring ide ďaleko za hranicu čistenia kódu. Je to proces pomocou ktorého môžeme zo zle navrhnutého kódu postupne dostať dobre navrhnutý, ľahko modifikovateľný, zrozumiteľný a robustný kód. V nasledujúcich odstavoch a podkapitolách sa budem venovať podrobnejšie všeobecným princípom a dôvodom refaktoringu ako aj jeho výhodám a nevýhodám.

Keď sa používa refaktoring pri vytváraní software je nutné si čas rozdeliť na 2 oddelené aktivity a to, pridávanie funkcionalít systému a refaktoring. Keď sa pridáva nová funkcionalita nemal by sa totiž modifikovať už existujúci kód, pridávajú sa schopnosti systému, nemenia sa. Pri pridávaní nových funkcií býva dobrým zvykom zároveň vytvárať aj funkčné testy a tým novú funkčnosť do určitej miery overiť. Pri refaktoringu je situácia iná, cieľom refaktoringu nie je pridať nové funkcie a nemeniť vonkajšie správanie a rozhranie systému. Preto ani nevznikajú nové testy (ak sa nepríde na to, že niektorý dôležitý test chýba). Po aplikácii refaktoringu je možné spustiť existujúce testy na overenie toho, že sme refaktoringom do systému nezaniesli nejakú chybu. Toto je možné práve preto, že sa refaktoringom nezmenilo vonkajšie správanie a rozhranie systému.

Pri vytváraní software je veľmi časté medzi týmito dvoma aktivitami prepínať. Pri pridávaní nových funkcií systému môžeme zistiť, že by bolo jednoduchšie danú funkcionalitu implementovať keby bol systém alebo časť systému navrhnutá inak. Vtedy je čas na refaktoring, novú funkcionalitu zatiaľ necháme tak a pustíme sa do zmeny systému tak, aby bolo novú funkcionalitu možné čo najjednoduchšie implementovať, stále musíme mať však na pamäti to, že nesmieme zmeniť vonkajšie správanie sa systému. Až potom môžeme implementovať novú funkcionalitu. Celý tento proces môže trvať len pár minút, ale počas toho je nutné vždy pamätať na to, ktorej aktivite sa práve venujeme.

2.1 Prečo refaktorovať

Aké sú vlastne dôvody refaktoringu? Refaktoring je v podstate extra práca, ktorá na prvý pohľad nemá na výsledný softwarový produkt žiadny vplyv, jeho cieľom nie je rozširovanie alebo skvalitňovanie funkcií systému. Môže sa zdať, že refaktoringom dosiahneme „iba“ čistejší kód a lepší design systému, za čo zaplatíme veľkou časťou z veľmi drahého času, ktorého často krát nie je dostatok ani na implementáciu požadovanej funkcionality. Toto je však nesprávny a veľmi krátkozraký pohľad na problematiku. Refaktoring je veľmi mocný nástroj, ktorý by sa mal používať na dosiahnutie určitých cieľov, nie to je žiadny zázračný nástroj alebo liek, ktorý premení aj neúspešný softwarový projekt na úspešný. Tomu čo refaktoringom môžeme dosiahnuť sa budem venovať v nasledujúcich odstavcoch.

2.1.1 Refaktoring zlepšuje design software

Bez používania refaktoringu pri vytváraní software, design softwarového produktu časom postupne upadá. Ako programátori do systému pridávajú nové a nové funkcie, prípadne upravujú súčasnú funkcionalitu, tak aby uspokojili krátkodobé ciele, väčšinou nedbajú na návrh systému a zmeny nerobia tak aby tento návrh rešpektovali. V praxi totiž býva väčšinou málo času a dôležitý je predovšetkým výsledok. Preto postupne takto tvorený kód stráca svoju štruktúru a je čím ďalej menej čitateľný a tým pádom aj ťažšie udržiavateľný. Často si programátori neuvedomujú aké následky môže takýto spôsob práce mať. Práve preto je tu refaktoring, pomocou neho je možné kód vyčistiť, presunúť časti tam kde by mali naozaj patriť tak, aby bol dodržaný pôvodný návrh systému alebo upraviť návrh tak, aby bol kód štruktúrovaný, čitateľný a udržiavateľný.

Veľmi častým znakom nesprávaného návrhu alebo dôsledok vyššie popísaného spôsobu písania kódu je duplicitný kód. To znamená, že na viacerých miestach systému je časť kódu, ktorá robí v podstate to isté. Je to totiž najrýchlejší spôsob ako rozšíriť systém o novú funkciu. Nájde sa kód, ktorý robí skoro to čo programátor potrebuje, skopíruje sa, trochu upraví a tým sa dosiahne požadovaná funkcionalita bez nutnosti zasiahnuť do už existujúceho kódu. Čím viac kódu však systém obsahuje, tým je ťažší na porozumenie a tým je zložitejšie ho v budúcnosti zmeniť. Môže sa stať, že pri zmene systém nebude pracovať tak ako by sme očakávali, to väčšinou preto lebo sme zmenili kód na len na jednom mieste a nie na všetkých miestach kam bol pri predchádzajúcich zmenách skopírovaný. Tu je vhodné použiť refaktoring, ktorého jedným z cieľov je odstrániť duplicitný kód a tým zlepšiť design systému. Tým, že sa duplicitný kód odstráni sa správanie systému nijak nezmení, nespôsobí to ani to, že bude systém pracovať rýchlejšie. Táto zmena však umožní jednoduchšiu možnosť úpravy kódu v budúcnosti, všetka funkcionalita je implementovaná práve raz a je umiestnená tam kde logicky patrí, čo je jedným zo základných pilierov dobrého designu.

2.1.2 Refaktoring zvyšuje zrozumiteľnosť kódu

Pri programovaní v prvom rade myslíme na to ako vyriešiť daný problém, píšeme kód tak aby bol funkčný. Často hlavne kvôli nedostatku času a pod tlakom termínov dopredu nepremyslíme a nenavrhneme design novej funkcionality. A prečo aj, veď počítaču (kompilátoru) je jedno ako je kód napísaný, vykoná presne to, čo mu prikážeme a je mu jedno, že je niečo napísané nelogicky, nečitateľne, neprehľadne, že kód obsahuje duplicity, atď. Okrem toho, aby bol samotný kód funkčný by sme mali dbať ale aj na to, aby bol zrozumiteľný a čo najjednoduchšie pochopiteľný. Musíme totiž myslieť aj na druhých programátorov, ktorí budú po nás kód čítať, prípadne upravovať, ale aj na to, že sa k nemu niekedy v budúcnosti vrátime práve my a budeme ho potrebovať opraviť alebo modifikovať, to môže byť aj po niekoľkých mesiacoch. To je pravdepodobne to najdôležitejšie na čo by sme mali pri písaní kódu myslieť. Nikoho totiž príliš nezaujíma, že je v nejakom algoritme viac

cyklov alebo tried ako by bolo nezbytne nutné alebo že kompilácia trvá o niečo dlhšie. Čo nás ale zaujímať bude je to, prečo vykonať jednoduchú zmenu a upraviť kód miesto pár hodín trvá niekoľko dní, až týždňov. Je to preto, lebo programátor, ktorý ma danú zmenu implementovať strávi väčšinu času tým, že kód študuje a zisťuje ako vlastne funguje, až potom je schopný nájsť príslušnú časť kódu a implementovať jednoduchú zmenu, čo mu naozaj zaberie už len tých pár hodín. Čomu sa ale chceme vyhnúť je práve ta dlhá doba štúdia snaženia sa pochopiť kód. V tom nám zase môže pomôcť refaktoring.

Dobrým zvykom je aplikovať refaktoring aj pri čítaní alebo upravovaní cudzieho kódu ak je napísaný príliš zložito, nejasne alebo nezrozumiteľne. Nielenže to pomôže ďalším programátorom v budúcnosti, pomôže to aj nám aby sme danému kódu lepšie porozumeli a aby sme mohli prípadnú zmenu implementovať jednoduchšie. Pri tom je samozrejme dobré spustiť príslušné testy, ktoré overia, či zmena spôsobená refaktoringom nezaniesla do systému chybu.

2.1.3 Refaktoring pomáha pri hľadaní chýb

Zrozumiteľný a čitateľný kód, o ktorom sa píše v predchádzajúcej kapitole nám pomôže jednoduchšie identifikovať prípadne chyby, čo je veľmi veľká výhoda a jednoznačný argument prečo refaktorovať. Ak totiž narazíme na pre nás nezrozumiteľný kód, aplikujeme refaktoring a zlepšime jeho čitateľnosť, tým viac tomuto kódu porozumieme a môžeme objaviť chyby, ktoré boli doteraz zamaskované v neprehľadných konštrukciách a v kusoch kódu, ktoré „nejako fungujú“, ale nik presne nerozumie ako a prečo.

2.1.4 Refaktoring zvyšuje rýchlosť práce

Refaktoring zvyšuje rýchlosť a efektivitu práce. To znie na prvý pohľad trochu zvláštne. To, že používaním refaktoringu ako komplexného procesu môžeme dosiahnuť dobrý design systému, zlepšiť už existujúci design, dosiahnuť čitateľnejší a zrozumiteľnejší kód alebo ľahšie odhaliť skryté chyby, je jednoznačne preukázateľný fakt. Ale to, že by refaktoring mohol zvýšiť rýchlosť práce tak jednoznačné nie je.[2]

Tým, že sa pri programovaní venujeme ešte aj upravovaním už hotového kódu, ktorý navyše funguje, si pridávame extra prácu, ktorá na požadovaný výsledok nemá viditeľný vplyv. Takže ako nám refaktoring urýchli prácu? Bez dobrého návrhu sa na začiatku vývoja softwaru pracuje veľmi rýchlo, to je však len dočasné, postupne ako je projekt zložitejší a zložitejší je efektivita práce nižšia a nižšia. Ladenie chýb v nezrozumiteľnom a ťažko čitateľnom kóde nám začne zaberať viac času ako samotné pridávanie nových funkcií, úpravy budú trvať dlhšie ako by sme čakali, lebo bude nutné upraviť aj zduplikovaný kód. Na udržanie rýchlosti a efektivity pri vývoji software je veľmi dôležitý

[2] Fowler, M.: Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.

dobrý design a zrozumiteľný kód, k čomu nám jednoznačne refaktoring slúži, tým pádom refaktoring v konečnom dôsledku aj zvyšuje rýchlosť a efektivitu práce.

2.2 Kedy refaktorovať

Keď už vieme prečo refaktorovať a aké to prináša výhody, je ďalším logickým krokom povedať si, kedy refaktorovať. Refaktoring nie je činnosť, ktorá by sa mala vykonávať oddelene od vývoja. Nie je to činnosť, na ktorú by sme si mali vyhradiť špeciálny čas a nerobiť nič iné len refaktorovať. V praxi takýto prístup ani nie je možný, nik nedá programátorom mesiac na to aby nič nevytvorili, len aby čistili už existujúci a funkčný kód. V nasledujúcich odstavoch sa budem venovať situáciám kedy je vhodné venovať sa refaktoringu.

2.2.1 Pridanie novej funkcionality

Najbežnejšia situácia, kedy použiť refaktoring je pri pridávaní novej funkcionality do systému. Jedným z dôvodov môže byť, keď sa dostaneme ku kódu, ktorému nerozumieme alebo je napísaný neprehľadne, vtedy je vhodné použiť refaktoring, upraviť kód tak, aby bol zrozumiteľný či už pre nás alebo pre ďalších programátorov, ktorí s týmto kódom prídu do styku. Tým pádom kódu lepšie porozumieme a bude pre nás jednoduchšie novú funkcionalitu zapracovať. Ďalším dôvodom na refaktoring môže byť situácia, keď zistíme že by bolo jednoduchšie a čistejšie implementovať novú funkcionalitu keby bol systém navrhnutý inak. Vtedy je dobré odložiť pridanie novej funkcionality bokom, aplikovať refaktoring tak, aby bol systém postavený vhodnejšie a až potom pridať novú funkcionalitu. Nielenže sa nám nová funkcia bude pridávať jednoduchšie, ale uľahčíme tým prácu iným alebo sebe, keď budeme v budúcnosti v rovnakej situácii.

2.2.2 Oprava chyby

Ďalšou situáciou, kedy je vhodné použiť refaktoring je pri oprave chyby. Keď ladíme kód a hľadáme chybu musíme danému kódu dobre rozumieť. Za prvé aby sme samotnú chybu našli a za druhé aby sme boli schopný chybu odstrániť a nezaniesli pri tom do systému ďalšiu chybu. Ako som už niekoľkokrát spomenul, ak je kód nezrozumiteľný je potreba aplikovať refaktoring a kód tak vyčistiť a sprehľadniť. Pri oprave chyby je veľmi dôležité kódu rozumieť a dobre sa v ňom orientovať, takže refaktoring v tejto situácii je viac než vhodný.

2.2.3 Revízia kódu

V niektorých firmách bývajú dobrým zvykom pravidelné revízie kódu, čo je veľmi užitočné. Revízie kódu môžu pomôcť menej skúseným programátorom naučiť sa niečo nové od skúsenejších kolegov, spôsobuje to, že sa vedomosti o komplexnom systéme dostanú medzi viacerých ľudí, viac ľudí sa

môže podieľať na riešení jedného problému, tak že keď vidia kód riešenia môže ich napadnúť niečo, čo autora pri riešení nenapadlo, atď. Je to ale aj vhodná situácia na refaktoring.

Autorovi sa jeho kód môže zdať čistý, čitateľný a jednoznačný, ale pre ostatných kolegov až taký jasný byť nemusí a dôležité je aby kódu rozumel nielen jeho autor, ale aj ľudia, ktorí s ním prídu do styku neskôr. K čomu nám pri revíziách kódu môže pomôcť refaktoring. Keď prechádzame cudzím kódom, aplikujeme refaktoring a vidíme, že je kód o poznanie jasnejší, navrhneme to autorovi kódu a tým mu môžeme pomôcť vyjadriť jasnejšie to čo mal na mysli a ostaným kolegom daný kód lepšie a rýchlejšie pochopiť.

2.3 Kedy nerefaktorovať

Existujú situácie kedy nie je refaktoring vôbec vhodný. Jedným z príkladov, kedy sa aplikovať refaktoring neoplatí je, keď je rýchlejšie, lacnejšie a bezbolestnejšie daný kód alebo funkcionalitu prepísať celú znova. Na to ako takýto kód vyzerá asi neexistuje žiadny presný popis alebo postup ako ho odhaliť. Jedným zo znakov ale môže byť to, že je tento kód nefunkčný. To môžeme zistiť jedine tak, že spustíme príslušné testy a zistíme, že je daný kód plný chýb a je skoro nemožné ho v rozumnom čase opraviť. Je nutné si pamätať, že kód, na ktorý sa oplatí aplikovať refaktoring musí byť v prvom rade funkčný.

Ďalším dôvodom alebo situáciou kedy lepšie vyhnúť sa refaktoringu je čas, keď sa projekt blíži k dôležitému termínu. Je to hlavne preto, lebo zisk z tohto refaktoringu sa prejaví až neskôr po tomto termíne, čiže neskoro. Odkladať však refaktoring kvôli nedostatku času je ale chyba, je preukázateľné, že refaktoringom efektivita a produktivita rastie a tým pádom sa aj šetrí čas. Nedostatok času je často naopak znakom toho, že by sme mali refaktorovať.

2.4 Problémy pri refaktoringu

Pri procese, ktorý nám pomáha zlepšiť návrh aplikácií, pomáha pri hľadaní chýb, zvyšuje produktivitu práce a má ďalšie výhody, je už ťažšie uvedomiť si, kde sú jeho limity, kde a kedy ho nie je vhodné použiť, dokonca kedy môže byť dokonca kontraproduktívny. V nasledujúcich kapitolách sa budem venovať práve tým situáciám, ktoré sú problematické alebo pri ktorých je použitie refaktoringu nevhodné.

2.4.1 Databáze

Jednou z problémových oblastí pri aplikovaní refaktoringu sú databáze. Väčšina biznis aplikácii je veľmi úzko zviazaná s dátovým model v databáze. To je jeden z dôvodov prečo je veľmi zložité dátový model zmeniť, druhým dôvodom býva migrácia dát z pôvodného modelu do modelu nového.

Ak sa jedná o neobjektové databáze, jedna z možností ako sa s týmto problémom vyrovnať je vytvorenie samostatnej vrstvy, ktorú vložíme medzi databázu a objektový model aplikácie. Týmto spôsobom izolujeme rozdiely v dátových modeloch. Ak nastane zmena v dátovom modeli, jediné čo je nutné zmeniť je medzi vrstva, aplikačnej logiky sa zmena vôbec nedotkne. Tým sa samozrejme aj zvýši zložitosť riešenia. Táto technika sa však nepoužíva len pri aplikovaní refaktoringu, používa sa aj napríklad ak je dátový model veľmi komplexný alebo je aplikácia postavená na databáze, ktorú nemáme úplne pod kontrolou. Nie je nutné s touto prostrednou vrstvou počítať hneď od začiatku, stačí keď ju vytvoríme až keď to bude nezbytne nutné, pri ďalších zmenách nám to ušetrí veľa práce.

2.4.2 Zmena rozhrania

Jednou z charakteristík a výhod objektovo orientovaného programovania je to, že implementácie funkcionality je zapuzdrená do objektov s určitým rozhraním pomocou, ktorého je možné s nimi komunikovať. To nám umožňuje zmenu vnútornej štruktúry systému bez toho aby sme zmenili jeho vonkajšie správanie. To čo je dôležité je práve rozhranie, niekedy sme ale nútení zmeniť aj rozhranie.

Ako sme si definovali v prvej kapitole refaktoring je proces alebo činnosť, pri ktorej meníme vnútornú štruktúru systému bez zmeny jeho vonkajšieho správania. Problémom ale je, že mnoho jednotlivých vzorov refaktoringu je práve o zmene rozhrania (napr. *Rename Method*). Nie je samozrejme problém zmeniť názov metódy, ak máme prístup ku všetkým miestam, kde sa táto metóda používa. Aj keď je metóda verejná, môžeme ju bez ťažkostí premenovať ak môžeme upraviť aj všetky miesta, kde je volaná. Problém nastáva až v situácii kedy je nutné zmeniť rozhranie, ktoré je používané kódom, do ktorého nemáme prístup a nemôžeme ho zmeniť. To môže nastať keď potrebujeme zmeniť verejne publikované rozhranie. Tým môže napríklad byť rozhranie webovej služby alebo rozhranie objektu používaného pomocou remotingu, ktoré je používané externou aplikáciou.

Ak je nutné pri refaktoringu zmeniť takéto rozhrania, musíme zároveň zachovať staré aj nové rozhranie. Aspoň po takú dobu kým sa na zmenu pripravia aplikácie a užívatelia používajúci toto rozhranie. Táto situácia by sa mala riešiť tak, aby metódy starého rozhrania volali metódy nového rozhrania, jednoznačne by sme nemali kopírovať telo metódy a vytvoriť tak duplicitný kód. Ďalšiu vec, ktorú by sme mali spraviť je označiť metódy starého rozhrania, tak aby bolo jasné že ide o zastarané veci a poskytnúť novšiu alternatívu. V .NET na tento účel slúži atribút *Obsolete*.

2.4.3 Refaktoring a výkon

Všeobecná obava pri použití refaktoringu sa týka výkonu software. Tým, že sa snažíme aby bol kód čo najlepšie nahrnutý, štruktúrovaný, čitateľný a zrozumiteľný spôsobíme zároveň to, že bude kód zložitejší, komplexnejší a tým pádom bude bežať pomalšie. To je veľmi zásadná vec a mali by sme jej venovať pozornosť. Jednoznačne by sme nemali uprednostňovať čistotu návrhu a kódu pred

výkonom aplikácie. Čo je ale dôležité si uvedomiť je to, že ak chceme vytvoriť rýchly systém alebo vyladiť pomalý systém tak aby bol rýchly, musíme mať systém ktorý sme ladiť schopný. Čo to znamená? Musíme najprv napísať systém, ktorý budeme schopný ľahko upraviť a vyladiť tak aby spĺňal aj naše požiadavky na výkon, v čom nám refaktoring pomôže. Až potom sa môžeme pustiť do ladenia a zvyšovania výkonu.

Je niekoľko spôsobov ako písať rýchly a výkonný software. Jedným z nich je spôsob, pri ktorom sa pri návrhu systému každému z jeho komponent priradí časový limit. Tento limit nesmie byť prekročený a zároveň si komponenty medzi sebou nemôžu nevyužitý čas „požičiavať“. Tento spôsob dosahovania výkonu a rýchlosti systému sa používa pri systémoch, ktoré požadujú striktne aktuálne dáta, ako napríklad stimulátory srdca.

Ďalším zo spôsobov je ten, pri ktorom každý programátor pri svojej práci dbá na to aby kód, na ktorom pracuje bol čo najrýchlejší a čo najefektívnejší. Tento prístup však v praxi moc nefunguje. Kód písaný takýmto štýlom je neprehľadnejší a je ťažšie pracovať s ním, tým pádom je efektivita vývoja nižšia. Takýto spôsob by mal zmysel, keby takto napísaný systém naozaj bežal rýchlejšie, v praxi to tak ale bohužiaľ nie je.

Zaujímavou vecou o výkone software je to, že vo väčšine softwarových systémoch sú za ich nízky výkon zodpovedné iba malé bloky kódu. Takže ak sa optimalizuje systém ako celok pri jeho vývoji a optimalizuje sa všetok kód rovnako, zistíme že 90% kódu sme optimalizovali zbytočne, pretože väčšina kódu nie je spúšťaná tak často aby to malo za následok znižovanie výkonu celého systému. Takto strávený čas je stratený čas.

Z tejto 90% štatistiky vychádza aj tretí spôsob. Pri tomto spôsobe sa na výkon aplikácie pri jeho vývoji vôbec nehľadí. Dôležitejšie je aby bol systém dobre navrhnutý, kód prehľadný a ľahko modifikovateľný. Optimalizáciou sa budeme zaoberať až v príslušnej fáze životného cyklu projektu. To sa deje nasledovne. Program sa spustí pod profilerom, ktorý program monitoruje. Podľa toho zistíme, ktorá časť systému spotrebuje viac času a prostriedkov ako je potrebné. Týmto spôsobom nájdeme tú malú časť kódu, ktorá spôsobuje nízky výkon a môžeme sa sústrediť na jej optimalizáciu. Takto nespotrebujeme množstvo času na optimalizáciu všetkého kódu ale sústredíme sa naozaj len na ten problémový. Podobne ako pri aplikovaní refaktoringu, tak aj pri optimalizácii sa postupuje po malých krokoch, po ktorých sa spúšťajú testy, aby sa zistilo, či sa zmenami nezaniesli do systému chyby. Ak zistíme, že sa ani po zmenách výkon aplikácie nezmenil, vrátime sa späť a skúsime to inak.

Pri procese optimalizácie nám dobre navrhnutý software, ktorý dosiahneme aj práve vďaka refaktoringu môže pomôcť. V prvom rade preto, lebo nám na optimalizáciu zostane čas, ako sme si ukázali skôr, refaktoring zvyšuje efektivitu práce. V druhom rade je vďaka dobre navrhnutému kódu analýza výkonnosti presnejšia, pretože nás zavedie k jednoznačne umiestneným a väčšinou malým častiam kódu, ktoré nízky výkon spôsobujú a keďže je kód čistý a zrozumiteľný je aj jednoduchšie optimalizovať ho. Čo z toho teda vyplýva? Áno, používaním refaktoringu pri vývoji software a snaha

o čistý návrh a kód preukázateľne znižuje výkon. Čo je ale podstatnejšie, v optimalizačnej fáze nám práve čistý návrh a kód uľahčí prácu a ušetrí čas.

3 Katalóg refaktoringov

V tejto kapitole sa budem venovať jednotlivým refaktoringom v teoretickej rovine, postupne prejdem od základných k zložitejším, vysvetlím kedy a ako daný refaktoring použiť, aké z použitia plynú výhody, prípadne nevýhody. Názvy refaktoringov budú uvedené v anglickom jazyku, tieto názvy sú používané v odbornej literatúre ako aj v nástrojoch určených pre refaktoring. Násilný alebo doslovný preklad by mohol skôr uškodiť ako pomôcť, takže pri každom z nich veľmi stručne vysvetlím, čo konkrétny názov znamená. Táto kapitola bude tiež slúžiť ako zoznam jednotlivých refaktoringov, na ktorý sa budem neskôr odkazovať v praktickej časti práce pri konkrétnych ukážkach aplikácie refaktoringu na reálnej aplikácii. Kapitola je organizovaná do niekoľkých podkapitol, každá z nich obsahuje zoznam refaktoringov, ktoré spolu tematicky súvisia.

3.1 Vytváranie metód

Veľká časť refaktoringu spočíva v tvorení metód, rušení metód, presúvaní kódu medzi metódami, tak aby časti kódu, ktoré spolu logicky súvisia boli na jednom mieste. Typickým problémom sú príliš dlhé metódy. Tie totiž obsahujú príliš mnoho informácií a zložitej logiky, to všetko na jednom mieste, čo je za prvé neprehľadné ale aj ťažko udržiavateľné. Kľúčovým refaktoringom v tejto časti je *Extract method (4.1.1)*, ktorý z určitého bloku kódu vytvorí samostatnú metódu. Presným opakom je *Inline method (4.1.2)*, ktorý miesto volania metódy nahradí jej telom. V nasledujúcich odstavcoch sa jednotlivým refaktoringom budem venovať podrobnejšie.

3.1.1 Vyňatie metódy (*Extract method*)

Ak máme blok kódu, ktorý vykonáva nejakú logicky ucelenú činnosť, môžeme z tohto kódu vytvoriť samostatnú metódu, tento refaktoring sa nazýva *Extract method* (vyňatie metódy).

```
public void PrintInfo()
{
    // Print User Detail
    Console.Out.WriteLine("Name: " + this._user.Name);
    Console.Out.WriteLine("LastName: " + this._user.LastName);
    ....
}
```

```
public void PrintInfo()
{
    PrintUserDetail();
    ....
}

private void PrintUserDetail()
{
    Console.Out.WriteLine("Name: " + this._user.Name);
    Console.Out.WriteLine("LastName: " + this._user.LastName);
}
```

Kód 1. Ukážka *Extract method*

3.1.1.1 Motivácia

Extract method je jeden z najčastejšie používaných refaktoringov. Keď je nejaká metóda príliš dlhá, je vhodnejšie ju rozdeliť na viacero kratších metód práve pomocou *Extract method* (Kód 1.). Ďalším kandidátom môže byť kód, ktorý je okomentovaný aby bolo prípadnému čitateľovi (alebo aj autorovi pri neskoršej revízii) jasné na čo daný kód slúži. Vtedy je vhodnejšie miesto opisovania funkcionality v komentári použiť *Extract method*, vytvoriť z tohto kódu metódu a pomenovať ju tak a aby názov jednoznačne popisoval funkcionalitu. Výhodou je za prvé zjednodušenie pôvodnej metódy a za druhé znížime pravdepodobnosť duplikovaného kódu. Vytvorením novej metódy poskytneme dané preťaženie. Táto technika je v praxi veľmi užitočná, len si treba uvedomiť, že je nutné dobre pomenovať metódy aby bolo hneď z názvu jasné akú funkcionalitu zastupujú. Cieľom tohto refaktoringu nie je ani tak zmenšiť fyzickú dĺžku metódy ako o redukciu množstva logiky, ktorú vykonáva a samozrejme o to aby bol kód metódy jasný a čitateľný. Preto ak je aj názov novej metódy dlhší ako nahradený kód, ale zvýši sa tým čitateľnosť kódu, oplatí sa tento krok vykonať.

3.1.1.2 Postup

i. Vytvoríme novú metódu a jej názov, podľa toho čo daná metóda robí, nie ako to robí.
ii. Skopírujeme extrahovaný kód z pôvodnej do novej metódy.
iii. Zistíme, či kód využíva premenné, ktoré sú lokálne v pôvodnej metóde. Ak áno, budú to lokálne premenné alebo parametre novej metódy.
iv. Premenné, ktoré sú používané aj mimo novej metódy budú jej parametre.
v. Premenné, ktoré sú použité iba v novo vytvorenej metóde budú jej lokálnymi premennými.
vi. Nahradíme extrahovaný kód v pôvodnej metóde volaním novo vytvorenej metódy.
vii. Preložíme kód a spustíme príslušné testy.

3.1.2 Metóda na riadku (*Inline method*)

Inline method (metóda na riadku) je presným opakom *Extract method*, slúži na zrušenie metódy a na nahradenie jej volania jej telom. Používa sa vtedy, keď je telo metódy krátke a jasné ako jej názov.

```
public bool Check()
{
    ...
    bool error = IsValueBelowLimit(_limit);
    ...
}

private bool IsValueBelowLimit(decimal limit)
{
    return this._value < limit;
}
```

```
public bool Check()
{
    ...
    bool error = this._value < _limit;
    ...
}
```

Kód 2. Ukážka *Inline method*

3.1.2.1 Motivácia

Ako sme si už povedali, jedným z dobrých návykov pri programovaní je používať krátke metódy s jasnými názvami, vedie to k prehľadnejšiemu a štruktúrovanejšiemu kódu, ktorý je jednoduchšie čitateľný (Kód 2.). Niekedy však narazíme na metódu, ktorej telo je krátke a jasné ako samotný názov metódy. Ak na takúto metódu narazíme, mali by sme sa jej zbaviť. Ďalším prípadom kedy môžeme *Inline method* použiť je, keď narazíme na skupinu metód, ktoré sú výsledkom nesprávneho refaktoringu, vtedy ich môžeme zoskupiť do jednej metódy a znovu použiť *Extract method* na správne rozdelenie metód. Najčastejšie je však tento refaktoring používaný na odstránenie príliš jednoduchých metód.

3.1.2.2 Postup

i. Skontrolujeme či metóda, ktorú chceme odstrániť nie je implementovaná aj v niektorom z potomkov triedy, ktorá túto metódu definuje.
ii. Nájdeme všetky volania metódy.
iii. Nahradíme ich telom metódy.
iv. Preložíme a spustíme testy.
v. Odstránime metódu.

3.1.3 Dočasná premenná na riadku (*Inline temp*)

Ak máme dočasnú premennú, ktorej je hodnota priradená iba raz a to veľmi jednoduchým výrazom, odstránime všetky použitia tejto premennej a nahradíme výrazom, ktorým jej bola priradená hodnota. Tento refaktoring sa nazýva *Inline temp* (dočasná premenná na riadku).

```
decimal amount = this._order.Amount;
return amount < this._limit;
```

```
return this._order.Amount < this._limit;
```

Kód 3. Ukážka *Inline temp*

3.1.3.1 Motivácia

Keď je hodnota priradená dočasnej premennej iba raz a táto premenná sa nepoužíva na veľa miestach, môže byť prehľadnejšie nahradiť jej použitie výrazom, ktorým jej bola priradená hodnota. Treba si však uvedomiť to, že ak táto premenná drží výsledok nejakej metódy a my všetky jej použitia nahradíme volaním metódy spustíme výpočet tejto hodnoty x krát, čo môže mať výrazný dopad na výkon aplikácie. Použitie tohto refaktoringu môže byť užitočné v jednoduchších prípadoch kde pomôže zvýšiť čitateľnosť kód (Kód 3.).

3.1.3.2 Postup

i. Označíme dočasnú premennú modifikátorom *const*, čím si overíme, že je premennej hodnota priradená naozaj iba raz.
ii. Nájdeme všetky jej použitia a nahradíme ich výrazom, ktorým jej bola prvý krát priradená hodnota.
iii. Pri každej náhrade preložíme kód.
iv. Odstránime deklaráciu a priradenie hodnoty dočasnej premennej.
v. Preložíme a spustíme testy.

3.1.4 Vysvetľujúca premenná (*Explaining variable*)

Explaining variable (vysvetľujúca premenná) je refaktoring, ktorý sa používa keď máme zložitý výraz. Vtedy si tento výraz rozložíme na časti a ich výsledok uložíme do dočasných premenných pomenovaných tak, aby bolo jasné hodnotu akého výrazu zastupujú.

```
if (platform.ToLower().IndexOf(“windows”) > -1 &&
    browser.ToLower().IndexOf(“ie”) > -1 &&
    wasInitialized() && resize > 0)
{
    ...
}
```

```
bool isWindows = platform.ToLower().IndexOf(“windows”) > -1;
bool isIE = browser.ToLower().IndexOf(“ie”) > -1;
bool wasResized = resize > 0;

if (isWindows && isIE && wasInitialized() && wasResized)
{
    ...
}
```

Kód 4. Ukážka *Explaining variable*

3.1.4.1 Motivácia

Výrazy môžu byť niekedy veľmi zložité a tým pádom aj nečitateľné. V takýchto prípadoch nám dočasné premenné môžu pomôcť, pretože ak si zložité výrazy rozložíme na niekoľko kratších, logicky oddelených a výsledok si uložíme do dobre pomenovanej premennej, výrazne sa zvýši čitateľnosť daného kódu a umožní nám to jednoduchšie ladenie a hľadanie prípadných chýb. Tento postup sa najčastejšie používa pri vyhodnocovaní podmienok alebo pri dlhých algoritmoch, pri ktorých si čiastkové výsledky môžeme ukladať do dočasných premenných.

Tento refaktoring je jednoduchý a veľmi používaný. Je však dobré miesto neho zvážiť aké sú možnosti použitia *Extract method*. Tým totiž skrátime metódu a umožníme logiku používať aj z iných miest. Sú však situácie, kedy nie je *Extract method* vhodné použiť, použijeme teda *Explaining variable* (Kód 4.).

3.1.4.2 Postup

i. Deklarujeme si dočasnú premennú a priradíme jej výsledok časti refaktorovaného výrazu alebo algoritmu.
ii. Nahradíme príslušnú časť výrazu alebo algoritmu dočasnou premennou. V prípade, že sa daná časť výrazu (algoritmu) opakuje, nahradíme všetky výskyty dočasnou premennou.
iii. Preložíme a spustíme testy.
iv. Opakujeme tento postup pre ostatné časti výrazu (algoritmu).

3.1.5 Rozdelenie dočasných premenných (*Split temporary variables*)

Ak máme dočasnú premennú, ktorej je hodnota priradená viac než jedenkrát a nie je to v rámci cyklu alebo premenná na zbieranie určitých informácií, vytvoríme dočasnú premennú pre každé jej priradenie.

```
decimal temp = 2 * (this._height + this._width);
Console.Out.WriteLine(“Perimeter: ” + temp.ToString());
temp = this._height * this._width;
Console.Out.WriteLine(“Area: ” + temp.ToString());
```

```
const decimal perimeter = 2 * (this._height + this._width);
Console.Out.WriteLine(“Perimeter: ” + perimeter.ToString());
const decimal area = this._height * this._width;
Console.Out.WriteLine(“Area: ” + area.ToString());
```

Kód 5. Ukážka *Split temporary variables*

3.1.5.1 Motivácia

Dočasné premenné vytvárame v rôznych situáciách. Niektoré z nich prirodzene vedú k tomu, že je danej premennej hodnota priradená viac než raz. Príkladom môže byť napríklad cyklus (`for (int i = 0; i < 10; i++)`) alebo premenné, ktoré zbierajú určitú informáciu.

Často sa dočasné premenné používajú na uchovania výsledku časti algoritmu na neskoršie použitie. Tento typ dočasných premenný by mal byť nastavovaný len raz. Ak sa hodnota takejto premennej nastavuje viac než raz, je to znakom toho, že ma premenná rôznu funkciu v rôznych častiach metódy. Takéto premenné by mali byť nahradené viacerými premennými, pre každé priradenie jednou. Používať jednu dočasnú premennú na viac účelov v jednej metóde je pre prípadného čitateľa veľmi mätúce, preto by sme sa tomu mali vyhnúť (Kód 5.).

3.1.5.2 Postup

i. Zmeníme názov dočasnej premennej pri jej deklarácii ako aj na mieste, kde je jej prvýkrát priradená hodnota. Ak je niektoré z ďalších jej priradení vo forme temp = temp + výraz, je to znak toho, že táto premenná slúži na zber informácii, takúto premennú necháme tak.
ii. Označíme novo vytvorenú premennú modifikátorom const.
iii. Nahradíme všetky použitia pôvodnej premennej použitím novej premennej od deklarácie až po nasledujúce priradenie.
iv. Vytvoríme novú premennú s vhodným názvom pri každom ďalšom priradení pôvodnej premennej.
v. Preložíme a spustíme príslušné testy.

3.1.6 Odstránenie priradenia hodnoty parametrom (*Remove assignments to parameters*)

Remove assignments to parameters (odstránenie priradenia hodnoty parametrom), je refaktoring, ktorý sa používa v situácii keď kód metódy parametrom, ktoré do nej vstupujú priradí novú hodnotu, miesto toho by sme mali použiť dočasnú premennú a ďalej pracovať s ňou.

```
public bool ComputeRates(Row row, decimal? rate, int tipId)
{
       if (rate > 100 || rate < 1)
           rate = null;
       ...
}
```

```
public bool ComputeRates(Row row, decimal? rate, int tipId)
{
       decimal newRate = rate;
       if (newRate > 100 || newRate < 1)
           newRate = null;
       ...
}
```

Kód 6. Ukážka *Remove assignments to parameters*

3.1.6.1 Motivácia

Je nutné si ujasniť to, čo to znamená priradiť hodnotu parametru. Je to prípad, keď do metódy vstupuje parameter, ktorý sa odkazuje na nejaký objekt a my priradením tomuto parametru zmeníme objekt, ktorý odkazuje. Potom zavolanie metódy na tomto objekte zmení stav objektu vnútri metódy ale už nie mimo nej, čiže na mieste kde bola daná metóda volaná. Ak je to zámer, je vhodnejšie použiť dočasnú premennú a upravovať ju, zvyšuje to čitateľnosť kódu a je jasné, že dané správanie je zámerom programátora a nie je to len výsledkom správania sa jazyka (Kód 6.).

Parametre môžu byť predávané dvoma spôsobmi a to odkazom a hodnotou. Objekty sú v predávané štandardne odkazom a primitívne typy (int, double, byte, short, bool...) hodnotou. Pri predaní parametra hodnotou, sa pri zmene vnútri metódy hodnota mimo metódy nezmení. Je oveľa jasnejšie a čistejšie používať parametre ako vstupné dáta do metódy a prípade potreby zmeny hodnoty parametra použiť dočasnú premennú. Práve na tento účel slúži tento refaktoring.

3.1.6.2 Postup

i. Vytvoríme dočasnú premennú pre daný parameter a priradíme jej hodnotu parametra.
ii. Nahradíme všetky výskyty použitia parametra dočasnou premennou.
iii. Preložíme a spustíme príslušné testy.

3.1.7 Nahradenie metódy objektom (*Replace method with method object*)

Pokiaľ máme dlhú metódu, ktorá používa svoje lokálne premenné takým spôsobom, že je veľmi ťažké použiť *Extract method,* môžeme vytvoriť samostatný objekt, ktorý bude danú funkcionalitu zapuzdrovať. Lokálne premenné metódy sa stanú atribútmi nového objektu, čo nám umožní metódu v rámci objektu rozdeliť na viacero kratších metód.

3.1.7.1 Motivácia

Ako sme si už niekoľko krát povedali, je dobrým zvykom programovať krátke a jasné metódy. Ak je problém rozložený na viacero menších problémov a každý z nich je riešený na samostatnom mieste, je jednoduchšie pochopiť celý problém.

Pri rozkladaní dlhých metód na kratšie vzniká problém s lokálnymi premennými danej metódy. Niekedy sú lokálne premenné použité tak zložito, že použitie *Extract method* je veľmi ťažké, niekedy skoro nemožné.

3.1.7.2 Postup

i. Vytvoríme triedu, ktorú pomenujeme podľa metódy, ktorú refaktorujeme.
ii. Novej triede pridáme referenciu na zdrojový objekt a atribúty na všetky lokálne premenné, dočasné premenné a parametre pôvodnej metódy.
iii. Vytvoríme konštruktor, ktorý bude mať ako parametre zdrojový objekt a všetky parametre pôvodnej metódy.
iv. Vytvoríme verejnú metódu, ktorú nazveme „compute“.
v. Skopírujeme telo pôvodnej metódy do novo vytvorenej metódy v novej triede.
vi. Preložíme.
vii. Nahradíme kód pôvodnej metódy vytvorením objektu novej triedy s príslušnými parametrami a zavolaním metódy „compute“.

Keďže sú teraz všetky lokálne a dočasné premenné atribúty triedy, môžeme metódu algoritmu rozdeliť na viacero kratších metód, bez toho aby sme si museli posielať parametre. Toto sme práve kvôli zložitému použitiu lokálnych premenných v pôvodnej metóde nemohli.

3.1.8 Náhrada algoritmu (*Substitute algorithm*)

Refaktoring *Substitute algorithm* (náhrada algoritmu) slúži na kompletnú zmenu kódu algoritmu efektívnejším alebo čistejším riešením.

3.1.8.1 Motivácia

Refaktoring ako celok nám môže pomôcť rozdeliť komplexný kód na viacero menších a jednoduchších častí, niekedy však narazíme na algoritmus, ktorý síce funguje správne, ale je napísaný veľmi zložito. Ak poznáme jednoduchšie riešenie, mali by sme tento algoritmus celý odstrániť a nahradiť jednoduchším alebo prehľadnejším riešením. K takejto situácii môže dôjsť, keď sa postupne o danej problematike dozvieme viac, nájdeme nejaký knižnicu, ktorá nám pri riešení môže pomôcť alebo jednoducho tým, že ako sa ako programátori zlepšujeme a sme v danej technológii zručnejší ako keď sme algoritmus napísali prvý krát.

Ak by sme potrebovali upraviť algoritmus tak, aby pracoval trochu inak, je vhodné ešte pred samotnou úpravou použiť *Substitute algorithm* a nahradiť algoritmus niečím jednoduchším. Keď sa rozhodneme použiť tento refaktoring je dobré uistiť sa, či je daný kód rozložený na menšie časti aby bola náhrada čo najjednoduchšia, prípadne sa dala rozdeliť na viacero krokov.

3.1.8.2 Postup

i. Pripravíme si jednoduchšiu verziu algoritmu.
ii. Nahradíme pôvodný algoritmus novým
iii. Preložíme a spustíme testy.

3.2 Rozhranie metód

Objektovo orientované programovanie je z veľkej časti o rozhraniach. Vytvoriť rozhranie, ktoré je zrozumiteľné a ľahko použiteľné je jednou z najdôležitejších schopností pri programovaní dobrého objektovo orientovaného software. Táto kapitola sa venuje refaktoringom, ktoré súvisia práve s rozhraniami.

Jednou z najjednoduchších vecí pri úprave rozhrania je premenovanie metódy (*Rename method*) alebo triedy. Jasné názvy sú veľmi dôležité, keď rozumieme názvu metódy, vieme si predstaviť čo robí, vieme ju aj použiť v správnej situácii.

Ďalšími často používanými refaktoringami v tejto oblasti sú *Add parameter* a *Remove parameter*. V neobjektových programovacích jazykoch sú typické funkcie s dlhým zoznamom parametrov, pri objektovo orientovanom programovaní je to opačne. Používajú sa metódy s menším počtom parametrov a existujú refaktoringy na ďalšie zníženie počtu parametrov.

Ak máme metódu, ktorá očakáva ako parameter viacero hodnôt z jedného objektu, je vhodnejšie ako parameter poslať celý tento objekt (*Preserve whole object*). Ak takýto objekt neexistuje, môžeme ho vytvoriť a potom použiť (*Introduce parameter object*). Ak parametre dostáva metóda z objektu, do ktorého ma prístup môžeme tieto parametre odstrániť použitím refaktoringu *Replace parameter with method*. Pomocou *Parametrize method* je možné skombinovať niekoľko podobných metód.

Ďalším užitočným refaktoringom v tejto oblasti je *Separate query from modifier,* ktorým od seba oddelíme metódy, ktoré menia stav objektu a len vracajú nejakú hodnotu na základe stavu objektu ale nemenia ho. Dobré rozhranie navonok ukazujú len to, čo je naozaj nutné, to znamená, že metódy, ktoré môžu byť schované by mali byť schované (*Hide method* a *Remove setting method*).

Pri vytváraní objektu určitej triedy pomocou konštruktoru musí programátor poznať triedu, ktorej objekt chce vytvoriť, to však neplatí vždy. Práve pri takýchto situáciách môžeme miesto konstruktoru použit *Replace constructor with factory method.*

Jednou z vlastností moderných objektovo orientovaných jazykov je mechanizmus spracovávania chýb pomocou výnimiek. Programátori, ktorý s objektovo orientovaným programovaním nemajú moc skúseností používajú tzv. chybové kódy ako návraty metód, aby tak dali najavo, že nastala chyba. Refaktoring *Replace error code with exception* slúži na odstránenie takéhoto spôsobu ošetrovania chýb. Cez to všetko, že je koncept výnimiek dobrý a veľmi užitočný, nehodí sa vždy výnimky používať, vhodnejšie je použiť podmienku (*Replace exception with test*).

3.2.1 Premenovanie metódy (*Rename method*)

Rename method (premenovanie metódy) je refaktoring, ktorý sa používa na zmenu názvu metódy, keď jej názov nezodpovedá jej funkcii.

```
public decimal GetUsrAccWLmt(User u)
{
        ...
}
```

```
public decimal GetUserAccountWeekLimit(User user)
{
        ...
}
```

Kód 7. Ukážka *Rename method*

3.2.1.1 Motivácia

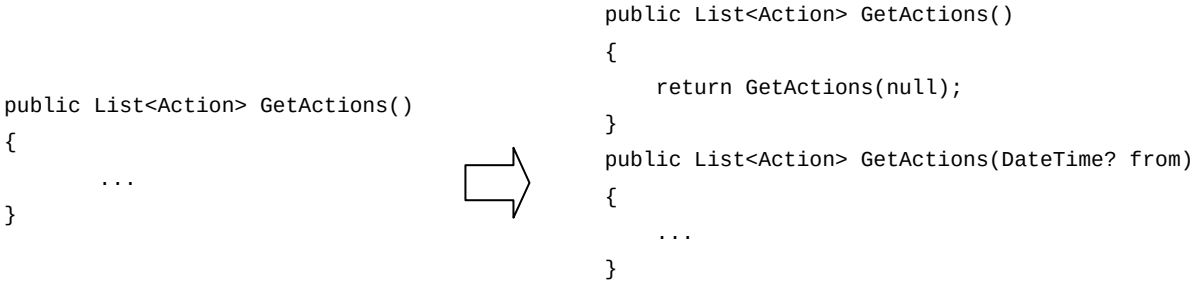
Ako som už spomenul, býva dobrým zvykom rozložiť kód na viacero krátkych metód. Oproti dlhým a komplexným metódam to ma radu výhod. Ak je sú však tieto krátke metódy pomenované zle, môže nám to viac uškodiť ako pomôcť. Stratíme veľa času zisťovaním, čo vlastne daná metóda robí, pretože to nie je z jej názvu jasné. Najjednoduchším spôsobom ako dobre pomenovať metódu je predstaviť si komentár popisujúci chovanie metódy a použiť ho ako názov. Je to však ťažšie ako to môže vyzerať, hneď na prvý pokus sa nám obvykle nepodarí vytvoriť správny a vystihujúci názov. Preto, keď postupom času prídeme na to, že sa je daná metóda pomenovaná nesprávne alebo, že by bolo vhodnejšie pomenovať ju inak, nemali by sme to nechať len tak s tým, že to nie je dôležité, že je to „iba“ názov metódy. Mali by sme použiť refaktoring *Rename method* a zmeniť názov na správnejší (Kód 7.). Musíme mať stále na mysli, že kód okrem samotného prekladača budú čítať predovšetkým ľudia. Správne pomenovanie metód alebo tried je schopnosť, ktorá sa pravdepodobne nedá naučiť, dá sa postupne získať len a len praxou.

3.2.1.2 Postup

i. Zistíme, či je daná metóda implementovaná v predkovi alebo potomkovi triedy. Ak áno, opakujeme tento postup pre každú implementáciu.
ii. Deklarujeme novú metódu s novým názvom, skopírujeme telo pôvodnej metódy.
iii. Zmeníme pôvodnú metódu tak aby volala novú metódu.
iv. Preložíme a spustíme testy.
v. Vyhľadáme všetky volania starej metódy a nahradíme ich volaním novej metódy. Po každej zmene kód preložíme a otestujeme.
vi. Pokiaľ nie je stará metóda súčasťou verejného rozhrania, môžeme ju odstrániť.
vii. Preložíme a spustíme testy.

3.2.2 Pridanie parametra (*Add parameters*)

Add parameter (pridanie parametra) sa požíva keď metóda potrebuje viac informácii od volajúceho.



Kód 8. Ukážka *Add parameters*

3.2.2.1 Motivácia

Pridanie parametra je veľmi častým a používaným refaktoringom, ktorý už pravdepodobne použil každý programátor a to nie raz. Dôvod alebo motivácia k tomuto refaktoringu je veľmi jednoduchá. Potrebujeme upraviť metódu a táto zmena vyžaduje informácie, ktoré pred tým nemala k dispozícii. Riešením je pridanie nového parametra, prípadne viacerých parametrov (Kód 8.).

Zvyšovaním počtu parametrov sa však z kódu stáva kód menej čitateľný. Preto ešte predtým ako parameter pridáme by sme si mali premyslieť, či je tento krok nutný a či neexistuje iné, čistejšie riešenie. Je totiž možné že novú informáciu, ktorú metóda potrebuje sa dá získať už posielaných parametrov. Ďalšou možnosťou môže byť rozšírenie vlastností objektov posielaných ako parametre aby sa na základe nich dala požadovaná informácia získať. Ak však neexistuje iná možnosť ako rozšíriť zoznam parametrov, môžeme ešte uvážiť použitie refaktoringu *Introduce parameter object.*

3.2.2.2 Postup

i. Zistíme, či nie je daná metóda implementovaná v predkovi alebo potomkovi. Ak áno je tento postup nutné zopakovať pre každú z implementácií.
ii. Deklarujeme novú metódu s novým zoznamom parametrov a skopírujeme telo starej metódy.
iii. Preložíme.
iv. Zmeníme telo starej metódy tak aby volala novú metódu. Hodnoty nových parametrov buď zadáme (ak máme nejaké default hodnoty) alebo v prípade objektov, zadáme *null*.
v. Preložíme a spustíme príslušné testy.
vi. Vyhľadáme všetky volania starej metódy a nahradíme ich volaním novej. Po každej náhrade kód preložíme a otestujeme.
vii. Odstránime starú metódu. Ak je však metóda súčasťou nejakého verejného rozhrania, nemôžeme ju vymazať. Preto ju len označíme atribútom *Obsolete,* čím dáme najavo, že ide o starú metódu a je potreba prejsť na novú
viii. Preložíme a spustíme testy.

3.2.3 Odstránenie parametra (*Remove parameter*)

Tento refaktoring sa používa v prípade keď metóda nejaký z parametrov už nepoužíva, vtedy je potrebné ho odstrániť (*Remove parameter*).

```
public Order GetOrder(byte sequence)
{
   ...
}
```

```
public Order GetOrder()
{
   ...
}
```

Kód 9. Ukážka *Remove parameter*

3.2.3.1 Motivácia

Programátori veľmi často a veľmi radi používajú predchádzajúci refaktoring (*Add parameter*) keď potrebujú metódu rozšíriť, často krát ale zabúdajú už nepoužívané parametre rušiť a zjednodušovať tak rozhrania metód (Kód 9.). Na jednu stranu, tým že nepotrebný parameter nezrušíme nijak neovplyvníme správanie a funkčnosť metódy, tak prečo si dávať prácu s odstraňovaním parametra a následným opravovaním volania metódy?

Je nutné si uvedomiť, že parametre metódy sú na to aby sa používali. Z pohľadu používateľa danej metódy je mätúce, keď do metódy pošle rôzne hodnoty a dostane rovnaký výsledok aj keď očakával niečo iné. Tento refaktoring je veľmi jednoduchý a treba ho teda jednoznačne používať. Problém môže nastať ak s parametrom, ktorý chceme odstrániť pracuje nejaká z iných implementácii, či už predok alebo potomok danej triedy.

3.2.3.2 Postup

i. Zistíme, či je daná metóda implementovaná v nejakom z predkov alebo potomkov danej triedy a či využíva parameter, ktorý chceme odstrániť. Ak áno, nebudeme tento refaktoring robiť.
ii. Deklarujeme novú metódu už bez parametra, ktorý chceme odstrániť a skopírujeme telo zo starej metódy.
iii. Preložíme.
iv. Upravíme telo pôvodnej metódy tak, aby metóda volala novú metódu.
v. Preložíme a spustíme príslušné testy.
vi. Nájdeme všetky volanie starej metódy a nahradíme ich volaním novej metódy. Po každej náhrade, kód preložíme a otestujeme.
ix. Odstránime starú metódu. Ak je však metóda súčasťou nejakého verejného rozhrania, nemôžeme ju vymazať. Preto ju len označíme atribútom *Obsolete,* čím dáme najavo, že ide o starú metódu a je potreba prejsť na novú.
x. Preložíme a spustíme testy.

3.2.4 Oddelenie dotazu od zmeny (*Separate query from modifier*)

Ak máme metódu, ktorá vracia stav objektu a zároveň jeho stav mení, mali by sme z tejto metódy vytvoriť dve samostatné. Jednu na vrátenie stavu a druhú na zmenu stavu objektu. Na tento účel slúži refaktoring *Separate query from modifier* (oddelenie dotazu od zmeny).

```
decimal GetAccountBalanceAndSetDisabledFlag()
```

```
decimal GetAccountBalance()
void SetDisabledFlag()
```

Kód 10. Ukážka *Separate query from modifier*

3.2.4.1 Motivácia

Metóda, ktorá nemení stav objektu ale len vracia nejakú z jeho vlastností, prípadne vypočíta nejakú hodnotu na základe jeho vlastností je veľmi užitočná, môžeme ju použiť všade tam kde ju potrebujeme bez toho aby sme museli rozmýšľať v kedy ju môžeme použiť, pripadne volanie môžeme bez obáv presunúť na iné miesto.

Je dobrým zvykom oddeliť od seba metódy, ktoré stav objektu nemenia a tie ktoré ho menia. Jedným z postupov ako takéto metódy rozdeliť je, že ak metóda vracia hodnotu, nemala by meniť stav objektu, potom je na prvý pohľad jasné ktoré z metód stav menia a ktoré nie. Osobne s týmto postupom príliš nesúhlasím. Uznávam, že je na prvý pohľad jasné, ktorá metóda slúži na čo, ale niekedy je potrebné vykonať na objekte určitú operáciu a zistiť ako dopadla a potrebný výsledok si vrátiť. Ak by táto metóda nevrátila nič, je následne nutné pridať kód, ktorý by testoval stav objektu aby bolo jasné či bola daná operácia úspešná. Čo v prípade použitia operácie na viacerých miestach znamená duplikovanie kódu na zistenie výsledku alebo úspešnosti operácie a to ako sme si už povedali je jedným zo znakov toho, že kód nie je navrhnutý dobre a je nutný refaktoring. Práve preto by som sa ja použitiu tohto refaktoringu vyhol, uvádzam ho len pre úplnosť katalógu.

3.2.4.2 Postup

i. Vytvoríme novú metódu, ktorá bude vracať to, čo pôvodná metóda.
ii. Upravíme pôvodnú metódu tak, aby všade tam kde vracia hodnotu vracala výsledok novo vytvorenej metódy.
iii. Preložíme a spustíme testy.
iv. Všetky volania pôvodnej metódy nahradíme volaním novo vytvorenej metódy, ktorá vracia hodnotu. Ešte predtým zavoláme pôvodnú metódu tak, že jej výsledok nikam nepriradíme.
v. Pôvodnej metóde upravíme návratový typ na void a odstránime všetky return výrazy.
vi. Preložíme a spustíme testy.

3.2.5 Parametrizácia metódy (*Parametrize method*)

Tento refaktoring môžeme použiť ak viac metód s podobnou funkcionalitou s tým rozdielom, že pracujú s rôznymi hodnotami na vstupe, ktoré majú definované v ich tele.

```
private void RaiseByTenPercentage()
private void RaiseByFivePercentage()
```

```
private void Raise(byte percentage)
```

Kód 11. Ukážka *Parametrize method*

3.2.5.1 Motivácia

Duplikovaný kód, problém s ktorým sa pri programovaná často stretávame. Jedným z príkladov duplicity kódu je, keď máme skupinu metód, ktoré vykonávajú nejakú z variant tej istej funkcionality s rôznymi vstupnými parametrami. Vtedy je vhodné z týchto metód vytvoriť jednu a variantu funkcionality riadiť cez jej parametre (Kód 11.). To zvyšujú flexibilitu riešenia, ďalšie varianty funkcionality vytvoríme pridaním parametra a samozrejme odstraňuje duplicitu v kóde.

3.2.5.2 Postup

i. Vytvoríme novú metódu s takými parametrami aby ju bolo možné použiť miesto nahradzovaných metód.
ii. Preložíme.
iii. Postupne nahradzujeme pôvodnú metódu volaním novej metódy s príslušnými parametrami.
iv. Preložíme a spustíme príslušné testy. Opakujeme postup pre každú z pôvodných metód

Môžeme naraziť na situáciu, kedy nie je možné takto odstrániť celé metódy, ale len ich časti. Vtedy ako prvý krok použijeme *Extract method* na vytvorenie skupiny metód, ktoré sa dajú vyššie popísaným postupom zlúčiť do jednej.

3.2.6 Zachovanie celého objektu (*Preserve whole object*)

Ak máme metódu, do ktorej ako parametre chodia viaceré hodnoty z jedného objektu je vhodnejšie ako parameter posielať celý tento objekt.

```
string name = this._user.FirstName + “” +this._user.LastName;
int age = this._user.Age;
PrintUserInfo(name, age);
```

```
PrintUserInfo(this._user);
```

Kód 12. Ukážka *Preserve whole object*

3.2.6.1 Motivácia

Táto situácia nastáva keď viaceré hodnoty z jedného objektu posielame do metódy ako samostatné parametre. Vhodnejšie je poslať celý objekt (Kód 12.). To má hneď niekoľko výhod. Prvá z nich je, že sa skráti počet parametrov, tým pádom je kód čitateľnejší. Ďalšou výhodou je, že ak bude v budúcnosti daná metóda potrebovať ďalšie hodnoty z tohto objektu, stačí zmeniť kód danej metódy, aby tieto hodnoty využívala, rozhranie sa nezmení.

Je tu však aj nevýhoda použitia tohto refaktoringu. Keď sa ako parametre používajú iba hodnoty a nie celé objekty nie je metóda nijak zviazaná s objektom, z ktorého hodnoty prišli. V prípade, že pošleme ako parameter celý objekt vznikne tak závislosť metódy na objekte. Ak je táto závislosť nemožná alebo nevhodná kvôli architektúre systému, nemali by sme tento refaktoring použiť.

V prípade, že nemáme definovaný objekt, ktorý by tieto parametre obsahoval, môžeme použiť *Introduce parameter object* a tento objekt vytvoriť.

3.2.6.2 Postup

i. Upravovanej metóde pridáme nový parameter (*Add parameter*) celého objektu, ktorého informácie sa v metóde používajú.
ii. Preložíme a spustíme testy.
iii. Zistíme, ktoré z parametrov sa dajú získať z objektu, ktorý sme pridali ako nový parameter.

3.2.7 Objekt ako parameter (*Introduce parameter object*)

Ak do metódy posielame skupinu parametrov, ktoré spolu logicky súvisia, je vhodné pre ne vytvoriť triedu a ako parameter potom posielať objekt tejto triedy.

```
public List<Order> GetOrders(DateTime? from, DateTime? to);
public List<Order> DeleteOrders(DateTime? from, DateTime? to);
```

```
public List<Order> GetOrders(DateTimeInterval  interval);
public List<Order> DeleteOrders(DateTimeInterval interval);
```

Kód 13. Ukážka *Introduce parameter object*

3.2.7.1 Motivácia

Často môžeme vidieť skupinu premenných, ktoré spolu nejako súvisia. Táto skupina sa môže používať ako parametre nejakej metódy, viacerých metód alebo ako atribúty tried. Keď na niečo také narazíme je vhodné stráviť chvíľu tým, že túto skupinu premenných spojíme do samostatnej triedy (Kód 13.). Tým zjednodušíme rozhranie metód, ktoré tieto premenné používame (ako sme si už povedali, kratšie zoznamy parametrov sú jednoduchšie na porozumenie).

3.2.7.2 Postup

i. Vytvoríme novú triedu, ktorá bude reprezentovať skupinu parametrov, ktoré chceme nahradiť.
ii. Preložíme kód.
iii. Použijeme *Add parameter* na pridanie nového parametra (objektu novo vytvorenej triedy). Na všetkých miestach kde sa dané metódy používajú
iv. Na všetkých miestach kde sa daná metóda volá budeme do tohto parametra posielať *null.*
v. Preložíme kód a spustíme testy.
vi. Postupne prechádzame po parametroch a v tele metódy nahradzujeme ich použitie použitím hodnoty z nového parametra. Zároveň pri všetkých volaniach hodnotou daného parametra naplníme novo vytvorený objekt (nový parameter).
vii. Spracovaný parameter odstránime z hlavičky metódy.
viii. Opakujeme až kým neodstránime všetky parametre, ktoré sú teraz reprezentované novým parametrom (triedou vytvorenou v prvom bode).

3.2.8 Odstránenie modifikátora (*Remove setting method*)

Tento refaktoring sa používa na odstránenie metódy, ktorá nastavuje hodnotu atribútu, ktorý mal slúžiť iba na čítanie.

```
public int OrderId
{
    get { return this._order.Id; }
    set { this._order.Id = value; }
}
```

```
public int OrderId
{
    get { return this._order.Id; }
}
```

Kód 14. Ukážka *Remove setting method*

3.2.8.1 Motivácia

Ak nejaká trieda obsahuje atribúty, ktoré sú nastavené pri vytvorení objektu a nemali by byť počas jeho života menené, nemala by trieda obsahovať metódu (vlastnosť) cez ktorú je možné tento atribút nastaviť (Kód 14.). V praxi sa však často stretávame s tým, že pre všetky atribúty triedy existujú metódy (vlastnosti), cez ktoré je možné atribút čítať ako aj nastavovať. Programátori totiž automaticky pri programovaní bez rozmyslu vytvárajú vlastnosti, ktoré sa dajú čítať a aj nastavovať. Tým, že niektorú z vlastnosti sprístupnime iba na čítanie, jasne dáme používateľovi triedy najavo ako sa daná vlastnosť používa, preto je dobre tento spôsob a aj tento refaktoring používať.

3.2.8.2 Postup

i. Zistíme, kde všade je vlastnosť nastavovaná, mali by to byť iba konštruktori, ktoré túto vlastnosť nastavujú.
ii. Ak sa v konštruktoroch nastavuje hodnota týchto atribútov cez verejnú vlastnosť (metódu), upravíme kód tak aby nastavoval priamo privátny atribút.
iii. Preložíme a spustíme testy.
iv. Upravíme verejnú vlastnosť tak, aby nebolo možné atribút nastaviť.
v. Preložíme.

3.2.9 Schovať metódu (*Hide method*)

Refaktoring *Hide method* (schovať metódu) sa používa na metódy, ktoré nie sú mimo triedy, v ktorej sú deklarované použité (Kód 15). Takéto metódy sa označia ako súkromné (modifikátor *private,* prípadne *internal*).

```
public int PrivateMethod()
{
    ...
}
```

```
private int PrivateMethod()
{
    ...
}
```

Kód 15. Ukážka *Hide method*

3.2.9.1 Motivácia

Pri aplikovaní refaktoringu stojíme často pred rozhodnutiami ohľadom viditeľnosti metód. Je relatívne jednoduché v prípade potreby spraviť zo súkromnej metódy verejnú. Táto situácia nastane, keď ostatné objekty požadujú po nejakom ďalšom objekte funkcionalitu, ktorá je momentálne skrytá. Vtedy metódu jednoducho označíme ako verejnú. Ťažšie už býva zhodnotiť, ktoré z metód prípadne vlastností by mali byť pred okolitým svetom skryté. Mali by sme pravidelne prechádzať kódom a dbať na to, že metódy, ktoré sú mimo triedy nepoužívané my sme mali označiť ako súkromné. Vhodné je to hlavne u tried, ktoré sú komplexnejšie a ich rozhranie je bohaté. Tým, že skryjeme nepoužívané metódy výrazne zjednodušíme a sprehľadníme jej rozhranie.

3.2.9.2 Postup

i. Pravidelná kontrola (revízia) kódu s cieľom skryť nepoužívané metódy.
ii. Ak takúto metódu nájdeme mali by sme ju urobiť čo najviac súkromnú.
iii. Po každej zmene preložiť kód. Prekladač prípadnú chybu odhalí, takže nie je nutné spúšťať testy.

3.2.10 Nahradenie konštruktora metódou (*Replace constructor with factory method*)

Tento refaktoring slúži na jednoduchšie vytváranie objektov. Názov tohto refaktoringu je *Replace constructor with factory method*, čo znamená nahradenie konštruktora metódou.

```
public Account(AccountType type)
{
    this._type = type;
}
```

```
static Account CreateAccount(AccountType type)
{
    switch(type)
    {
        case AccountType.Regular:
           return new Account();
        case AccountType.Student:
           return new StudentAccount();
        ...
  }
}
```

Kód 16. Ukážka *Replace constructor with factory method*

3.2.10.1 Motivácia

Typickým príkladom použitia tohto refaktoringu je situácia keď máme triedu a jej objekty môžu byť rôzneho typu (typom sa myslí napríklad typ bankového účtu, nie typ v programovacom jazyku). Ako parameter konštruktoru je poslaný typ požadovaného objektu, ten je vytvorený a jedna z jeho vlastností tento typ reprezentuje, aby ho bolo v budúcnosti možné k danému typu priradiť. Táto konštrukcia je v praxi celkom bežná. Bežne však nastáva aj situácia, keď je túto konštrukciu nutné nahradiť hierarchiou tried. To znamená, že každý typ objektu bude reprezentovaný samostatnou triedou. Tu však konštruktory nemôžeme použiť, pretože tie nám môžu vrátiť iba objekt triedy, ktorej konštruktor voláme. Na to využijem *factory* metódu (pojem *Factory* je známy z oblasti návrhových vzorov [3]). Tá nám poskytne jednotný a jednoduchý spôsob vytvárania jednotlivých typov triedy. Takýto prístup môžeme použiť všade tam, kde je použitie konštruktorov zložité, neprehľadné alebo sa jednoducho nehodí.

3.2.10.2 Postup

i. Vytvoríme *factory* metódu a v jej tele vytvoríme a vrátime objekt súčasným konštruktorom.
ii. Nahradíme všetky výskyty vytvárania objektu konštruktorom za vytváranie pomocou novo vytvorenej metódy.
iii. Po každej náhrade preložíme a spustíme testy.

[3] Gamma, E., R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, 1995.

iv. Označíme konštruktor ako súkromný (modifikátor *private*).

v. Preložíme.

3.3 Generalizácia

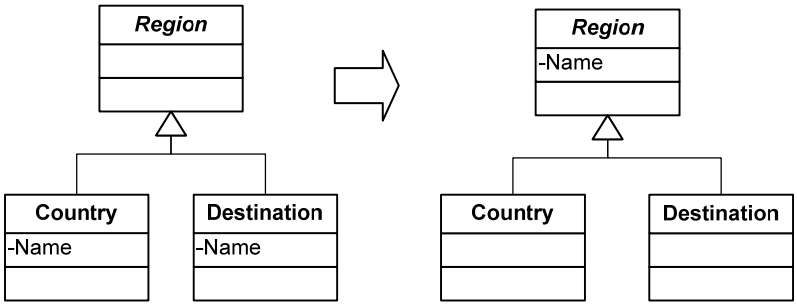
Generalizácia produkuje ďalšiu špecifickú skupinu refaktoringov. Väčšinou ide o presúvanie metód po strome dedičnosti, či už je smerom nahor alebo nadol. Príkladom refaktoringov, ktoré presúvajú metódy alebo vlastnosti po hierarchii smerom nahor sú *Pull Up Field* a *Pull Up Field,* presným opakom k nim sú refaktoringy *Push Down Method* a *Push Down Field.* Konštruktor ako špeciálny druh metódy nemôžeme presúvať po hierarchii dedičnosti, môžeme však presúvať jeho telo, refaktoring *Pull Up Constructor Body* slúži na presunutie tela konstruktoru smerom nahor. Namiesto presúvania tela konštruktora smerom nadol môžeme použiť refaktoring *Replace Constructor with Factory Method*, ktorý je popísaný v predchádzajúcej kapitole.

Ak máme viac metód, ktoré majú takmer rovnaké telo, ale líšia sa v nejakých detailoch je vhodné použiť refaktoring *Form Template Method*, ktorý nám umožní oddeliť len miesta, v ktorých sa metódy líšia.

Spolu s presúvaním metód a vlastností tried, je možné upravovať strom dedičnosti vytváraním nových tried. *Extract Subclass, Extract Superclass* a *Extract Interface* sú refaktoringy, ktorých výsledkom je vytvorenie nového elementu. Ak po čase natrafíme na triedy, ktoré sa nepoužívajú alebo zistíme, že je strom dedičnosti príliš zložitý môžeme použiť refaktoring *Collapse Hierarchy* a nepotrebné triedy odstrániť.

3.3.1 Vytiahnutie atribútu (*Pull Up Field*)

Pull Up Field (vytiahnutie atribútu) použijeme vtedy ak majú viaceré triedy rovnakú vlastnosť. Presunieme ju do nadradenej triedy.



Obr. 1. Ukážka *Pull Up Field*

3.3.1.1 Motivácia

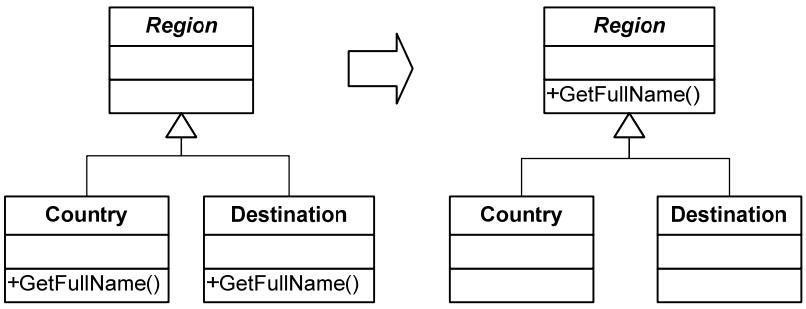
Ak sú triedy, ktoré majú spoločného predka vyvíjané nezávisle, napríklad rôznymi programátormi alebo sú výsledkom nejakého predchádzajúceho refaktoringu, často sa v nich bude vyskytovať duplicitný kód. Môže ísť napríklad o vlastnosti. Tieto vlastnosti môžu mať rovnaký alebo podobný názov, na to aby sme takého metódy mohli identifikovať je nutné sa pozrieť ako sú tieto vlastnosti používané. Ak zistíme, že sú používané podobne, môžeme ich generalizovať (presunúť do triedy predka). Výsledkom je odstránenie duplicity, čo ako sme si povedali v úvodných kapitolách zvyšuje kvalitu kódu a prispieva k lepšej udržovateľnosti (Obr. 1.).

3.3.1.2 Postup

i. Preskúmať použitie vlastností, či sú používané rovnakým spôsobom.
ii. Ak vlastnosti nemajú rovnaké názvy, treba ich premenovať. Použijeme názov, ktorý bude potom použitý v triede predka.
iii. Preložíme a spustíme testy.
iv. Vytvoríme novú vlastnosť v triede predka. Ak ide o súkromnú premennú je nutné z nej spraviť premennú *protected* aby bola viditeľná v potomkoch.
v. Preložíme a spustíme testy.
vi. Ak ide o privátnu premennú, resp. protected, mali by sme ju zapuzdriť do protected vlastnosti a k jej hodnote pristupovať cez ňu.

3.3.2 Vytiahnutie metódy (*Pull Up Method*)

V prípade, že máme rovnaké metódy, ktoré vracajú rovnaké výsledky vo viacerých triedach, použijeme tento refaktoring (*Pull Up Method* – vytiahnutie metódy) a danú metódu presunieme do nadradenej triedy.



Obr. 2. Ukážka *Pull Up Method*

3.3.2.1 Motivácia

Odstránenie duplicitného kódu je veľmi dôležité. Aj keď kód, ktorý je skopírovaný na viaceré miesta plne funkčný, nie je nič horšie ladiť chybu a a opravu distribuovať na všetky miesta kde je daný kód rozkopírovaný. Do systému sa tak zanášajú nové a nové chyby. Nie je totiž vždy jednoduché duplikovaný kód nájsť.

Preto keď narazíme na metódy, ktoré sú v dvoch rôznych triedach identické mali by sme ich presunúť do nadradenej triedy, práve použitím tohto refaktoringu. Nie je to však vždy jednoduché, metódy môžu mať rovnaké telo, ale už nemusia mať rovnaký názov a zoznam parametrov. Preto aj identifikácia takýchto metód môže byť zložitá.

Tento refaktoring je väčšinou používaný v dvoch krokoch. Jedným môže byť premenovanie metód tak aby sme zjednotili ich názov alebo úprava zoznamu parametrov, tak aby sme naozaj dostali identické metódy. Niekedy môže nastať situácia keď metóda používa elementy (vlastnosti, premenné alebo metódy), ktoré sú definované v podtriede a nie v nadradenej triede. Vtedy môžeme použiť tento refaktoring aj na tieto elementy a presunúť ich do nadradenej triedy (Obr. 2.).

Ak sa však refaktorované metódy v niečom líšia, použijeme refaktoring *Form template method*, ktorý je popísaný v závere tejto kapitoly.

3.3.2.2 Postup

i. Preveríme, že sú metódy, ktoré chceme refaktorovať naozaj identické. Ak nie sú a je to možné upravíme ich tak aby identické boli.
ii. Ak majú triedy rôznu hlavičky, upravíme obe tak ako chceme aby boli používané v nadradenej triede.

iii. Vytvoríme novú metódu v nadradenej triede a označíme ju ako *virtual*, v podtriedach dané metódy označíme modifikátorom *override*.
iv. Preložíme a spustíme testy.
v. Skopírujeme telo jednej metódy do nadradenej triedy a vymažeme ju v podradenej triede.
vi. Preložíme a spustíme testy.
vii. Prezrieme používanie tejto metódy a tam kde to ide zmeníme typ na nadradenú triedu.

3.3.3 Vytiahnutie konštruktora (*Pull Up Constructor Body*)

Tento refaktoring použijeme ak konštruktory tried so spoločným predkom obsahujú spoločný kód.

```
public Country : CommonBusinessObject {
        public Country(int id, string name, int population) {
                this._id = id;
                this._name = name;
                this._population = population;
        }
}
```

```
public Country : CommonBusinessObject {
        public Country(int id, string name, int population)
                : base (id, name)
        {
                this._population = population;
        }
}
```

Kód 17. Ukážka *Pull Up Constructor Body*

3.3.3.1 Motivácia

Konštruktory sú špeciálne druhy metód. Majú presne definované, čo sa v nich smie robiť a čo nie. Ako sme si povedali vyššie, ak majú triedy metódy, ktoré majú spoločný blok kódu, prvým krokom je vytvorenie z tohto bloku novú metódu (*Extract Method*) a potom túto metódu premiestniť do triedy predka aby ju mohli používať potomkovia, ktorý ju potrebujú (Kód 17.). Pri konstuktoroch je situácia podobná, často sa opakuje spoločný, inicializačný, blok kódu.

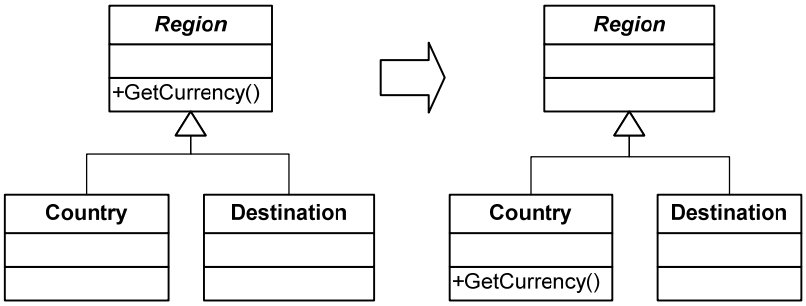
Ak nastane situácia kedy bude tento refaktoring príliš zložitý, mali by sme zvážiť použitie refaktoringu *Replace Constructor with Factory Method.*

3.3.3.2 Postup

i. Definujeme nový konštruktor v triede predka.
ii. Presunieme spoločný kód z konštruktora potomka do nového konštruktora v predkovi. Môže to byť aj všetok kód.
iii. Ako prvý krok v konštruktore potomka zavoláme novo vytvorený konštruktor predka.
iv. Preložíme a spustíme testy.

3.3.4 Posunutie metódy dolu (*Push Down Method*)

Metóda v triede predka je relevantná a iba pre niektorých potomkov. Presunieme ju do relevantných tried.



Obr. 3. Ukážka *Push Down Method*

3.3.4.1 Motivácia

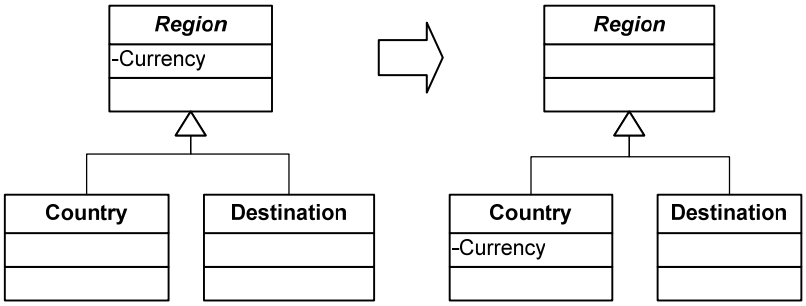
Tento refaktoring je protikladom k *Pull Up Method.* Používa sa pri presúvaní metód do trieda potomkov, kde majú väčší zmysel a logicky patria (Obr. 3.). Často sa používa spolu s refaktoringom *Extract Subclass.*

3.3.4.2 Postup

i. Deklarujeme metódu, ktorú presúvame vo všetkých podtriedach a skopírujeme jej telo. Môže byť potreba definovať aj privátne alebo protected premenné ktoré sa v metóde používajú.
ii. Odstránime metódu v triedy predka. Môže byť potreba upraviť volania danej metódy alebo typy parametrov metód a konštruktorov.
iii. Preložíme a spustíme testy.
iv. Odstránime metódu zo všetkých tried, ktoré ju nepoužívajú.
v. Preložíme a spustíme testy.

3.3.5 Posunutie atribútu dolu (*Push Down Field*)

Vlastnosť je deklarovaná v nadradenej triede ale používa sa len v jednom potomkovi, vhodnejšie je túto vlastnosť presunúť do tohto potomka.



Obr. 4. Ukážka *Push Down Field*

3.3.5.1 Motivácia

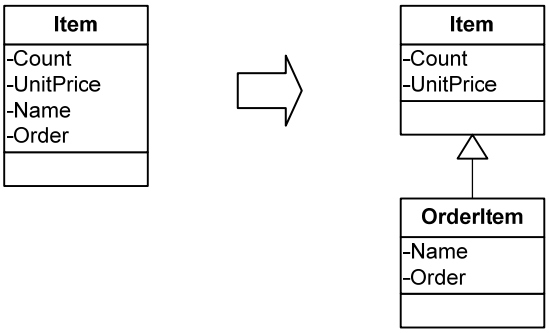
Tento refaktoring je protiklad refaktoringu *Pull Up Field.* Použijeme ho vtedy keď nejakú vlastnosť nepotrebujeme v triede predka ale len v jednom jej potomkovi (Obr. 4.).

3.3.5.2 Postup

i. Deklarujeme odstraňovanú vlastnosť vo všetkých potomkoch.
ii. Odstránime vlastnosť z triedy predka.
iii. Preložíme a spustíme testy.
iv. Odstránime deklaráciu vlastnosti zo všetkých tried kde sa daná vlastnosť nepoužíva.
v. Preložíme a pustíme testy.

3.3.6 Vyňatie podtriedy (*Extract Subclass*)

Trieda obsahuje vlastnosti alebo metódy, ktoré sa používajú len pre niektoré inštancie. Vytvoríme triedu, ktorá bude obsahovať túto skupinu funkcionalít.



Obr. 5. Ukážka *Extract Subclass*

3.3.6.1 Motivácia

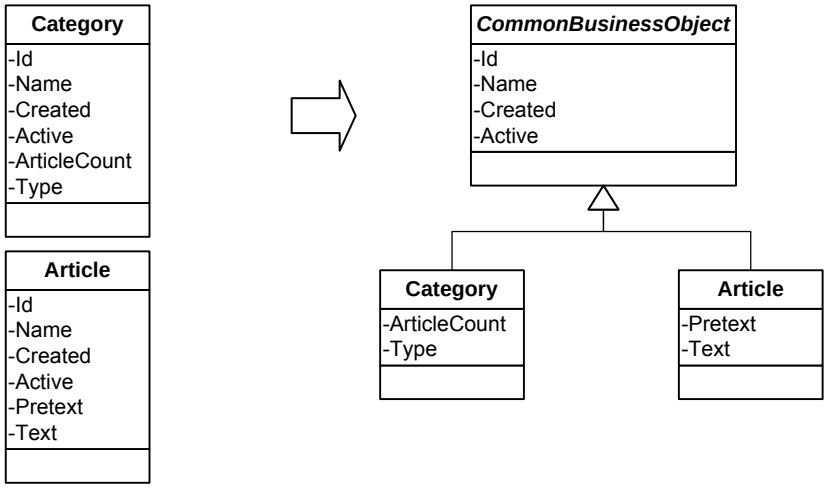
Hlavným dôvodom použitia tohto refaktoringu býva zistenie, že niektoré vlastnosti a metódy ktoré trieda obsahuje sú používané len niektorými inštanciami. To je obvykle znak toho, že by sme mali vytvoriť špeciálnu triedu, ktorá bude obsahovať práve tieto vlastnosti a metódy (Obr. 5.). K tomuto refaktoringu existuje alternatíva v podobe *Extract Class*, takže ide o otázku či bude vhodnejšia dedičnosť alebo delegovanie. *Extract subclass* je jednoduchšia cesta, ale má svoje obmedzenia. Napríklad nie je možné zmeniť správanie objektu po tom ako je vytvorený ako inštancia nejakej triedy. Pri delegovaní to možné je, len sa použije iný objekt, ktorý danú funkcionalitu poskytuje.

3.3.6.2 Postup

i. Vytvoríme novú triedu, ktorá bude dediť z refaktorovanej triedy.
ii. Vytvoríme konštruktory pre novú triedu.
iii. Všetky miesta kde sa vytvára objekt pôvodnej triedy a je vyžadovaná jej podtrieda upravíme vytváranie tak, že sa budem vytvárať objekt novej triedy
iv. Pre vlastnosti a metódy, ktoré má nová trieda obsahovať postupne použijeme refaktoringy *Push down field* a *Push down method.*
v. Po každej zmene kód preložíme a spustíme testy.

3.3.7 Vyňatie nadtriedy (*Extract Superclass*)

Dve triedy majú spoločnú funkcionalitu. Vytvoríme pre ne spoločného predka a spoločnú funkcionalitu presunieme do nej.



Obr. 6. Ukážka *Extract Superclass*

3.3.7.1 Motivácia

Ako sme si už v úvodných kapitolách povedali, duplicitný kód je jedným z najväčších problémov pri tvorbe software. Z duplikovania kódu plynie množstvo problémov, pri zmene sme nútení túto zmenu propagovať na všetky miesta kde je kód skopírovaný, čím je zmena časovo náročnejšia a náchylnejšia na zanesenie chýb do systému.

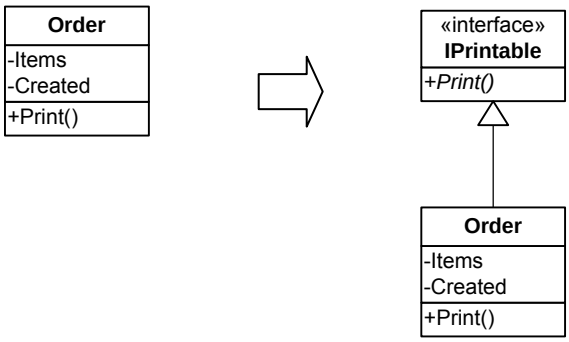
Jednou formou duplicitného kódu je keď viaceré triedy robia rovnaké alebo veľmi podobné veci. Dedičnosť je technikou, pomocou ktorej sa môžeme duplicitnému kódu vyhnúť. Túto možnosť si ale väčšinou uvedomíme až potom ako vytvoríme niekoľko tried, ktoré majú spoločné vlastnosti.

3.3.7.2 Postup

i. Vytvoríme prázdnu *abstraktnú* triedu a pôvodnú triedu upravíme tak aby dedila z novo vytvorenej triedy.
ii. Pre všetky spoločné vlastnosti a triedy postupne použijeme refaktoringy *Pull up field* a *Pull up method.*
iii. Po každej zmene program preložíme a spustíme testy.
iv. Prejdeme ešte metódy, ktoré zostali v podradených triedach, či nemôžeme použiť na nejaké spoločné bloky kódu *Extract method* a potom *Pull up method*.
v. Po všetkých zmenách prejdeme použitia daných tried, ak nájdeme miesta kde sa používajú len vlastnosti a metódy z nadradenej triedy, môžeme napríklad zmeniť napríklad typ parametra na nadradenú triedu.

3.3.8 Vyňatie rozhrania (*Extract Interface*)

Niekoľko tried má spoločnú skupinu vlastností alebo metód. Z tejto skupiny je vhodné vytvoriť rozhranie (*Extract Interface*).



Obr. 7. Ukážka *Extract Interface*

3.3.8.1 Motivácia

Triedy sú používané rôznymi spôsobmi, môže byť použité celé poskytované rozhranie alebo len časť z neho. Niekedy je určitou skupinou konzumentov triedy používaná len špecifická podmnožina rozhrania. Ďalším spôsobom býva pracovanie so všetkými triedami, ktoré dokážu spracovať potrebné požiadavky.

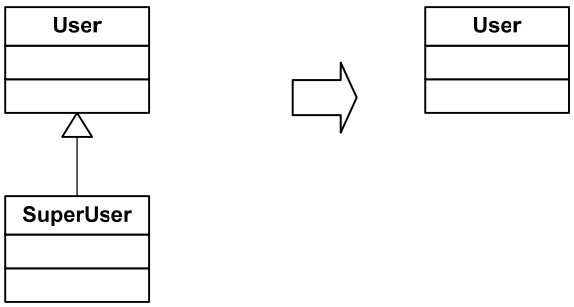
Pre tieto prípady je vhodné triedy navrhnúť tak aby sa na ne dalo pozerať z abstraktnejšieho pohľadu, ktorý zahŕňa požadovanú množinu operácií a vlastností. V jazykoch, ktoré podporujú viacnásobnú dedičnosť je situácia pomerne jednoduchá. Vytvoríme jednu triedu pre každú požadovanú skupinu vlastností a metód a upravíme triedy tak aby dedili zo všetkých týchto tried. Jazyk C# tak aj Java je jazyk, v ktorom viac násobná dedičnosť nie je podporovaná. Riešením sú práve rozhrania, pretože trieda môže dediť z jednej nadradenej triedy a implementovať niekoľko rozhraní zároveň (Obr. 7.).

3.3.8.2 Postup

i. Vytvoríme prázdne rozhranie.
ii. Deklarujeme spoločné vlastnosti a metódy v rozhraní.
iii. Upravíme príslušné triedy tak aby implementovali nové rozhranie.
iv. Na miestach kde sa dané triedy používajú a používajú sa len vlastnosti a metódy nového rozhrania zmeníme typ premennej na rozhranie.
v. Kód preložíme a spustíme testy.

3.3.9 Stlačiť hierarchiu (*Collapse Hierarchy*)

Trieda predka a trieda potomka nie sú príliš odlišné. Jednoduchšie bude spojiť ich do jednej.



Obr. 8. Ukážka *Collapse Hierarchy*

3.3.9.1 Motivácia

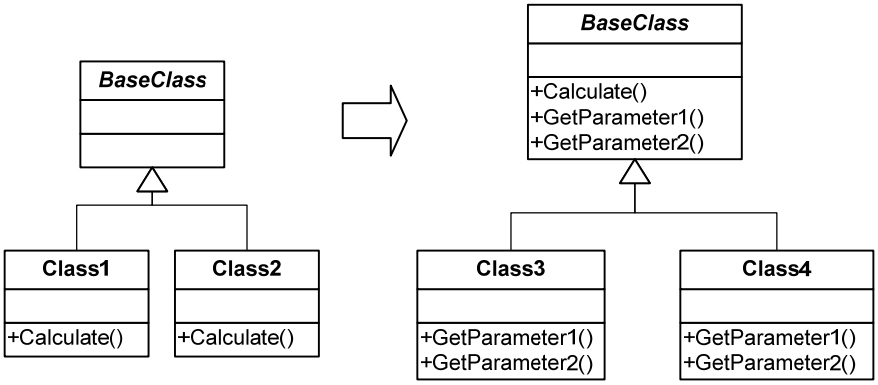
Ak je človek zvyknutý na objektovo orientovaný návrh, môže sa stať, že bude objektový model zbytočne zložitý. Refaktoring často presúva metódy alebo vlastnosti v hierarchii. Niekedy sa môžeme stretnúť s triedou, ktorá je potomkom inej ale nepridáva žiadnu alebo nie veľkú funkcionalitu. Vtedy je vhodné spojiť tieto triedy do jednej (Obr. 8.).

3.3.9.2 Postup

i. Vyberieme triedu, ktorá bude odstránená, či už je to trieda predka alebo potomka.
ii. Použijeme predchádzajúce refaktoringy (*Pull Up Field, Pull Up Method, Push Down Method, Push Down Field*) na presunutie funkcionality z jednej triedy do druhej.
iii. Po každom kroku kód preložíme a spustíme testy.
iv. Upravíme referencie odstraňovanej triedy na triedu druhú. Toto zahŕňa deklaráciu premenných, typy parametrov v metódach a konštruktoroch.
v. Odstránime prázdnu triedu.
vi. Preložíme a spustíme testy.

3.3.10 Šablónová metóda (*Form Template Method*)

Máme dve metódy v triedach s rovnakým predkom, ktoré majú podobný priebeh, vykonávajú podobné kroky v rovnakom poradí, ale v niektorých krokoch sa líšia.



Obr. 9. Ukážka *Form Template Method*

3.3.10.1 Motivácia

Ako sme si už povedali dedičnosť je mocný nástroj na eliminovanie duplicitného kódu. Kedykoľvek uvidíme dve metódy ktoré robia to isté hneď použijeme refaktoring *Pull up method*, ktorým metódu presunieme do nadradenej triedy a tým odstránime duplicitu. Čo ale v prípade, že metódy nie sú úplne identické? Stále potrebujeme odstrániť väčšinu duplicity ale tak aby sme zachovali rozdiely.

Častým prípadom bývajú metódy, ktoré obsahujú podobné kroky v rovnakom poradí, kroky sa však môžu líšiť. V takomto prípade toto poradie môžeme presunúť do nadradenej metódy a použitím polymorfizmu zabezpečíme to aby sa pre rôzne triedy volal rôzny kód (Obr. 9.). Takáto metóda sa nazýva *template method* [3].

3.3.10.2 Postup

i. Metódy rozložíme na menšie metódy tak aby boli buď identické alebo úplne odlišné.
ii. Použijeme *Pull up method* na presun identických metód do nadradenej triedy.
iii. Pre tie metódy, v ktorých sa triedy líšia použijeme *Rename method* aby sme zjednotili ich názov.
iv. Po každej zmene kód preložíme a spustíme testy.
v. Použijeme *Pull up method* na pôvodné metódy a vytvoríme *virtuálnu* hlavičku pre metódy, v ktorých sa triedy líšia (už majú zjednotené názvy) v nadradenej triede.
vi. Preložíme a spustíme testy.
vii. Odstránime ostatné metódy v podradených triedach, po každej zmene preložíme a spustíme testy.

4 Refaktoring a .NET

Praktické ukážky refaktoringu budú v nasledujúcej kapitole predvedené na aplikácii vytvorenej v prostredí .NET, presnejšie v jazyku C# a ASP.NET 2.0. Preto, ešte predtým ako sa dostanem k samotnej aplikácii a ukážkam refaktoringu, budem sa v tejto kapitole venovať platforme .NET. Na začiatku krátko platformu opíšem, spomeniem stručnú históriu, prípadne aké sú jej smerovania a plány do budúcnosti.

4.1 .NET Framework

4.1.1 História

V roku 1985 spoločnosť Microsoft predstavila na platforme IBM-PC operačný systém Windows, ktorý spôsobil revolúciu v používaní počítačov. Spolu s vznikom tohto operačného systému prišla na svet aj prvá verzia aplikačného rozhrania pre programovanie v prostredí Windows (Windows 1.0 API), ktorá umožňovala vytváranie aplikácii nad týmto operačným systémom. O osem rokov neskôr, v roku 1993, vyšla nová verzia operačného systému a to Windows NT. Tento systém obsahoval prvú verziu programového rozhrania, známeho ako Win32 API. Okrem podpory 32-bitového režimu a tisíce nových funkcií, Win32 API boli skoro totožné ako Windows 1.0 API. Programovanie na prvých systému Windows bolo väčšinou na systémovej úrovni a programované v jazyku C. V neskorých 90-tych rokoch sa už programovanie vo Windows delilo do niekoľkých kategórií a to, systémové, aplikačné a skriptovacie programovanie. Každá z týchto kategórií vyžadovala rôzne znalosti, využívali sa rôzne jazyky, nástroje a techniky.

Na systémové programovanie a vytváranie komplexných a robustných aplikácií sa využíval jazyk C, prípadne C++, ktoré priamo využívali Win32 API a prípadne technológiu COM (Component Object Model) na vytváranie a distribuovanie znovu použiteľných komponent. Správa pamäti bola v réžii programátora, čo znamenalo podrobnú znalosť fungovania systému Windows. Zásobníky, bity, byty, ukazovatele boli veci, ktoré programátor musel do podrobna poznať a vedieť využiť.

V prípade tvorby tradičných aplikácií mnohé softwarové firmy využívali miesto jazyku C a C++ jazyk Visual Basic. Tento jazyk mal vlastnú, jednoduchšiu syntax, vlastnú sadu funkcií (API) a výborné vývojové prostredie. Ďalej nekládol na programátora také vysoké nároky, programátor sa napríklad nemusel starať o správu pamäti, čo znamenalo, že programovať mohli aj obyčajní „smrteľníci“. Visual Basic dobre spolupracoval s technológiou COM, čo umožňovalo zdieľať kód medzi systémami a vývojármi.

Ak ste ako programátor chceli preniknúť do sveta webového programovania, potrebovali ste samozrejme ďalšie jazyky a nástroje. Používali sa jazyky ako VBScript, JScript spolu s novými

aplikačnými rozhraniami a jazykmi akými boli „klasické“ ASP (Active Server Pages). Popularita webového programovanie prudko stúpla v časoch internetového boomu v neskorých 90-tych rokoch, po ktorom nasledoval vznik XML a webových služieb. Stručne povedané oblasť programovania sa stala ešte viac rozdelenou.

Medzitým spoločnosť Sun Microsystems vyvíjala svoju Java platformu, ktorá zlučovala sadu spoločných nástrojov a technológií. Nezáležalo na tom, či ste programovali komponentu, ktorá mala byť zdieľaná medzi viacerými systémami alebo programátormi, klientskú alebo webovú aplikáciu, stále ste používali jeden spoločný jazyk (Java), knižnice (Java Development Kit, JDK), vývojové prostredie a nástroje. Bohatá sada knižníc a nástrojov sa časom stále a stále zväčšovala. V porovnaní v týmto bol vývoj aplikácií na platforme Windows oveľa zložitejší, čo spôsobilo to, že postupne viac a viac zákazníkov Microsoftu prechádzalo na konkurenčnú platformu, čo často krát znamenalo prechod na iný operačný systém (Linux). Toto si však spoločnosť Microsoft nemohla dovoliť, preto bolo jasné, že je nutné prísť s riešením. Týmto riešením sa stala platforma s názvom .NET Framework.

4.1.2 Popis platformy

Microsoft .NET Framework je softwarovou kompomentou, ktorá je súčasťou operačného systému Windows. Platforma zachováva spätnú kompatibilitu s Win32 a technológiou COM, priniesla nový programovací jazyk C#. Ten bol vytvorený z toho najlepšieho čo poskytovali súčasné najpoužívanejšie jazyky, ktorými boli C++, Visual Basic a Java. Platforma priniesla aj ďalšie nové jazyky, každý z nich bol postavený na rovnakom základe a tak mohol využívať všetky možnosti platformy. [4]

V súčasnosti (v čase písania tejto práce) je platforma .NET vo verzii 3.5. Na prvej verzii zacala spoločnosť Microsoft pracovať v neskorých 90-tych rokoch a prvú beta verziu .NET Frameworku 1.0 vydala v roku 2000. Nasledujúca tabuľka (Tab.1.) ukazuje chronologický zoznam verzií platformy .NET.

Verzia	Presné číslo verzie	Dátum vydania
1.0	1.0.3705.0	5.1.2002
1.1	1.1.4322.573	1.4.2003
2.0	2.0.50727.42	7.11.2005
3.0	3.0.4506.30	6.11.2006
3.5	3.5.21022.8	19.11.2007

Tab. 1. Verzie Visual Studia.NET

[4] http://en.wikipedia.org/wiki/Microsoft_.NET#Microsoft_.NET

4.1.3 Kľúčové znaky platformy

Interoperability

Pretože je častokrát žiadúca komunikácia medzi staršími a novšími aplikáciami, plaforma .NET ponúka možnosť prístupu k funkcionalite implementovanej v programoch mimo prostredia .NET. Prístup ku COM objektom poskytuje menný priestor *System.Runtime.InteropServices* a *System.EnterpriseServices* a prístup k iným funkcionalitám pomocou funkcie *P/Invoke.*

Common Language Runtime

Programovacie jazyky .NET Framerworku sa kompilujú do tzv. medzi jazyka (intermediate language), ktorý sa nazýva *Common Intermediate Language* alebo *CIL* (pôvodný názov bol *Microsoft Intermediate Language, MSIL*) . Tento jazyk nie je interpretovaný ale kompilovaný pomocou mechaniznu *just-in-time* kompilácie (presne včas) do natívneho kódu systému. Kombinácia týchto princípov sa nazýva *Common Language Infrastructure (CLI)*, ktorej implementácia spoločnosťou Microsoft sa nazýva *Common Language Runtime (CLR).*

Jazyková nezávislosť

.NET Framework prináša systém spoločných typov (CTS – *Common Type System*). CTS definuje všetky možné dátové typy, programovacie konštrukcie, ktoré podporuje CLR a aj to ako môžu spolupracovať. Vďaka tejto vlastnosti je možné na platforme .NET pracovať s inštanciami tried, ktoré sú naprogramované v ktorom koľvek .NET jazyku. Každý programátor tak môže používať svoj preferovanejší jazyk.

Knižnica základných tried (Base Class Library)

Knižnica základných tried (BCL) je súčasťou .NET Framworku, poskytuje funkcionalitu všetkým .NET jazykom. Ponúka triedy, ktoré zapúzdrujú množstvo často používaných funkcií, ako napríklad práca so súborovým systémom (čítanie alebo zápis do súboru), grafické vykresľovanie, prístup k databáze alebo manipulácia s XML dokumentom.

Jednoduchosť nasadzovania

Inštalácia software musí byť pozorne riadená aby napríkladnedošlo ku kolíziám s predchádzajúcimi verziami, čo môže zahŕňať rôzne bezpečnostné nároky. Aj v tejto oblasti .NET Framerwork poskytuje postupy a nástroje na uľahčenie práce.

Bezpečnosť

V oblasti bezpečnosti ponúka .NET Framework bezpečnostný model pre všetky aplikácie, ktoré na platforme bežia, rieši bežné zraniteľné a problematické miesta ako je napríklad chyba pretečenia zásobniku (buffer overflow), ktorá býva často zneužívaná škodlivým softwareom.

Prenositeľnosť

Návrh .NET Frameworku je v samej podstate platformovo nezávislý. To znamená, že program, ktorý je postavený nad .NET Frameworkom by mal bežať bez akejkoľvek zmeny na inom systéme, kde je .NET Framework implementovaný. Komerčné implementácie spoločnosti Microsoft zahŕňajú operačné systémy Windows, Windows CE a Xbox 360.

5 Nástroje a podpora refaktoringu

Refaktoring je veľmi užitočná technika, ktorou môžeme výrazne zvýšiť kvalitu už existujúceho kódu. Túto techniku je možné aplikovať vpodstate ručne bez akéhokoľvek špeciálneho nástroja a to vcelku jednoducho. Jediné čo potrebujeme je textový editor s možnosťou vyhľadávania textu. Tento postup je jednoduchý, vpostate rýchly, ale možný iba pri veľmi jednoduchých situáciach, v jednoduchých projektoch. Čím je kód komplexnejší a projekt väčší, tým je ručný refaktoring ťažší, namáhavejší a zdĺhavejší. Podpora refaktoringu bola vo vývojových prostrediach donedávna veľmi nízka a nástrojov určených na refaktoring bolo málo, čo bol aj hlavný dôvod, prečo nebol refaktoring ako užitočná technika medzi programátormi veľmi známa a používaná. Aj keď určitá podpora refaktoringu vo vývojových prostrediach bola, väčšina práce bola vykonávaná ručne. V tejto kapitole sa budem venovať práve nástrojom slúžiacim na automatizovaný refaktoring. V úvode spomeniem, prečo a veľmi stručne ako sa takýto nástroj používa, čo jeho používaním môžeme získať a aký do ma reálny dopad na vývoj software v praxi. Ďalej popíšem vlastnosti, ktoré by mal nástroj na refaktoring spĺňať. V ďalších podkapitolách stučne spomeniem aké nástroje sú v súčastnosti k dispozícii v rôznych programovacích jazykoch a vývojových prostrediach. Keďže táto práca používa ukážky v jazyku C#, budem sa v druhej polovici tejto kapitoly venovať nástrojom, podpore a možnostiam refaktoringu na platforme .NET.

Refaktoring pomocou nástroja na to určeného je niečo celkom iné ako refaktoring vykonávaný ručne. Aj keď máme pri vývoji k dispozícii testy (ak sa teda riadime zásadami správneho vývoja software), ktoré nám pri refaktoringu môžu pomôcť a slúžia nám obrazne povedané ako záchranná sieť, je ručný refaktoring veľmi časovo náročný. Práve aj pre tento dôvod programátori často refaktoring nerobia aj keď sami vedia, že by bol vhodný, zabralo by im to totiž príliš veľa času. Častokrát nastáva situácia keď si programátor pomyslí napríklad „asi by bolo vhodné zmeniť názov, ale...“ a následne si uvedomí čo všetko by musel skontrolovať, ktoré testy by musel spustiť, až daný kód nechá bez zmeny a povie si že ten názov nie je zase až taký zlý. Aj na zjednodušenie takýchto situácií boli a sú vytvárané nástroje na analýzu kódu a podporu automatizovaného refaktoringu, či sú už priamo integrované do vývojového prostredia, dodávané ako rozšírenia vývojoveho prostredia alebo ako samostatné aplikácie.

Ako taký nástroj vôbec funguje? Zoberme si ako príklad refaktoring *Extract method*, ktorý je v praxi jeden z najpoužívanejších. Keby sme tento refaktoring chceli použiť ručne, je tu niekoľko vecí, na ktoré musíme myslieť, čo môže byť zdĺhavé. Pri použití nástroja, typicky označime požadovaný blok kódu, klikneme na príslušnú položku menu určeného pre refaktoring, a nástroj spraví väčšinu práce za nás. V prvom kroku zistí, či je označený blok kódu validný na vytvorenie metódy. Označený blok kódu totiž nemusí obsahovať všetky potrebné informácie, ktoré sú nutné na vytvorenie novej metódy, môže obsahovať priradenie hodnoty nejakej premennej bez toho aby

obsahoval všetky jej výskyty, atď. O tieto veci sa však starať nemusíme, spraví to za nás nástroj. Po úspešnej validácii nástroj vypočíta koľko a ktoré parametre majú byť posielané novej metóde ako vstup, potom je užívateľ typicky vyzvaný na to aby zadal názov novej metódy. Následne je vrámci aktuálnej triedy vytvorená nová metóda a označený blok kódu je nahradený jej volaním s príslušnými parametrami. Po zvyknutí si na konkrétny nástroj, takýto proces trvá len niekoľko sekúnd, keď to porovnáme s postupom (popísaný v 3. kapitole), je to výrazná úspora času.

Tým, že sa za pomoci nástrojov stáva refaktoring menej časovo náročný, dizajnové chyby pri tvorbe software nemajú také následky ako mali kedysi. Nástroje nám teda neslužia len na to aby nám urýchlili vykonanie konkrétneho refaktoringu a ušetrili nám množstvo manuálnej práce ale takisto spôsobujú to, že sa celý vývoj software urýchli. Ako je to možné? Tým, že je menej časovo náročné dodatočne opraviť dizajnové chyby, nie je nutné mať na začiatku vývoja príliš podrobný a „dokonalý“ návrh. V praxi toho často ani možné nie je, návrh sa postupne mení a prispôsobuje sa požiadavkám, ktoré na začiatku vývoja väččinou nie sú úplné. V minulosti sa muselo pri návrhu myslieť dopredu a skoro by som povedal predvídať budúcnosť, aby bolo možné určiť kam a ako sa bude za nejaký čas vývoj uberať. Pretože ak sa už raz spravil návrh, nebolo ho skoro možné následne upraviť podľa nových požiadaviek, ktoré prichádzali od zákazníkov, programátori museli „bojovať“ aby novú funkcionalitu nejakým spôsobom zapracovali do súčasného návrhu aj keď vedeli, že návrh už nezodpovedá požiadavkám. Zmena by bola totiž príliš časovo náročná ak vôbec možná. Dnes, s nástrojmi umožňujúcimi automatizovaný refaktoring je možné navrhovať software podľa aktuálnych potrieb. Postupne po čase, keď sa požiadavky menia alebo pribúdaju, návrh sa môže stať nevhodným a kód ťažšie čitatelným, vtedy je možné za pomoci nástrojov návrh a kód upraviť podľa potrieb, čo nie je taký problém ako bol v minulosti.

Automatizovaný refaktoring pomocou nástroja do určitej miery ovplyvňuje aj testovanie. Keďže je refaktoring vykonávaný automaticky, nie je nutné spúšťať testy po každej zmene kódu. Vždy ale však budú existovať refaktoringy a situácie, ktoré sú príliš komplexné na to aby boli plne automatické, takže testovaniu kódu po jeho úprave sa asi nikdy nevyhneme.

Zatiaľ som hovoril o nástrojoch v teoretickej rovine, aby som vysvetlil v čom je ich prínos a prečo ich používať. V nasledúcich dvoch podkapitolách stručne popíšem aké by mali sú požiadavky na takýto nástroj a to z dvoch pohľadov, technického a praktického.

5.1 Technické požiadavky na nástroj

Hlavný dôvod, kvôli ktorému sú vytvárané nástroje pre refaktoring je to, aby umožnili programátorovi rafaktorovať kód automaticky a bez nutnosti zmenený kód otestovať aby sa zamedzilo zaneseniu chyby. Testovanie po každej zmene kódu môže byť veľmi časovo náročné aj keď je to zväčša do veľkej miery automatizovaný proces, jeho odstránenie však výrazne zvýši

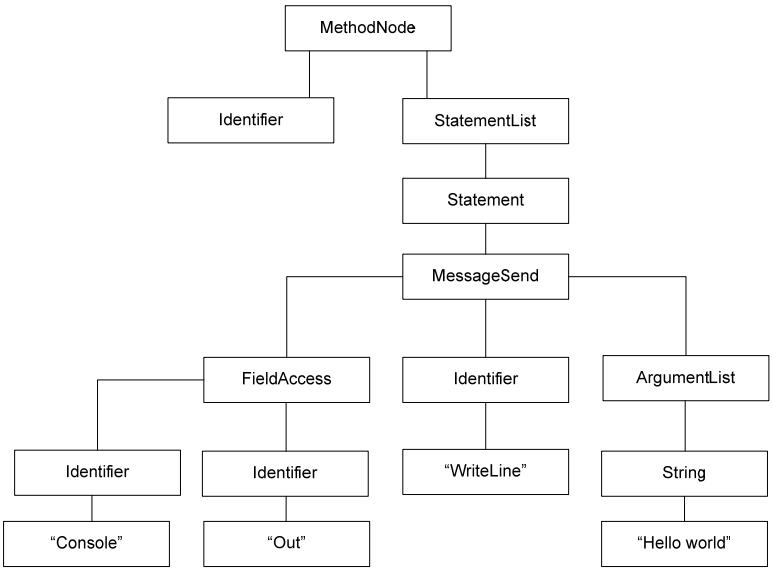
efektivitu práce. V tejto kapitole stručne popíšem technické požiadavky na nástroj, aby ním bolo možné automatizovane upravovať kód bez zmeny jeho správania.

Jednou z prvých a najdôležitejších vlastností, ktoré musí takýto nástroj mať, je schopnosť vyhľadávať rôzne programové entity v celom projekte. Programovou entitou myslím napríklad metódu, triedu, premennú, rozhranie, konštantu atď. Nástroj teda musí byť schopný napríklad vyhľadať všetky volania určitej metódy alebo vyhľadať referencie na nejakú premennú, miesta kde sa z danej premennej číta hodnota a miesta kde sa danej premennej nastavuje hodnota. Tento proces však nemôže byť len jednoduché hľadanie reťazca (napríklad názov metódy) a následné nahradenie iným reťazcom, nebolo by totiž možné rozlíšiť názov triedy od názvu metódy, rešpektovať menné priestory alebo viditeľnosť danej entity (prístupové modifikátory). Je nutná analýza kódu a informácie o entitách udržovať v nejakej štruktúre podobnej databáze. V prostredí Smalltalku sú tieto informácie udržované konštatne a sú k dispozícii vo forme, ktorá sa dá pohodlne prehľadávať. Programátor môže požadovanú entitu vyhľadať a vykonať zmenu. Táto zmena je automaticky preložená to tzv. *bytekódu* a uložená do „databáze“. V statickejších prostediach ako je napríklad prostedie .NET Frameworku alebo Javy, sú tieto takéto informácie k dispozícii až po preklade kódu. Na vytvorenie takejto „databáze“ je nutná sémantická analýza kódu. Táto analýza musí byť vykonaná na úrovni definície tried na určenie triednych premenných a definícií metód, na úrovni metód na určenie premenných a metód, ktoré sú vrámci analyzovej metódy volané. [5]

Väčšina rafaktoringov manipulujú časťami systému na úrovni metód, sú to väčšinou referencie na programové entity, ktoré sú zmenené. Príkladom môže byť premenovanie premennej, jednoduchá definičná zmena. Všetky referencie vrámci danej metódy v triede kde je metóda deklarovaná a vo všetkých triedach ktoré su z tejto triedy odvodené musia byť upravené. Iné refaktoringy manipulujú ešte pod touto úrovňou. Príkladom je vytvorenie novej metódy z existujúceho bloku kódu (*Extract Method*) metódy. Aby toto bolo možné spraviť, je potrebný tzv. *parse tree.* Ide o dátovú štruktúru, ktorá reprezentuje vnútornú štruktúru časti kódu, napríklad tela metódy.

```
public void HelloWorld()
{
      Console.Out.WriteLine(“Hello world”);
}
```

[5] Fowler, M.: Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.



Obr. 10. Ukážka kódu a odpovedajúceho *parse tree*

Refaktoring vykonaný pomocou nástroja musí čo najviac zachovať vonkajšie správanie sa systému. Nie je totiž možné aby aby sa systém po refaktoringu správal identicky ako pred zmenou. Môže sa stať to, že po zmene bude kód o niečo pomalší ako predtým, môžu to byť len desiatky milisekúnd, čo väčšinou nemá na systém žiadny vplyv. Existujú však situácie alebo typy software kde by takáto zmena znamenala nekorektné správanie alebo spôsobovala chyby. Ďalším problémom môže byť používanie reflection. Zoberme si príklad keď pomocou nástroja koretne premenujeme metódu, program je preložiteľný, cez to všetko sa nespráva tak ako pred zmenou. Pri spustení testov zistíme, že v nejakých prípadoch dochádza k výnimkam. Je to preto, že pri použití reflection môžeme volať metódy použitím metódy *Invoke*, ktorej ako parameter pošleme reťazec, ktorý reprezentuje názov volanej metódy. Túto situáciu však nástroj nie je schopný odhaliť.

Vo väčšine prípadov je však možné pomocou nástrojov aplikovať refaktoring bez narušenia správania sa systému, musíme mať však identifikované potenciálne rizikové miesta, o ktorých vieme, ze musia byť upravené ručne po aplikovaní refaktoringu.

5.2 Praktické požiadavky na nástroj

Nástroje na podporu refaktoringu majú predovšetkým slúžiť ľudom, programátorom a uľahčiť ich prácu. Nástroj môže byť akokoľvek mocný a plný rôznych užitočných funkcií, ak však nerešpektuje spôsob akým programátori pracujú nebude používaný. Preto jedna z najdôležitejších vlastností takéhoto nástroja je aby bol prakticky použiteľný spolu s ostatnými nástrojmi využívanými pri vývoji software.

Analýza a transformácie kódu potrebné na vykonanie refaktoringu môže byť časovo náročný proces. Pokiaľ vykonanie automatického refaktoringu bude trvať príliš dlho, programátor ho buď

používať nebude alebo refaktoring spraví ručne a bude čeliť následkom. Preto je pri vytváraní takéhoto nástroja nutné vždy brať do úvahy aj jeho rýchlosť, nielen funkčnosť.

Jednu z ďalších vecí, ktorú nám dovoľuje používanie nástroja na refaktoring je vytvoriť návrh určitým spôsobom, naprogramovať týmto spôsobom kus kódu aby sme videli ako to vyzerá a následne prvotný návrh zmeniť v prípade, že nájdeme lepšie riešenie. Keďže proces refaktoringu zachováva spravánie software, takže proces ktorým práve vykonaný refaktoring stornujeme je tiež proces zachovávajúci správanie systému a môžeme ho považovať za určitý druh refaktoringu a mal by byť v nástroji implementovaný. Je dôležité aby sme po zmene po refaktoringu boli schopný vrátiť sa späť a to bez ohľadu na to aká veľká zmena bola. V niektorých nástrojoch v nedávnej minulosti toto nebolo možné, preto si programátori museli pre istotu kód pred refaktoringom zazálohovať, vykonať refaktoring a v prípade, že výsledok nebol taký ako očakávali, museli kód obnoviť zo zálohy, čo bolo veľmi nepohodlné a nepraktické. Dnes je už situácia iná, nástroje poskytuju veľmi rýchlu odozvu, čo nám umožňuje rýchlu zmenu kódu.

Ďalšia vec ktorá je veľmi dôležitá je, aby bol nástroj priamo integrovaný do vývojového prostredia (IDE – Integrated Development Enviroment), takéto prostredie typicky ponúka takmer všetko čo k vývoju software programátor potrebuje. Typicky obsahuje editor zdrojového kódu, kompilátor, linker, nástroj na ladenie chýb a mnoho ďalších. Preto vytvorenie nástroja pre refaktoring ako samostatnej aplikácie spôsobí to, že ju skoro nikto nebude používať. Akonáhle však bude „na dosah“ priamo vo vývojovom prostredí bude používaný veľmi často.

Nástroje na podporu refaktoringu sú jedna z najepších ciest ako udržovať kód, ktorý sa stáva čím ďalej tým viac zložitý a neprehľadný postupne ako sa projekt zväčšuje a kód.

5.3 Dostupné nástroje

V prvej časti tejto kapitoly stručne popíšem aké sú možnosti nástrojov na refaktoring v rôznych vývojových prostrediach , ktoré sú v súčasnosti používané. Keďže je práca v praktických ukážkach zameraná na platformu .NET, v druhej časti sa budem podrobnejšie zaoberať nástrojmi, ktoré sú dostupné práve pre túto platformu.

5.3.1.1 Smalltalk

Smalltalk Refactoring Browser – prvý a pôvodný nástroj na automatizovaný refaktoring. Dodnes patrí medzi nástroje s najširšou paletou funkcií.

5.3.1.2 Java

IntelliJ Idea – integrované vývojové prostredie plné funkcií s integrovanou podporou refaktoringu.
Eclipse – open source vývojové prostredie so silnou integrovanou podporou refaktoringu.

JBuilder – vývojový nástroj firmy Borland, ktorý ma nejakú podporu refaktoringu, existuje do neho však niekoľko rozšírení.
JFactor – rozšírenie pre JBuilder a Visual Age na podporu refaktoringu.
JRefactory – ďalšie rozšírenie pre JBuilder na podporu refaktoringu.

5.3.1.3 .NET

Visual Studio.NET – primárny nástroj na vývoj aplikácii pre platformu .NET, má priamo zabudovanú podporu refaktoringu. Podporovaných refaktoringov je však málo a niektoré z nich sú veľmi pomalé. Našťastie existujú rozšírenia tohto nástroja, ktoré tento nedostatok napravujú.
ReSharper – veľmi mocné rozšírenie vývojového nástroja Visual Studio.NET, obsahuje veľkú sadu podporovaných refaktoringov. Až s použitím tohto rozšírenia sa dá Visual Studio.NET považovať za plnohodnotný vývojový nástroj.
C# Refactory – ďalšie rozšírenie Visual Studia.NET na podporu refaktoringu.

5.3.1.4 C/C++

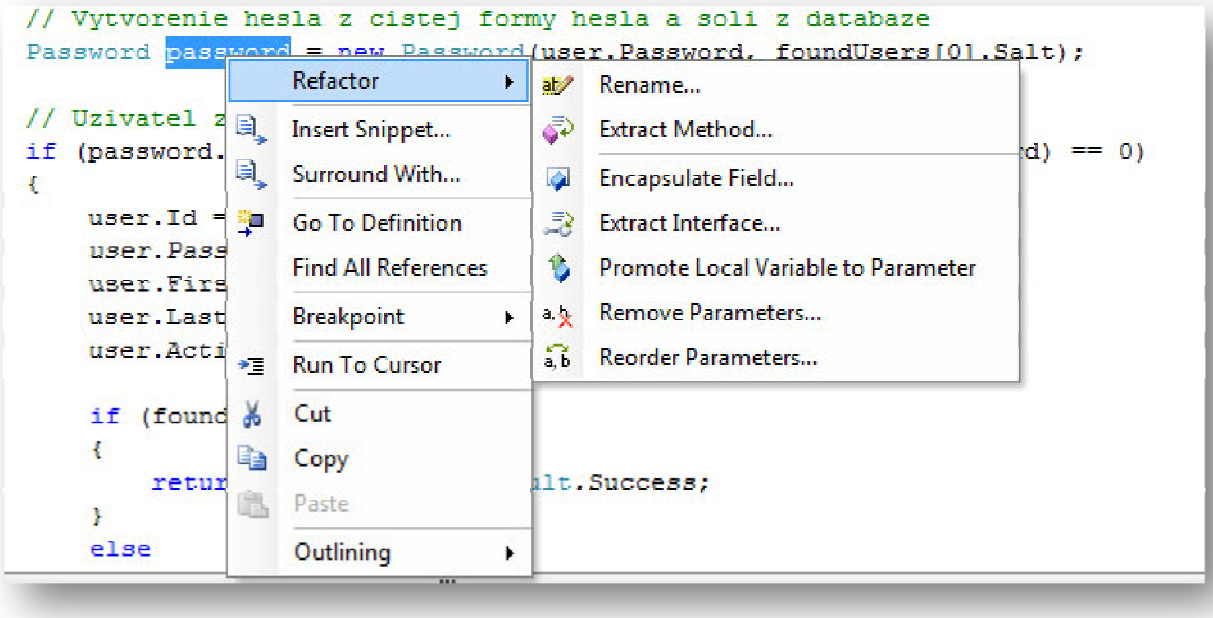
SlickEdit – nástroj na vývoj aplikácií v jazyku C/C++. Vo svojej poslednej verzii už obsahuje podporu refaktoringu.
Ref++ - rozšírenie Visual Studia.NET, ktoré poskytuje podporu refaktoringu aj pre jazyky C a C++.

5.3.2 Nástroje pre platformu .NET

V tejto časti sa budem podrobnejšie venovať nástrojom na refaktoring dostupným pre programovanie na platforme .NET. Najpoužívanejším nástrojom na vývoj aplikácii pre platformu .NET je Visual Studio.NET od spoločnosti Microsoft. Tento nástroj je k dispozícii v niekoľkých edíciách, zadarmo hlavne k študijným a predvádzacím učelom je k dispozícii v edídcii *Express*, komerčná edícia *Profesional* je už určená pre profesionálne použitie vo firmách a je teda platená. Najvyššia verzia sa nazýva *Team Edition*, ktorá ponúka kompletné riešenie pre vývojové tými. Od reportovania chýb, cez nástroje pre databázových špecialistov, ponúka aj množstvo štatistík, reportov a orgranizačných záležitostí.

5.3.2.1 Visual Studio .NET

Priamo v prostredí Visual Studia .NET existuje podpora refaktoringu. Pri editácii zdrojového kódu po kliknutí pravého tlačítka myši sa zobrazí kontextové menu, ktoré obsahuje položku „*Refactor*“, ktoré je k dispozícii aj z hlavného menu aplikácie. Toto menu nie je nijak zložité a ani neobsahuje velký počet položiek. Nachádza sa na ňom sedem možností refaktoringu.



Obr. 11. Ukážka menu na refaktoring vo Visual Studiu .NET

Funkcia *Rename* slúži na premenovanie nejakej programovej entity, môže byť použitá na premenovanie triedy, premennej, metódy, rozhrania, atď. Pri zvolení tejto možnosti je uživateľovi umožnené zvoliť nový názov a niekoľko ďalších možností ako je prehladávanie aj v textových reťazcoch, komentároch a náhľad zmeny pred jej vykonaním. Po potvrdení dialógu sa spustí preklad a analýza kódu aby bolo možné nájsť miesta ked všade treba zmenu aplikovať, celý proces trvá len niekoľko sekúnd. Možnosť tohto refaktoringu sa ponúka automaticky pri priemenovaní akéhokoľvej entity. Stačí zmeniť názov triedy, ten sa označí a pri kliknutí na ikonku je uživateľovi ponúknutá možnosť premenovania.

Druhou v poradí je funkcia *Extract method*, slúži na vykonanie jedného z najpoužívanejšieho refaktoringu, ktorý je popísany v kapitole 3. Postup je jednoduchý, uživateľ označí blok kódu, z ktorého chce vytvoriť samostanú metódu a klikne na túto položku v menu. Uživateľ zadá názov metódy, a potvrdí dialóg, systém rozpozná, či je novej metóde potrebné posielať parametre a na základe toho vytvorí definíciu metódy, vytvorí ju a nahradí o značený blok kódu jej volaním.

Ďalšou funkciou je *Encapsulate Field*, ktorá slúži a vytvorenie verejnej vlastnosti z privátnej premennej. Dobrým zvykom je pristupovať priamo k triednym privátnym premenným len v konstruktoroch a v ostatných metódach k nim pristupovať cez vlastnosti. Na ich vytvorenie slúži práve táto funkcia.

Štvrtou a poďla môjho názoru veľmi užitočnou funkciou je *Extract Interface*, pomocou tejto funkcie je možne z existujúcej triedy vytvoriť rozhranie. Keďže v prostredí .NET nie je možná viacnásobná dedičnosť, dosť často nastávajú situácie, keď potrebujeme aby mala skupina tried spoločné vlastnosti a metódy. Riešenie tohto problému sú rozhrania. Keď máme už hotovú triedu,

ktorá má implementované metódy a vlastnosti, ktoré potrebujeme aj v inej triedy, použijeme funkciu *Extract interface*, Visual Studio nám navrhne názov rozhrania, ktoré samozrejme môžeme zmeniť a vyberieme, ktoré verejné vlastnosti a metódy sa pri vytváraní rozhrania majú použiť. Po potvrdení dialógu Visual Studio vytvorí nový súbor a vloží do neho novo vytvorené rozhranie, ktoré automaticky implementuje aj trieda, z ktorej bolo rozhranie vytvorené.

Ďalšou funkciou je *Promote Local Variable to Parameter,* ktorá slúži na vytvorenie parametra metódy z lokalnej premennej použitej v metóde. Funkcia odstráni deklaráciu lokálnej premennej a upraví definíciu metódy.

Funkcia *Remove Parameters* slúži na odstránenie parametra z metódy, konstruktora alebo indexera, po potvrdení príslušného dialógu sa upraví definícia metódy a všetky miesta kde je metóda volaná. Dôvody a motivácia k tomuto refaktoringu sú popísane katalódu refaktoringov v podkapitole 3.2.3.

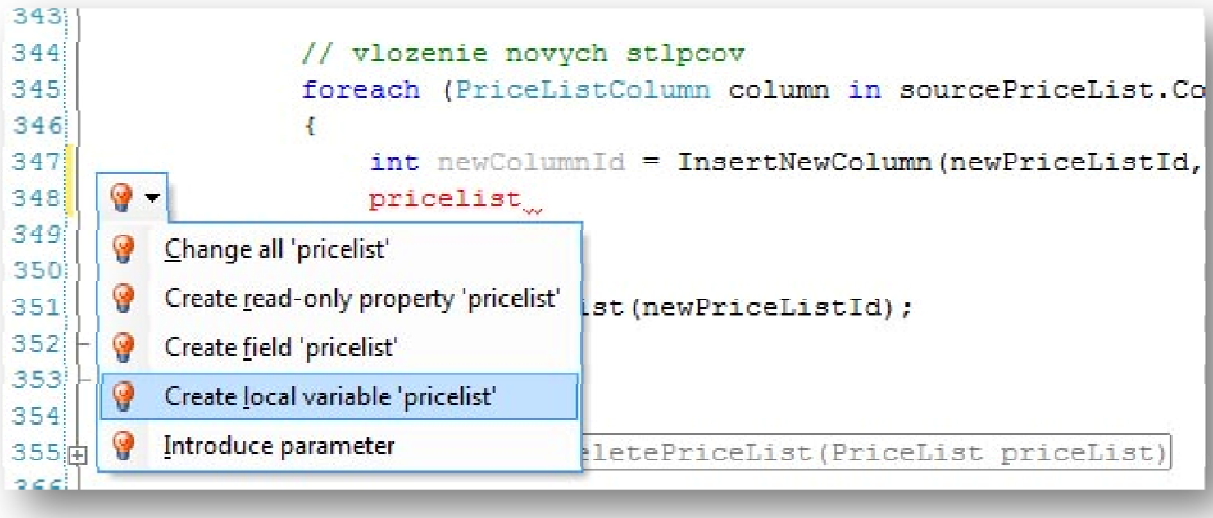
Zmena poradia parametrov metódy robená ručne skoro vždy vedie ku chybám, niektoré z nich odhalí prekladač, iné však nie, ide o situácie keď sú parametre rovnakého typu. Pri rozsiahlejšom projekte, robiť takúto zmenu ručne je skoro nemožné. Práve preto je vo Visual Studiu funkcia *Reorder Parameters*, ktorá tento problém plne automatizovane rieši za nás. Stačí vybrať túto funkciu, potom na dialógu usporiadať parametre a potvrdiť všetko ostatné je v réžii Visual Studia.

Musím zkonštatovať, že vzhľadom na to, že je Visual Studio.NET profesionálny komerčný nástroj na tvorbu software a je veľmi málo nástrojov, ktoré mu môžu konkurovať, jeho podpora refaktoringu je pomerne chudobná. Podpora siedmich funkcií je naozaj nedostačujúca aj keď uznávam, že sa z veľkej väčšiny v praxi viac funkcií používať nebude.

Existujú však nástroje tretích strán (väčšinou komerčné a teda platené), ktoré je možné do Visual Studia.NET doinštalovať ako rozšírenia a tak podporu refaktoringu značne rozšíriť. Spolu s týmito rozšíreniami je Visual Studio.NET aj v oblasti podpory refaktoringu nástroj, ktorému je veľmi ťažké konkurovať. V nasledujúcej časti sa budem zaoberať práve týmito rozšíreniami.

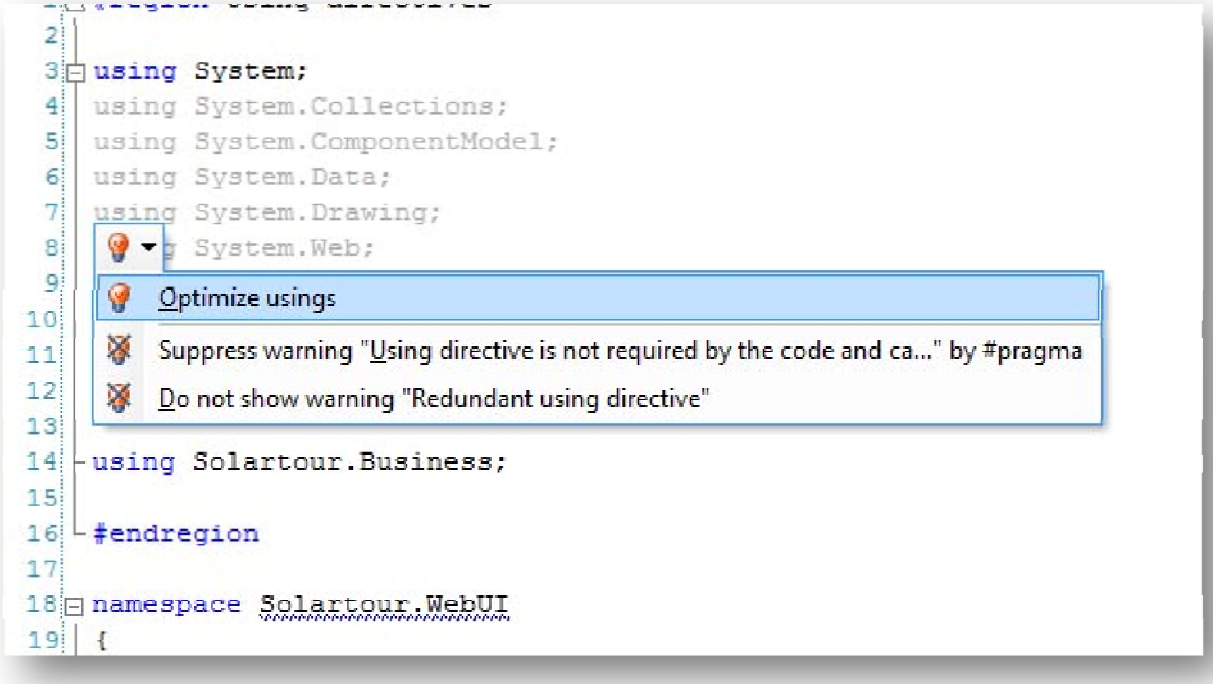
5.3.2.2 ReSharper

ReSharper od firmy JetBrains je komerčným a teda plateným rozšírením Visual Studia.NET, existuje však možnosť vyskúšať si ho na 30 dní zadarmo. Túto možnosť som využil ja na predstavenie tohto nástroja. K dispozícii je pre Visual Studio.NET 2005 a 2008. Nástroj neponúka iba vylepšenú podporu refaktoringu. Ponúka analýzu kódu, už počas jeho písania a označuje chyby bez nutnosti kompilácie a hned ponúkne aj návrh ako danú chybu odstrániť, čo výrazne zvyšuje efektivitu práce.



Obr. 12. Ukážka analýzy kódu

Ďalšou funkciou na zvýšenie rýchlosti pri práci je aj vylepšený *intelisense* (funkcia inteligentného dopĺňovania písaného kódu). Ktorá napríklad zahŕňa automatické uzatváranie zložených zátvoriek, návrhy ktoré menné priestory treba uviesť medzi *using* direktívy, pomoc pri formárovaní kódu a mnohé ďalšie vylepšenia.

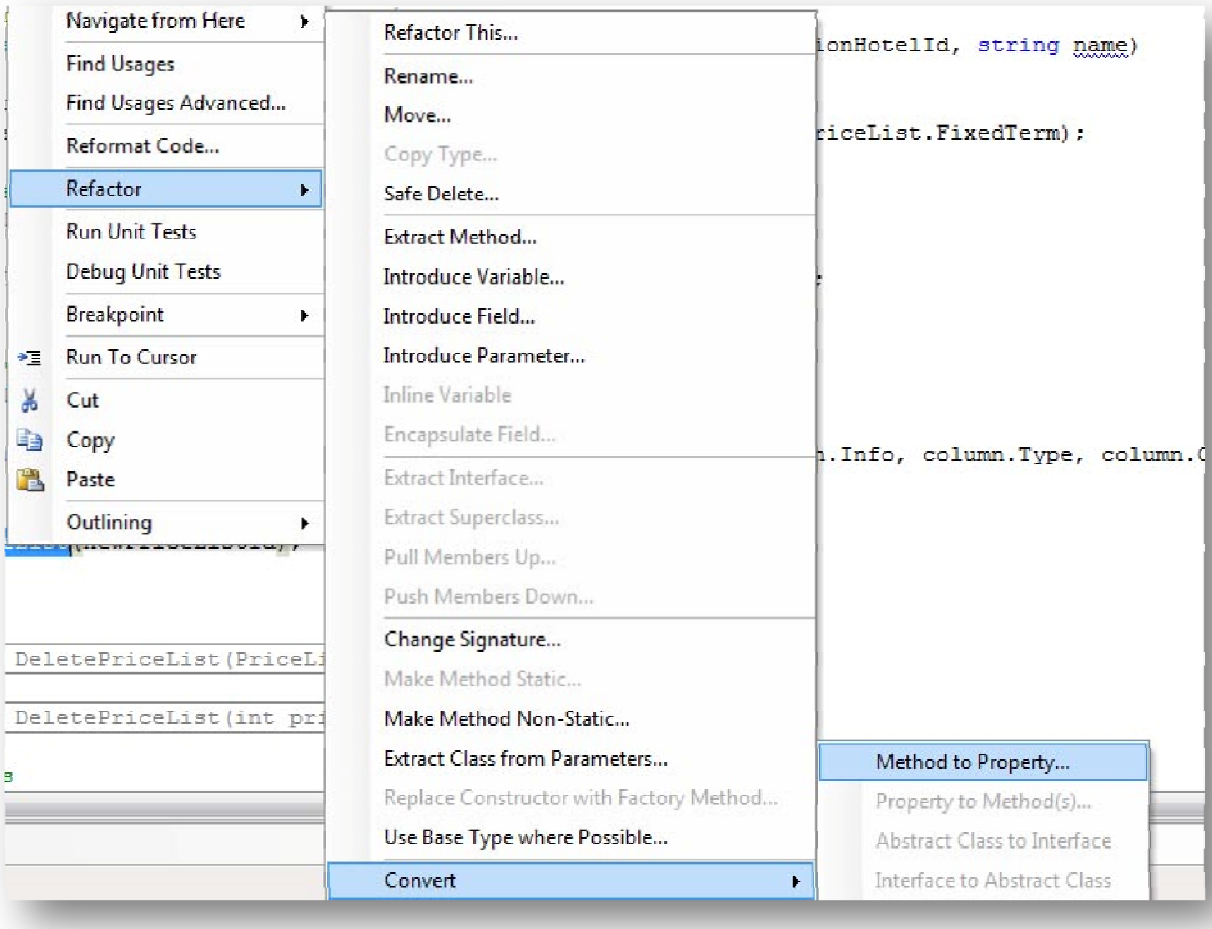


Obr. 13. Ďalšia ukážka použitia ReSharpera

Taktiež ponúka podporu vytvárania *unit* testov, ktoré sú pri vývoji software a refaktoringu veľmi dôležité. Automaticky detekuje testy vytvorené pomocou frameworku *NUnit,* ktoré môžeme spúšťať priamo pri editovaní kódu z kontextového menu. Poskytuje aj „prieskumníka“ *unit* testov, na prehľadnejšie zobrazovanie a organizáciu jednotlivých testov. Aj v tejto oblasti rozširuje Visual Studio.NET do značnej miery a robí z neho veľmi silný nástroj.

Skrátka tento nástroj je obsahuje množtvo veľmi užitočných funkcií, ktoré nielen uľahčujú prácu, zvyšujú efektivitu ale aj výrazne zpríjemňujú prácu pri vývoji software. Funkcie, ktoré som už spomenul a mnohé ďalšie, na ktoré nemám v rozsahu práce priestor sú dôležité a užitočné, nás však ale zaujíma to, čo tento nástroj ponúka v oblasti refaktoringu.

Resharper poskytuje podporu 28 jednotlivých refaktoringov, ktoré nám dovoľujú premenovanie, premiestňovanie a bezpečné mazanie programových symbolov a mnohé ďalšie. Len pre porovnanie resharper ponúka o 21 refaktoringov viac ako podporuje priamo Visual Studio.NET. Použitie je veľmi podobné použitiu refaktoringu, ktoré ponúka Visual Studio.NET. Blok kódu, ktorý chceme refaktorovať označíme a použijeme menu *Refactor* (či už kontextové alebo v hlavnom menu aplikácie).



Obr. 14. Rozšírené menu *refactor*

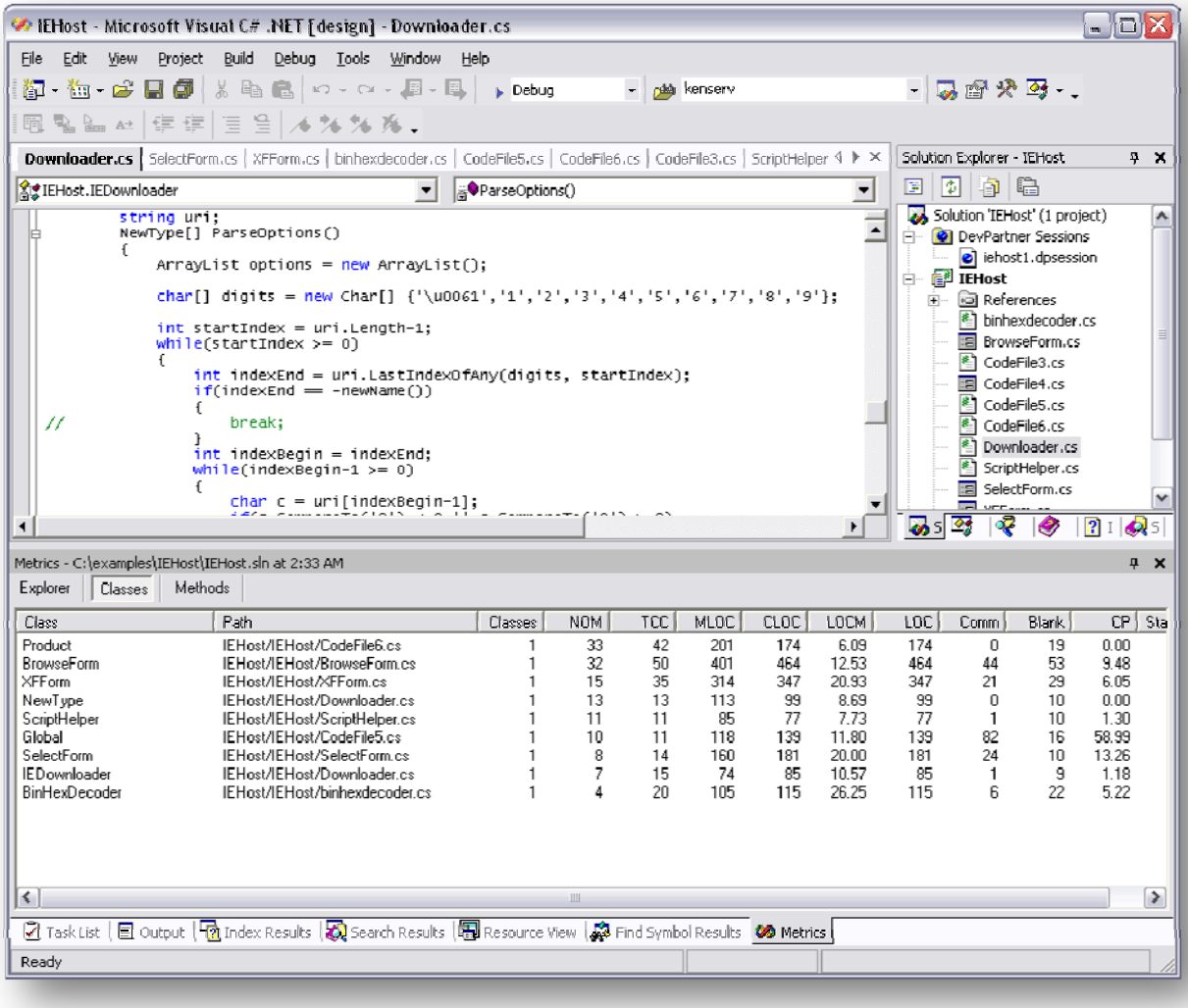
Z obrázka môžeme vidieť, že rozšírené menu obsahuje podstatne viac položiek a čo je dôležitejšie, aktívne sú len položky, ktoré majú v danom kontexte zmysel. Uľahčuje a zprehľaďňuje to prácu. Po vybratí nejakej z možností je postup takmer identický ako pri použití zabudovanej podpory refaktoringu. Zobrazí sa dialóg s možnosťami a po potvrdení je príslušný refaktoring vykonaný. Proces je veľmi intuitívny, preto sa ním nebudem podrobnejšie zaoberať.

Nástroj ReSharper od firmy Jetbrains považujem za jedno z najlepších rozšírení už aj tak výborného vývojového nástroja Visual Studia.NET. ReSharper hodnotím veľmi kladne a každému kto využíva Visual Studio.NET ho minimálne na odskúšanie doporučujem.

5.3.2.3 C# Refactory

C# Refactory je ďalším rozšírením Visual Studia.NET, ktoré podporuje refaktoring. Ako už názov napovedá ide len o podporu pre jazyk C#, narozdiel od jazykovo nezávislého ReSharpera opísaného vyššie. Tento nástroj tak ako aj ReSharper a priamo Visual Studio.NET poskytuje sadu refaktoringov. Ide presne o 13 refaktoringov (*Extract Method, Change method signature, Decompose conditional, Extract variable, Extract superclass, Extract interface, Copy class, Push up members, Rename type, Rename member, Rename parameter, Rename local variable, Encapsulate filed*), čo je síce viac ako poskytuje Visual Studio.NET ale menej ako konkurenčný ReSharper. Okrem dobre známych refaktoringov, ktoré premenujú programový element alebo ho presunú ponúka C# Refactory jeden zaujímavý a ním je *Decompose conditional*. Ktorý slúži na rozloženie zložitej podmienky na menšie a jednoduchšie časti.

Čo však tento produkt ponúka navyše oproti ReSharperu sú metriky. Tie môžu byť pre programátora alebo dokonca aj pre manažéra veľmi cenné. Či už z hľadiska merania pokroku alebo stupňa udržovateľnosti kódu.



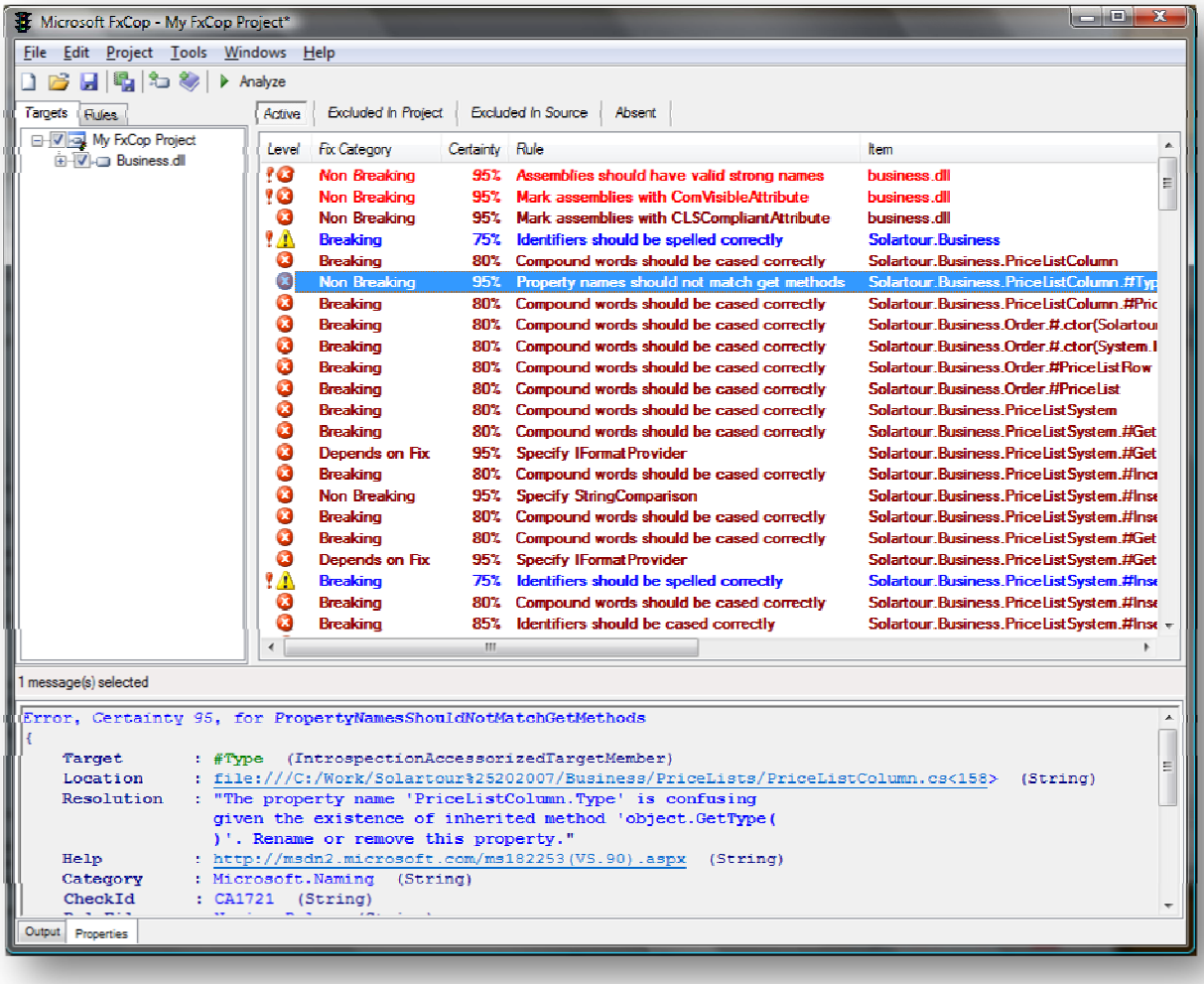
Obr. 15 Metriky merané pomocou C# Refactory – zdroj *http://www.xtreme-simplicity.net/*

Medzi jednoduché metriky patria napríklad počeť riadkov alebo počet príkazov. Metriky tohto typu nám môžu niečo povedať o veľkosti projektu, či už z pohľadu celého projektu, triedy alebo metódy. Sledovanie cyklomatickej zložitosti (*cyclomatic complexity* – ide o softwarovú metriku, ktorá bola vytvorená Thomasom McCabeom a slúži na meranie zložitosti programu, bližšie si o tejto metrike povieme neskôr) častí programu nám pomôže odhaliť oblasti, ktoré sú príliš zložité a teda náročné na údržbu. Takéto oblasti by sme mali práve za pomoci refaktoringu zjednodušiť. Funckiu takýchto metrík poskytuje už aj Visual Studio.NET vo svojej najnovšej verzii 2008 v Team Edition a podľa môjho názoru prehľadnejšie ako pomocou C# Refactory.

Nástroj určite rozširuje možnosti Visual Studia.NET v oblasti refaktoringu, podporuje však len jazyk C# a neposkytuje vylepšenia ani takú množinu refaktoringov ako konkurenčný ReSharper, nie je teda ničím výnimočným.

5.3.2.4 Microsoft FxCop

FxCop je aplikácia na analýzu kódu, ktorý je preložený do .NET Framework Common Language Runtime. To znamená, že dokáže analyzovať už existujúci preložený kód v assemblies (napríklad DLL knižnica). Tento nástroj nie je nástrojom, ktorý primárne podporuje refaktoring. Ako som už povedal, slúži na anlýzu kódu. Nástroj má nadefinovanú sadu pravidiel s rôznou dôležitosťou. Táto sada je nadefinovaná spoločnosťou Microsoft podľa ich vlastných pravidiel a zásad vývoja software. Je samozrejme možné túto sadu meniť, zakazovať niektoré pravidlá alebo pridávať vlastné, čím si každá firma môže zabezpečiť dodržovanie firemných metodík. Je potom jednoduchšie vykonávať revíziu kódu a kontrolovať dodržovanie dobrých zásad a praktík pri vývoji. Použitie tohto nástroja neodhaľuje chyby, skôr upozorňuje na miesta kde kód nie je čistý alebo porušuje definované pravidlá, môže ísť o názvoslovie, pravidlá ohľadom dedenia, viditeľnosti vlastností triedy atď. [6]



Obr. 16. Ukážka aplikácie FxCop

[6] Cwalina, K., B. Adams: Framework Design Guidelines, Addison-Wesley, 2005.

Jedným dôvodom prečo tento nástroj používať, ktorý som už spomenul je dodržovanie metodík vývoja. Ako nám však môže tento nástroj pomocť pri refaktoringu? Pri analýze kódu týmto nástrojom môžeme získať pohľad na vyvíjaný software z trochu iné uhlu. Nástroj popri označení problémového miesta ponúka aj krátky text, ktorý problém vysvetľuje a popisuje ako daný kód opraviť. Takéto opravy nám často pomôžu zlepšiť návrh, zvýšiť výkon alebo bezpečnosť systému. Zároveň nám to pomôže pri rozhodovaní, na ktoré časti sa pri refaktoringu máme zamerať alebo aspoň kde začať.

FxCop je distribuovaný ako externá exe aplikácia s grafickým užívateľským rozhraním alebo ako konzolová aplikácia, ktorá sa dá pripojiť priamo do Visual Studia .NET. Vy vyšších edíciach Visual Studia .NET je už analýza tohto typu priamo zabudovaná, takže nie je nutné používať externú aplikáciu [7].

[7] Cwalina, K., B. Adams: Framework Design Guidelines, Addison-Wesley, 2005.

6 Praktické využitie refaktoringu

Nasledujúca časť práce bude venovaná praktickým ukážkam refaktoringu na reálnej aplikácii. Pre demonštráciu som si vybral vlastný projekt, ktorý som naprogramoval pre niekoľkými rokmi za účelom naučiť sa programovať v prostredí .NET. V stručnosti ide o malý a relatívne jednoduchý redakčný systém spolu so správou cenníkov, ktorý je v súčasnosti použitý na správu obsahu webovej stránky jednej cestovnej kancelárie. Kompletné zdrojové texty aplikácie ako pred refaktoringom, tak aj po ňom sú k dispozícii na priloženom CD (Príloha 1). V úvode aplikáciu podrobnejšie predstavím, popíšem akým spôsobom bola vytvorená, popíšem dátový model, zodpovedajúci objektový model, ako je rozdelený kód atď. Po predstavení aplikácie budem na konkrétnych príkladoch v aplikácii ukazovať praktické využitie refaktoringu a čo presne refaktoringom získame, čo sa zmenilo, prípadne zlepšilo a ak to bude možné tak aj aký má daná zmena dopad na výkon aplikácie.

6.1 Popis aplikácie

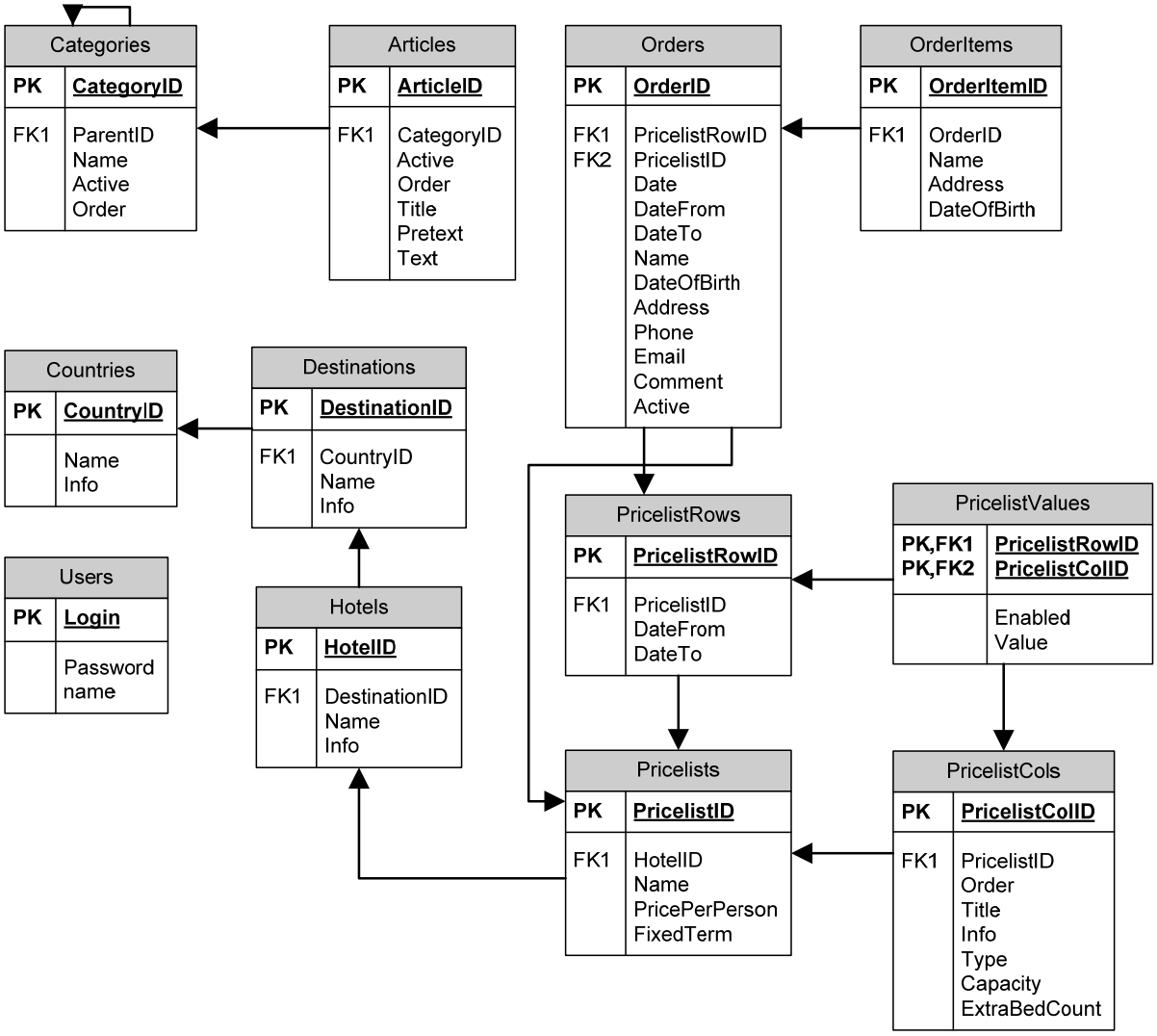
Ako som už v úvode spomínal, ukážky praktického využitia refaktoringu budem predvádzať na vlastnom projekte. Ide o jednoduchý redakčný systém, ktorého hlavnou funkciou je správa obsahu webovej stránky. Systém umožňuje vytvárať kategórie, ktoré reprezentujú rubriky stránky. Tieto rubriky tvoria stromovú štruktúru samotnej stránky. Do jednotlivých rubrík je možné pridávať články, ktoré tvoria samotný verejne dostupný obsah webovej stránky. Článkom ako aj rubrikám je možné nastaviť príznak aktívnosti, ktorý hovorí o tom či sa majú články resp. rubriky na stránke zobrazovať. Takisto je obom objektom možné určiť poradie v akom sa na stránke zobrazovať. Toto je základná funkcia aplikácie. Ďalej pribudla možnosť definovania cenníkov zájazdov a evidencia nezáväzných objednávok. Keďže zájazdy sú vždy viazané k určitému hotelu alebo ubytovaciemu zariadeniu, pribudla k správe cenníkov aj evidencia hotelov, letovísk a krajín.

K pohodlnému zadávaniu a editovaniu dát existuje administračné prostredie, do ktorého je nutné sa prihlásiť menom a heslom aby tak dáta mohla editovať len oprávnená osoba. Musí tak existovať ešte aj evidencia užívateľov administrácie spolu s užívateľskými právami. Poslednou aplikáciou je samotná webová stránka, ktorá je verejne prístupná.

V nasledujúcich podkapitolách aplikáciu podrobne popíšem. Najskôr sa budem venovať jednotlivým entitám a dátovému modelu. Potom popíšem objektový model, ktorý vychádza z modelu dátového. Nakoniec popíšem ako je projekt implementovaný a rozdelený do jednotlivých logicky oddelených časti a ako to vyzerá v prostredí .NET vo vývojovom prostredí Microsoft Visual Studia .NET 2005.

6.1.1 Dátový model

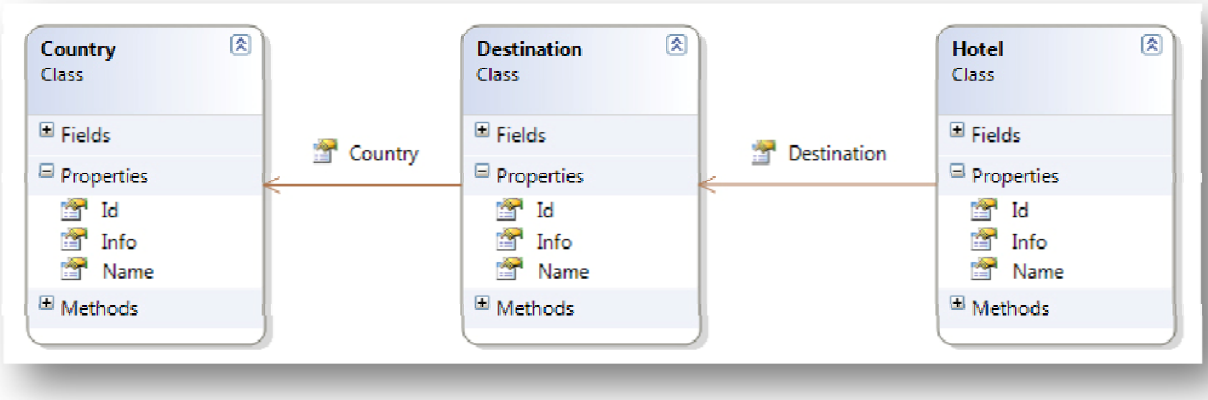
Z popisu aplikácie vyplývajú aj požiadavky na dátový model. V popise aplikácie som zmienil entity ako kategória, článok, cenník, krajina, destinácia, hotel a užívateľ. Aké majú tieto entity vlastnosti a aké sú medzi nimi vzťahy sú najlepšie vidieť na nasledujúcom diagrame (Obr. 17.). Ten hovorí sám za seba a myslím, že ho nie je nutné opisovať.



Obr. 17. E-R Diagram

6.1.2 Objektový model

Objektový model vychádza priamo z dátového modelu. Triedy sú vytvorené tak, že presne kopírujú stĺpce príslušných databázových tabuliek a nemajú žiadneho predka. Model teda nie je ničím zaujímavý a myslím, že je jednoducho predstaviteľný, preto ho nebudem ďalej podrobnejšie rozoberať.

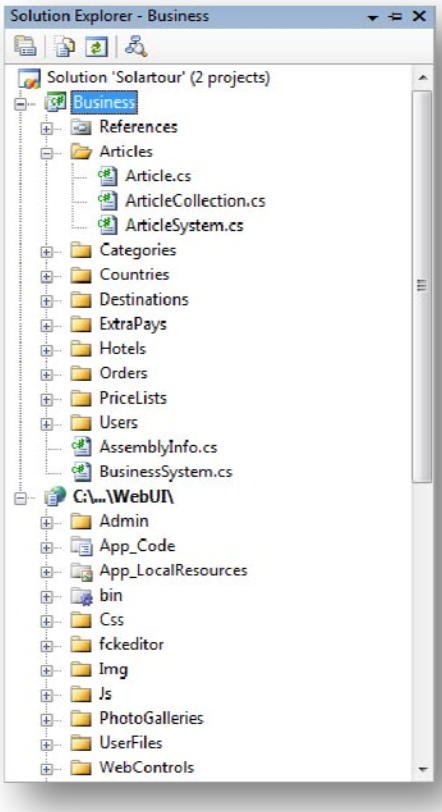


Obr. 18. Ukážka časti objektového modelu

Na predchádzajúcom obrázku (Obr. 18.) je názorná ukážka jednej časti objektového modelu, týkajúceho sa evidencie krajín, destinácií a hotelov. Vidíme, že všetky triedy majú rovnaké vlastnosti a nemajú žiadneho spoločného predka, týmto problémom sa budeme zaberať neskôr. Na obrázku vidieť aj akým spôsobom sú modelované väzby medzi objektmi. Ostatné triedy sú vytvorené presne v rovnakom duchu takže nie je nutné zaoberať sa všetkými triedami. V nasledujúcej kapitole stručne popíšem ako je projekt implementovaný a potom sa už budem venovať samotným refaktoringom projektu.

6.1.3 Implementácia a vrstvy aplikácie

Projekt je implementovaný v jazyku C# a ASP.NET 2.0. Ako vývojové prostredie bolo použité na začiatku Visual Studio .NET 2003, neskôr som prešiel na novšie Visual Studio .NET 2005. Ako databáza je použitá MySql. Celý projekt (v prostredí Visual Studia *solution*) je zložený z dvoch samostatných projektov. Prvý nazvaný *Business* je knižnica (*Class library*) a obsahuje všetky biznis objekty, triedy a prístup do databáze a biznis logiku. Druhý projekt je webová aplikácia, ktorá obsahuje ako webovú prezentáciu cestovnej kancelárie tak aj administračné rozhranie. Rozdelené sú len tým, že sú v rôznych adresároch.



Obr. 19. Ukážka projektu vo Visual Studiu.NET

Pravidlo, ktorým som sa pri programovaní tohto projektu snažil držať bolo to, že každá databázová entita má svoju triedu, ktorá by mala jedna k jednej mapovať databázovú tabuľku. Ďalej vždy existuje trieda reprezentujúca kolekciu daného objektu a trieda končiaca príponou *System*, ktorá obsahovala prístup do databáze, biznis logiku, skrátka všetko čo sa s danou entitou dalo robiť.

V rámci refaktoringu okrem skvalitnenia kódu projekt prevediem do Visual Studia.NET 2008 a projekty rozčlením do viacerých menších logicky oddelených projektov. Ďalším krokom, ktorým sa nebudem podrobnejšie zaoberať bude refaktor a optimalizácia databáze.

6.2 Stav pred refaktoringom

Ešte predtým ako začnem s aplikáciou refaktoringu, musíme si niečo povedať o stave aplikácie a zdrojovom kóde aby bolo možné zhodnotiť dosiahnuté výsledky po refaktoringu, prípadne porovnať stav pred refaktoringu so stavom po. Zmerať stav softwaru po stránke kvality zdrojového kódu je dosť komplikované. Čo je možné zmerať pomerne jednoducho je napríklad celkový počet riadkov zdrojového kódu, priemerný počet znakov na riadok, priemerný počet riadkov na súbor, počet tried a úroveň dedičnosti. Tieto hodnoty nám však niečo málo povedia o veľkosti projektu, nie však o jeho kvalite a zložitosti. Na zmeranie týchto hodnôt nám môže slúžiť napríklad už spomínaný nástroj

FxCop od spoločnosti Microsoft, ktorým môžeme skontrolovať dodržovanie dobrých zásad pri programovaní. Ďalšou hodnotou ktorú môžeme zmerať je tiež už spomínaná *cyklomatická zložitosť,* ktorú hneď v skratke opíšem. Na jej zmeranie použijem Visual Studio.NET 2008 Team Edition.

6.2.1 Cyklomatická zložitosť

Cyklomatická zložitosť je jednou zo softwarových metrík. Bola vytvorená Thomasom McCabeom a je používaná na meranie zložitosti programu, meria počet lineárne nezávislých ciest zdrojovým kódom programu. [8]

6.2.1.1 Definícia

$$M = E - N + 2P$$

Kde

M = cyklomatická zložitosť

E = počet hrán v grafe

N = počet uzlov v grafe

P = počet prepojených komponent

6.2.1.2 Základný princíp

Cyklomatická zložitosť je meraná pomocou grafu, ktorý popisuje tok programu. Uzly grafu reprezentujú príkazy programu (väčšinou jeden riadok kódu) a orientovaná hrana spája dva uzly ak druhý uzol (príkaz) môže byť vykonaný ihneď po prvom. Napríklad ak blok kódu neobsahuje žiadne podmienky (príkaz *if* alebo *switch*) existuje vždy len jedna možná cesta cez zdrojový kód, cyklomatická zložitosť tohto bloku kódu je teda 1. Čím je teda v kóde viac podmienok tým pádom viac ciest, cyklomatická zložitosť je vyššia.

6.2.2 Metriky

Ako som spomenul na začiatku kapitoly, zmerať stav a kvalitu softwaru z pohľadu zdrojového kódu je komplikované ak vôbec možné. Nejaké hodnoty súčasného stavu sú ale potrebné na neskoršie porovnanie a zhodnotenie. Prvými a ľahko merateľnými hodnotami sú:

celkový počet súborov	78
celkový počet riadkov zdrojového kódu	9812
celkový počet neprázdnych riadkov zdrojového kódu	6590
priemerný počet riadkov na jeden súbor	125

Tab. 2. Hodnoty hovoriace o celkovej veľkosti projektu

[8] http://en.wikipedia.org/wiki/Cyclomatic_complexity

celkový počet súborov	41
celkový počet riadkov zdrojového kódu	4555
celkový počet neprázdnych riadkov zdrojového kódu	2809
priemerný počet riadkov na jeden súbor	111

Tab. 3. Hodnoty projektu *Business*

celkový počet súborov	37
celkový počet riadkov zdrojového kódu	5257
celkový počet neprázdnych riadkov zdrojového kódu	3781
priemerný počet riadkov na jeden súbor	142

Tab. 4. Hodnoty webového projektu *WebUI*

Z predchádzajúcich tabuliek je možné vidieť, že projekt nie je príliš rozsiahly, na ukážku aplikácie refaktoringu nám však bude stačiť. Pri neskoršom hodnotení výsledkov uvidíme ako sa aplikácia refaktoringu a skvalitnenie zdrojového kódu prejavilo na týchto hodnotách. Predpokladám, že odstránením duplicít, či už použitím refaktoringu *Extract method* alebo *Pull up method* by sa mal celkový počet riadkov zdrojového kódu znížiť. Vytvorením nových tried, ktoré budú združovať spoločné vlastnosti a funkcionalitu zase narastie počet súborov. Keďže veľká časť biznis logiky je implementovaná priamo v prezentačnej vrstve na formulároch, čo je samozrejme nesprávne, bude veľká časť kódu presunutá do projektu *Business*, prípadne novo vytvorených projektov, tak aby nebola priamou súčasťou prezentačnej logiky. Tým sa zvýši počet riadkov a súborov v projekte *Business* a počet riadkov sa zníži vo webovom projekte *WebUI.*

Ďalšími metrikami, ktorými sa budem zaoberať sú cyklomatická zložitosť, class coupling, index udržovateľnosti a úroveň dedičnosti. Všetky tieto metriky je možné zmerať priamo vo Visual Studiu.NET vo verzii Visual Studio Team System 2008. Cyklomatickú zložitosť sme si opísali v predchádzajúcej podkapitole, takže len v skratke, čím nižšia hodnota tým lepšie. Naopak, hodnota pri indexe udržovateľnosti je čím vyššia tým lepšie, hodnoty sa pohybujú v intervale 0 až 100. Úroveň dedičnosti hovorí o hĺbke v strome dedičnosti a class coupling je hodnota, ktorá hovorí o počte objektov ktoré sú v rámci triedy alebo metódy referencované, platí čím nižšie číslo tým lepšie.

Trieda	**udržovateľnosť**	**cykl. zložitosť**	**úroveň dedičnosti**	**class coupling**	**počet riadkov**
PriceListColumnType	100	0	1	0	0
ExtraBedType	100	0	1	0	0
ExtraBedSystem	97	2	1	1	3
User	93	5	1	0	7
PriceListValue	93	11	1	3	11

ExtraBed	93	4	1	0	6
HotelCollection	92	3	2	4	4
CategoryCollection	92	3	2	4	4
ArticleCollection	92	3	2	4	4
Country	92	9	1	0	12
Category	92	8	1	0	12
Article	92	10	1	0	14
OrderItemCollection	92	3	2	4	4
OrderCollection	92	3	2	4	4
Room	92	8	1	0	12
Hotel	92	8	1	1	12
PriceListRowCollection	92	3	2	4	4
CountryCollection	92	3	2	4	4
PriceListCollection	92	3	2	4	4
PriceListColumnCollection	92	3	2	4	4
DestinationCollection	92	3	2	4	4
PriceListColumn	91	19	1	1	26
OrderItem	91	12	1	1	17
ExtraPay	91	15	1	3	23
Destination	91	10	1	1	14
Order	90	35	1	5	44
PriceList	82	20	1	8	31
PriceListValueCollection	81	8	2	8	11
PriceListRow	72	12	1	1	33
BusinessSystem	71	12	1	5	22
DestinationSystem	69	15	2	9	53
UserSystem	69	3	2	7	11
CategorySystem	69	11	2	8	45
CountrySystem	69	12	2	8	47
HotelSystem	69	11	2	10	50
ArticleSystem	62	14	2	9	63
PriceListSystem	60	110	2	23	426
OrderSystem	48	15	2	16	121
priemer	85,03	11,29	1,50	4,42	30,68
min	48	0	1	0	0
max	100	110	2	23	426

Tab. 5. Hodnoty metrík z pohľadu všetkých tried v projekte *Business*

V tabuľke (Tab. 5.) sú hodnoty metrík namerané pomocou Visual Studia.NET z pohľadu tried, ich priemerné, maximálne a minimálne hodnoty. Z tabuľky môžeme vyvodiť niekoľko záverov a predpokladov. Priemerný index udržovateľnosti je 85, čo nie zlé. Niektoré triedy sa však od tejto

hodnoty výrazne líšia, napríklad *PriceListSystem* a *OrderSystem,* teda triedy slúžiace na prácu s cenníkmi a objednávkami. Očakávam, že aplikovaním refaktoringu sa tento index výrazne zvýši. Druhou metrikou je cyklomatická zložitosť. Tu sa však jej hodnota nepohybuje v dopredu známom intervale. Platí však pravidlo čím nižšie číslo, tým lepšie. Vidíme, že najnižšia hodnota je 0 a tá sa objavuje len pri dvoch entitách a to *PriceListColumnType* a *ExtraBedType*, čo sú výčtové typy (enumerátory). Priemerná hodnota je 11,3 a z tabuľky vidieť, že výrazne vybočuje jedna trieda a to znovu *PriceListSystem.* Podobne je to aj pri posledných dvoj metrikách, kde znova vybočujú triedy *PriceListSystem* a *OrderSystem.* Môžeme teda predpokladať, že práve tieto dve triedy na tom budú najhoršie. Či už preto, že obsahujú najviac kódu a logiky alebo skrátka preto, že sú napísané nesprávne. Máme teda akýsi východiskový bod pre refaktoring, takisto pred sebou máme cieľ, ktorý chceme refaktoringom dosiahnuť a môžeme sa teda do aplikácie pustiť. Ešte predtým však na aplikáciu použijeme nástroj FxCop, ktorý nám poskytne zoznam problémových miest, ktoré nedodržujú zásady dobrého programovania. Budeme tak mať slušný prehľad o tom v akom stave aplikácia je a budeme mať jasno kde začať.

Po spustení analýzy na projekt *Business,* na webový projekt nie je analýzu možné spustiť, program FxCop našiel 177 problémov, z čoho je väčšina na triedu *PriceListSystem* a *OrderSystem*, čo sme aj na základe predchádzajúcich metrík mohli očakávať. Nasledujúca tabuľka (Tab. 6.) ukazuje stav projektu podľa analýzy programom FxCop.

kritické chyby	3
chyby	155
kritické varovania	7
varovania	12
Suma	**177**

Tab. 6. Hodnoty analýzy programu FxCop

Problémy nájdené pomocou programu FxCop sú rozdelené do niekoľkých kategórií a do niekoľkých podkategórií. Nám postačia však hlavné kategórie. Pri každom probléme je okrem kategórie a podkategórie uvedený aj zdroj, popis chyby a veľmi stručný návod ako problém odstrániť.

Myslím, že máme všetko potrebné na to, aby sme sa mohli pustiť do reálnej aplikácie refaktoringov. Máme zaznamenaný počiatočný stav aplikácie, z ktorého vychádzame a smer ktorým sa chceme pohnúť. V nasledujúcej kapitole popíšem postupne celý proces refaktoringu predstavenej aplikácie. Pri kľúčových krokoch uvediem príklady a dôvody použitého refaktoringu.

6.3 Aplikácie refaktoringu

V predchádzajúcej kapitole sme si niečo povedali o súčasnom stave aplikácie, zmerali nejaké hodnoty, ktoré nám v závere pomôžu zhodnotiť výsledky. Taktiež sme aplikáciu analyzovali pomocou nástroja FxCop. Výsledky tejto analýzy nám poskytli dobrý základ a štartovací bod. Keď pôjdeme po chybách a varovaniach, ktoré tento nástroj našiel, objavíme zároveň ďalšie slabé a nesprávne navrhnuté bloky kódu, na ktoré môžeme refaktoring aplikovať.

Ako som spomínal vyššie formuláre vo webovom projekte *WebUI* obsahuje okrem prezentačnej logiky aj hromadu biznis logiky, dokonca aj prístup do databáze. Toto je samozrejme nesprávne, môžeme totiž upraviť metódy a a triedy v projekte *Business* a zmena sa neprejaví vo webovej aplikácii. Preto prvým krokom ešte pred odstraňovaním chýb a varovaní nájdených nástrojom FxCop bude odstránenie biznis logiky z prezentačnej vrstvy a presun do príslušných tried v projekte *Business*.

6.3.1 Presun biznis logiky z prezentačnej vrstvy

Táto situácia je ako stvorená na použitie refaktoringu *Extract method* a *Move method*. Refaktoringom *Extract method* vytvoríme požadované metódy z blokov kódu, ktoré chceme odstrániť a pomocou *Move method* ich presunieme do príslušných tried. Ešte pred samotným refaktoringom by som rád ujasnil pojmy biznis a prezentačná logika. Prezentačná logika je správanie sa grafického užívateľského rozhrania napríklad po akcii užívateľa alebo na základe načítaných dát. Biznis logika reprezentuje procesy, ktoré sa dejú pri manipuláciou s dátami a nemajú nič spoločné s tým ako sú prezentované alebo ako sú fyzicky uložené. Napríklad pri vytvorení nového riadku cenníka (termín) je nutné overiť, či cenník už existuje, ak nie, vytvoriť prázdny, prípadne proces ukončiť s nejakou chybou atď.

Typickým príkladom tejto situácie je nasledujúci kód. Tento kód obsluhuje klik na tlačítko, po stlačení z databáze načíta identifikátor kategórie práve editovaného článku a presmeruje aplikáciu na inú stránku. Na tom by nebolo nič zlé, keby v kóde nebol priamo skladaný SQL dotaz, ktorý je potom priamo použitý na prístup do databáze. Takýto spôsob vedie k mnohým problémom, hlavne pri veľkých projektoch. Keď nastane napríklad zmena názvu tabuľky tak miesto opravy na mieste určenom na prístup do databáze je nutné prehľadať celý kód a opraviť všetky výskyty. To isté platí pre zmenu dátového typu stĺpca tabuľky atď.

```
protected void CategoryBack_Click(object sender, System.EventArgs e)
{
        string ID = Request["ArticleID"];
        OdbcConnection conn = new OdbcConnection(strConnect);
        conn.Open();
        OdbcCommand cmd = new OdbcCommand(
        "SELECT CategoryID FROM Articles WHERE ArticleID=" + ID, conn);
        OdbcDataReader reader = cmd.ExecuteReader();
        reader.Read();
```

```
        string categoryID = reader.GetString(0);
        reader.Close();
        conn.Close();
        Response.Redirect("default.aspx?ParentID=" + categoryID);
}
```

Kód 18. Ukážka prístupu do databáze priamo v prezentačnej vrstve

Podobných (často krát oveľa zložitejších) blokov kódu ako je ten v ukážke vyššie je v prezentačnej vrstve, hlavne v administračnej časti veľa. Na takýto kód použijeme refaktoring *Extract method*, vytvoríme metódu v tomto konkrétnom prípade napríklad *GetCategoryId(int articleId)*, ktorá z databáze načíta identifikátor kategórie, do ktorej článok patrí. Na refaktoring použijeme Visual Studio. Po dokončení projekt preložíme, otestujeme a zistíme, že všetko funguje tak ako predtým. Vytvorená metóda pracuje len s prístupom do databáze, nemá teda nič spoločné s prezentačnou logikou a existuje pravdepodobnosť že podobnú metódu využijeme na ďalších miestach v aplikácii. Metódu teda presunieme (refaktoring *Move method*) do triedy, ktorá slúži na prácu s kategóriami, čiže do triedy *CategorySystem.* Kód po refaktoringu bude vyzerať nasledovne.

```
protected void CategoryBack_Click(object sender, System.EventArgs e)
{
        int categoryId = CategorySystem.GetCategoryId(int.Parse(Request["ArticleID"]));
        Response.Redirect("default.aspx?ParentID=" + categoryId);
}
```

Kód 19. Kód po aplikácií refaktoringu

Je vidieť, že kód po refaktoringu je výrazne jednoduchší a úplne jasný na pochopenie. Okrem *Extract method* a *Move method* bol použitý ešte refaktoring *Inline temp.* Na odstránenie prvého riadka kódu, v ktorom sa do premennej *ID* priradila hodnota, táto premenná sa použila iba jeden krát, preto je práve vhodné použiť *Inline temp.*

Podobných miest je v prezentačnej vrstve veľa, preto nebudem uvádzať všetky. Hlavný je myslím postup, dôvod a výsledok. Postup je popísaný v katalógu refaktoringov a je zautomatizovaný tým, že na refaktoring používame nástroj (Visual Studio.NET). Dôvod je jasný, chceme dosiahnuť oddelenie biznis logiky od prezentačnej logiky a tým o sprehľadnenie kódu. Výsledok je možné vidieť na predchádzajúcich ukážkach kódu. V tomto duchu teda upravím zvyšok webovej aplikácie a dosiahnem ním to, že všetok kód týkajúci sa biznis logiky bude v projekte *Business*, ktorým sa budem zaberať ďalej.

6.3.2 Oddelenie biznis logiky od prístupu do databáze

V prvom kroku sme oddelili prezentačnú vrstvu od biznis logiky. Tým sme dosiahli to, že všetka logika je na jednom mieste a má jasne definované rozhranie. Tým, že máme logiku na jednom mieste, môžeme bez problémov postaviť ďalšiu, či už webovú alebo windows aplikáciu s tým, že bude používať rovnaké metódy ako už existujúca aplikácia, čo je veľmi dôležité. Pri zmene na nižšej vrstve

(*Business*) pri zachovaní jej rozhrania, zachováme aj správne fungovanie prezentačnej vrstve a to je presne to o čo sa snažíme. Je to aj znak a vlastnosť dobre navrhnutého viacvrstvového softwarového projektu.

Momentálne máme dve logicky oddelené vrstvy a to prezentačnú a biznis vrstvu. Prezentačnú sme v predchádzajúcej kapitole upravili, „vyčistili“ a ďalej sa s ňou nebudeme zaoberať. Ďalej nás bude zaujímať práve biznis vrstva implementovaná v projekte *Business*. Táto vrstva obsahuje ako biznis logiku tak aj prístup do databáze, čiže je priamo závislá na fyzickom uložení dát a to konkrétne na databáze MySql. Zmena spôsobu uloženia dát by v súčasnom stave nebola možná alebo aspoň nie jednoducho. Keďže ide o logiky oddelené funkciu, mal by pre prístup do databáze existovať oddelený priestor. Ide o podobný vzťah ako medzi prezentačnou a biznis logikou. Jedným riešením môže byť vytvorenie nového projektu *DataAccess*, ktorý by obsahoval triedu pre každú tabuľku v databáze. Tá by zabezpečovala základné *CRUD* (Create Read Update delete) operácie nad jednou tabuľkou. Biznis logika by pracovala s príslušnými triedami tohto projektu miesto skladania SQL dotazov a priameho prístupu do databáze. Týmto spôsobom pri zmene databázového servera stačí zmeniť alebo upraviť práve triedy v tomto projekte. Môžeme dokonca podporovať niekoľko dátových úložišť zároveň. V konkrétnom projekte je to samozrejme málo pravdepodobný scenár, ide však o princíp a ukážku refaktoringu.

Vytvorením nového projektu vznikajú určité problémy a to minimálne v tom, že nie je možné mať referenciu z projektu *Business* na projekt *DataAccess* a aj opačným smerom, museli by sme teda vytvoriť ešte jeden projekt. Na ukážku refaktoringu je to ale príliš zložité a zdĺhavé. Preto miesto nového projektu len vytvoríme nový adresár a menný priestor (*namespace*) v projekte *Business* a nazveme ho *DataAccess*. Tento menný priestor bude pre každú databázovú tabuľku obsahovať jednu statickú triedu s príponou *DA (DataAccess)*, ktorá bude poskytovať silne typované rozhranie do databáze. Princíp ukážem na triede, ktorá podľa metrík najzložitejšia (cyklomatická zložitosť) a to *PricelistSystem*. Pri ostatných triedach by bol postup analogický, ale kvôli jednoduchosti nedostatku priestoru nebudem zmenu aplikovať na všetkých triedach. Sú príliš jednoduché na to aby sa táto zmena reálne prejavila na metrikách.

Táto trieda zastrešuje všetku prácu s cenníkmi, to znamená vytváranie, mazanie a úprava. To však znamená aj prácu so stĺpcami, riadkami a cenami, čiže pracuje sa so 4 biznis entitami. Pre každú z nich vytvoríme *DataAccess* triedu, budú to teda triedy *PricelistDA, PricelistColumnDA, PricelistRowDA* a *PricelistValueDA*. Metódy týchto tried sa budú potom používať v triede *PricelistSystem* miesto priameho skladania SQL dotazov. Kód bude teda odtienený od fyzického uloženia v databáze a bude jednoduchší a ľahšie pochopiteľný. Navyše metódy novo vytvorených tried môžeme v budúcnosti využiť pri rozširovaní funkcionality.

```
public static bool LoadPricelistColumn(PricelistColumn column)
{
```

```
        OdbcConnection connection = new
        OdbcConnection(ConfigurationManager.AppSettings["connectionString"]);
        connection.Open();

        OdbcCommand command = new OdbcCommand("select * from PricelistCols where
        PricelistColID='" + column.Id + "'", connection);

        OdbcDataReader reader = command.ExecuteReader();

        bool result = false;

        if (reader.Read())
        {
                column.PricelistId = reader.GetInt32(reader.GetOrdinal("PriceListID"));
                column.Order = reader.GetInt32(reader.GetOrdinal("nOrder"));
                column.Name = reader.GetString(reader.GetOrdinal("Title"));
                column.Info = reader.GetString(reader.GetOrdinal("Info"));
                column.Type = (PricelistColumnType)reader.GetInt32(reader.GetOrdinal("Type"));
                column.Capacity = reader.GetInt32(reader.GetOrdinal("Capacity"));
                column.ExtraBedCount = reader.GetInt32(reader.GetOrdinal("ExtraBedCount"));

                result = true;
        }

        reader.Close();
        connection.Close();

        return result;
}
```

Kód 20. Ukážka metódy na načítanie stĺpca cenníka z databáze

Podľa vzoru uvedeného v ukážke vytvoríme podobné metódy pre všetky 4 biznis entity. Príslušný kód pre každú tabuľku nájdeme na rôznych miestach triedy *PricelistSystem.* Metódy vytvoríme použitím refaktoringu *Extract method* a *Move method.* Rovnakým spôsobom vytvoríme aj metódy pre *Insert, Delete* a *Update.* Pri prechádzaní kódom nachádzame množstvo kódu, ktorý sa stále opakuje, ide presne o ten kód pre ktorý sme vytvorili metódy v nových triedach, preto tieto miesta upravíme tak aby volali nové metódy. Podobne ako pri „čistení“ prezentačnej vrstvy od biznis logiky aj teraz kód sprehľadníme, zjednodušíme a odstránime množstvo duplicity.

Pri vytváraní *Insert* metódy na vytvorenie cenníka narazíme na metódu *InsertNewPricelist*, ktorá má tri parametre. Tieto reprezentujú vlastnosti cenníka a sú použité na vytvorenie SQL dotazu. Keďže všetky parametre súvisia s cenníkom, mali by sme miesto zoznamu parametrov použiť priamo objekt typu *Pricelist.* Je to aplikácia refaktoringu *Introduce parameter object.*

```
public static int InsertNewPricelist(int hotelID, string name, bool fixedTerm)
{
        ...
}
```

```
public static int InsertNewPricelist(Pricelist pricelist)
{
        return InsertNewPricelist(pricelist.Hotel.Id, pricelist.Name, pricelist.FixedTerm);
}
```

Kód 21. Ukážka *Introduce parameter object*

Ma to niekoľko výhod, tie sú spomenuté v kapitole 3. Spomeniem len jednu a tou je to, že pri rozšírení databázovej tabuľky o ďalší stĺpec nemusíme zmeniť hlavičku metódy a pridať nový parameter, stačí upraviť skladanie SQL dotazu, to sa deje ešte o úroveň nižšie v *DataAccess* triede. Keď už sa zaoberáme touto metódou, premenujeme ju na *CreatePricelist* (použitie refaktoringu *Rename method*), ktorá bude volať *InsertPricelist* metódu triedy *PricelistDA*.

V tejto kapitole sme sa zaoberali oddelením biznis logiky od prístupu do databáze a ukázali sme si použitie niekoľkých refaktoringov. Princíp bol veľmi podobný ako pri predchádzajúcej kapitole s rozdielom, že sme vytvárali nové triedy. Záver je teda podobný ako pri predchádzajúcej kapitole. Logicky sme oddelili dve vrstvy, sprehľadnili a zjednodušili sme kód, umožnili zmenu fyzického uloženia dát a zjednodušili situáciu pri zmene štruktúry databáze. Zaviedli sme aj štandardný prístup k novým biznis entitám a poskytli jednotné rozhranie. Zmeny boli však aplikované na triedu *PricelistSystem* a štandardné *DataAccess* triedy boli vytvorené iba pre 4 tabuľky aj to nie úplne. Na ukážku refaktoringu a ilustráciu postupu to ale stačí. V praxi by však bolo vhodné celý postup aplikovať na celý projekt *Business* a tak získať štandardný prístup k databáze ako aj jednoduché pridanie novej biznis entity. V nasledujúcej kapitole sa budem venovať úprave *class modelu* a dedičnosti.

6.3.3 Úprava hierarchie tried

Triedy reprezentujúce databázové tabuľky a zároveň biznis entity budeme nazývať *biznis objekty*. Vlastnosti týchto objektov jedna k jednej kopírujú stĺpce príslušnej tabuľky a nemajú teda žiadneho predka. To je z návrhového hľadiska nesprávne, triedy ktoré k sebe logicky patria by mali mať spoločnú nadradenú triedu alebo aspoň implementovať rovnaké rozhranie. Preto vytvoríme novú triedu, ktorá bude základnou triedou pre všetky *biznis objekty*. Túto triedu nazveme *BusinessObject* a bude z nej dediť každá trieda reprezentujúca *biznis objekt*, teda databázovú tabuľku. Keď prejdeme cez všetky triedy zistíme, že všetky majú spoločnú jednu vlastnosť a to *Id* (identifikátor objektu). Túto vlastnosť teda presunieme do nadradenej triedy použitím refaktoringu *Pull Up Field*. Celý tento proces je vlastne refaktoring *Extract Superclass* pomocou ktorého vytvoríme nadradenú triedu so množinou spoločných vlastností. Týmto krokom dostaneme lepší návrh a zároveň sa zbavíme duplicity v podobe opakujúcej sa vlastnosti *Id* v a každej triede.

```
public abstract class BusinessObject
{
        private int _id;

        protected BusinessObject() { }

        protected BusinessObject(int id)
        {
          this._id = id;
        }

        public int Id
        {
```

```
            get { return this._id; }
            set { this._id = value; }
        }
}
```

Kód 22. Ukážka novo vytvorenej triedy.

Z každej triedy teda odstránime privátnu premennú a príslušnú vlastnosť *Id* a triedu označíme tak, že dedí z triedy *BusinessObject*. Krok je to veľmi jednoznačný a jednoduchý, preto nebudem uvádzať ukážky kódu. Po tomto kroku projekty preložíme a overíme si, či funkčnosť zostala zachovaná. Teraz môžeme pokračovať ďalej. Po ďalšom skúmaní tried zistíme, že niektoré z nich majú spoločné dve vlastnosti a to *Name* a *Info* (názov a popis). Znovu pomocou *Extract superclass* vytvoríme pre triedy *Article, Category, Country, Destination, ExtraPay, Hotel* a *Order* spoločnú triedu s týmito dvoma vlastnosťami a nazveme ju *CommonBusinessObject.* Postup je analogický ako pri vytváraní triedy *BusinessObject*.

```
public abstract class CommonBusinessObject : BusinessObject
{
        private string _name;
        private string _info;

        protected CommonBusinessObject()
                : base()
        { }

        protected CommonBusinessObject(int id)
                : base(id)
        {
        }

        public string Name
        {
                get { return this._name; }
                set { this._name = value; }
        }

        public string Info
        {
                get { return this._info; }
                set { this._info = value; }
        }
}
```

Kód 23. Ukážka kódu triedy *CommonBusinessObject*.

Podobne ako pri predchádzajúcom kroku sme o niečo vylepšili *class model* a zbavili sa duplicity v podobe dvoch vlastností. V budúcnosti, keď budeme vytvárať novú triedu a bude obsahovať tieto dve vlastnosti mamé ušetrenú nejakú prácu. V konkrétnom príklade je to práca skoro zanedbateľná, ide však len o ilustráciu princípu.

V tejto kapitole sme si v praxi ukázali refaktoringy *Extract superclass* a *Pull up field.* V prípade, že by triedy, ktorých sa refaktoring týkal obsahovali aj spoločné metódy, použili by sme aj refaktoring *Pull up method,* ktorý je veľmi podobný ako *Pull up field.* Povedali sme si aj to, čo nám dané refaktoringy priniesli, prípadne v čom nám uľahčia prácu v budúcnosti. V nasledujúcej kapitole sa budem venovať problémom, chybám a varovaniam, ktoré boli odhalené nástrojom FxCop.

6.3.4 Odstránenie chýb nájdených nástrojom FxCop

V úvode tejto kapitoly sme si v rámci zachovania stavu aplikácie pred refaktoringom, spustili analýzu kódu nástrojom od spoločnosti Microsoft FxCop. Tento nástroj kontroluje pravidlá a zásady dobrého programovania. V tejto časti sa budem zaoberať odstraňovaním práve týchto problémov.

Pri prechádzaní zoznamu chýb a varovaní zisťujem, že množstvo chýb sa týka nesprávneho názvoslovia, konkrétne ide o triedu *PriceList* (cenník) a všetky triedy s touto predponou. Toto slovo nie je zložením slova *Price* a *List* ale ide o jedno anglické slovo, preto by písmeno *L* nemalo byť veľké. Správny tvar slova je teda *Pricelist* a to je presne to, čo FxCop označuje za chybu. Na odstránenie tejto chyby použijeme asi najznámejší a najpoužívanejší refaktoring a to *Rename* (či už premenovanie triedy alebo názvy premennej alebo názvu vlastnosti). Refaktoring môžeme vykonať aj ručne a to textového vyhľadania slova *PriceList* a nahradením slovom *Pricelist.* Refaktoring však za nás pohodlne vykoná Visual Studio.NET, resp. jeho rozšírenie JetBrains ReSharper musíme ho však aplikovať na všetky triedy, vlastnosti a premenné ktoré obsahujú slovo *PriceList.* Týmto veľmi jednoduchým spôsobom sme sa zbavili cez 50 chýb a varovaní. Ešte na chvíľu pri názvosloví zozname a napravíme ostatné chyby tohto typu. Ďalšou chybou týkajúcou sa názvu je pomenovanie parametru metódy *InsertNewPricelist,* presne parametra *bFixedTerm*. FxCop reklamuje predponu *„b“* v názve parametra a má samozrejme pravdu, táto predpona tam nemá čo robiť. Použijeme teda opäť refaktoring *Rename* a parameter premenujeme na *fixedTerm*.

Ďalšou početnou skupinou chýb, sú chyby ohľadom kolekcií. Väčšina chýb a varovaní sa týka toho, že by kolekcie mali byť silne typované a poskytovať silne typovanú implementáciu rozhrania *IList*, ktorá poskytuje metódy na pridávanie, odoberanie a vyberanie prvkov z kolekcie. Keďže platforma .NET od verzie 2.0 poskytuje tzv. generické triedy použijeme miesto *CollectionBase* triedu *Collection<T>*, z ktorej budú dediť všetky biznis kolekcie. Zbavíme sa tak chýb hlásených FxCopom ale aj duplicitného kódu.

```
public class ArticleCollection : CollectionBase
{
        public virtual void Add(Article article)
        {
          this.List.Add(article);
        }

        public virtual void Remove(Article article)
        {
          this.List.Remove(article);
        }

        public virtual Article this[int index]
        {
          get
          {
             return (Article)this.List[index];
          }
        }
}
```

Kód 24. Pôvodný kód triedy

Kód na pridávanie a odoberanie prvku z kolekcie sa vo všetkých biznis kolekciách opakuje, líši sa len v rôznom type objektu. Toto jednoducho vyriešime, keď zmeníme všetky biznis kolekcie tak aby dedili z triedy *Collection<T>*.

```
public class ArticleCollection : Collection<Article>
{
}
```

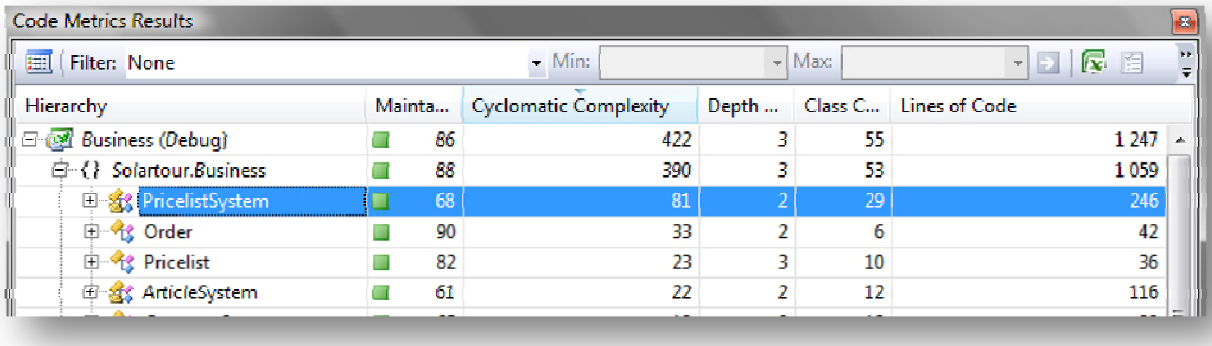
Kód 25. Trieda po zmene nadradenej triedy

Vidíme, že po zmene nadradenej triedy sme mohli vypustiť celý kód triedy. Toto platí pre všetky biznis kolekcie v projekte. Odstránili sme tak stále opakujúci sa kód a aj chyby nájdené FxCopom.

V zozname sa nachádza ešte niekoľko chýb týkajúcich sa nepoužívaných premenných. Tieto premenné môžeme jednoducho zmazať, zbavíme sa tak nepoužívaného kódu a sprehľadníme ho. Zoznam chýb síce ešte prázdny nie je, na ukážku a demonštráciu refaktoringu a princípu to myslím stačí, úplne vyčistenie od chýb a celkový redesign projektu ani nie je predmetom tejto práce. V tejto kapitole sme sa zaberali odstraňovaním návrhových chýb a upozornení nedodržiavania zásad dobrého programovania, nájdených nástrojom FxCop. Ukázali sme si ukážky niekoľkých refaktoringov, aj to čo nám priniesli. V nasledujúcej kapitole sa budem venovať posledným úpravám a ukážkam refaktoringu.

6.3.5 Posledné úpravy

Máme za sebou niekoľko krokov, ktorými sme zlepšili a vyčistili kód, ukážkovej aplikácie. Očistili sme webový projekt od biznis logiky a prístupu do databáze. Ukázali sme si na skupine biznis entít ako oddeliť biznis logiku od fyzického uloženia v databáze. Ďalej sme upravili hierarchiu tried tak, že sme vytvorili dve triedy, ktoré obsahujú spoločné vlastnosti všetkých, resp. množiny biznis entít a v predchádzajúcej kapitole sme sa venovali odstraňovaním chýb ktoré odhalila analýza nástroja FxCop. V tejto kapitole sa budem venovať posledným ukážkam a úpravám. Zameriam sa podľa súčasných metrík na najzložitejšiu triedu, ktorou je *PricelistSystem*. Jej hodnota cyklomatickej zložitosti výrazne vybočuje od priemeru (viď Obr. 20.).



Obr. 20. Ukážka metrík

Vysoká hodnota je daná aj tým, že trieda je obsahuje najviac kódu a pracuje s najväčším počtom tried. Tejto triede sme pomohli už predchádzajúcimi krokmi. Ako môžeme vidieť na metrikách zo začiatku (viď Tab. 5), zvýšil sa index udržovateľnosti, znížila sa aj cyklomatická zložitosť a výrazne klesol počet riadkov. To bolo spôsobené tým, že sme časť prístupu do databáze zjednotili a presunuli do samostatných *DataAccess* tried. Po rozbalení roletky triedy *PricelistSystem* zistíme, že metóda ktorá ma najvyššiu zložitosť je trieda *InsertNewRow*, táto metóda má nasledujúcu hlavičku (Kód 26.)

```
public static int InsertNewRow(int pricelistId, string dateFrom, string dateTo)
{
        ...
}
```

Kód 26. Ukážka hlavičky metódy *InsertNewRow*

Jej telo je plné podmienok a neprehľadných konštrukcií. Refaktorovať by ju bolo zbytočné a príliš pracné. Je to príklad toho, kedy je lepšie celý blok kódu zmazať a napísať ho znovu. Po bližšom skúmaní zistíme, že šlo v podstate len o ručný prevod dátumu z reťazcového formátu do dátového typu *DateTime* (začiatočnícka neznalosť). Kód teda prepíšeme bez akých koľvek podmienok na par riadkov, preložíme a pre istotu otestujeme a na záver ešte použijeme refaktoring *Inline temp* a kód bude vyzerať nasledovne (Kód 27).

```
public static int InsertNewRow(int pricelistId, string dateFrom, string dateTo)
{
        return InsertNewRow(pricelistId, DateTime.Parse(dateFrom), DateTime.Parse(dateTo));
}
```

Kód 27. Metóda *InsertNewRow* po refaktoringu

Metóda, ktorá mala ešte pred pár minútami 19 riadkov a cyklomatickú zložitosť 7 je teraz metódou s jedným riadkom a zložitosťou 1. Je zrozumiteľná a jasná hneď na prvý pohľad.

Ďalšou metódou ktorá má relatívne vysokú zložitosť je metóda *DeletePricelist*, ktorej kód je nižšie (Kód 28.).

```
public static void DeletePricelist(int pricelistId)
{
    Pricelist pricelist = GetPricelist(pricelistId);
    // delete values
    foreach (PricelistRow row in pricelist.Rows)
    {
        foreach (PricelistColumn column in pricelist.Columns)
        {
                DeletePricelistValue(row.Id, column.Id);
        }
    }
    PricelistColumnDA.DeletePricelistColumns(pricelistId);
    PricelistRowDA.DeletePricelistRows(pricelistId);
    PricelistDA.DeletePricelist(pricelist);
}
```

Kód 28. Metóda *DeletePricelist*

Na prvý pohľad je metóda v poriadku, v prvých krokoch vymaže ceny v cenníku, potom stĺpce, riadky a nakoniec samotný cenník. Vytvoríme metódu v *DataAccess* triede, ktorá vymaže všetky ceny v na základe identifikátora cenníka, tak sa zbavíme vnoreným cyklom a zároveň aj načítanie cenníku z databáze. Tento krát sa nám pomocou metrík optimalizovať kód, ktorý na prvý pohľad vyzerá byť v poriadku a možno by sme si ho ani neboli všimli. Ukázali sme si, ako nám meranie sledovanie metrík ako je napríklad cyklomatická zložitosť, môžu pomôcť k identifikovaniu slabých, neoptimalizovaných alebo zle napísaných miest v projekte. Toto je celkom zaujímavá cesta ako nájsť miesta, ktoré potrebujú refaktoring. Hlavne to platí pre rozsiahle projekty alebo pre človek, ktorý sa v danom projekt neorientuje najlepšie.

V tejto kapitole sme si previedli posledné úpravy a ukázali si ako môžeme pri refaktoringu s využitím metrík postupovať. Narazili sme aj na kód, ktorý bolo vhodnejšie zmazať a napísať znovu ako sa pokúšať o refaktoring, čo býva bežná situácia v praxi, hlavne keď je kód napísaný začiatočníkom. Nasleduje záver tejto kapitoly, kde sa pokúsim zhodnotiť dosiahnuté výsledky.

6.4 Dosiahnuté výsledky

Hodnotenie alebo meranie výsledku refaktoringu je veľmi ťažké. Ako sme si už na začiatku práce povedali, dobre navrhnutý softwarový projekt je jednoduchšie rozšíriteľný, jednoduchšie sa v ňom hľadajú a odstraňujú chyby. Je takisto jednoduchšie zapojiť nového programátora do prác na dobre navrhnutom projekte, pri ktorom sa dodržiavali zásady dobrého programovania. Keď aj tento programátor prichádza z iného prostredia a je zvyknutý má dobré programovacie návyky, nemá problém aktívne sa v rozumnom čase zapojiť do práce. Pri neudržovanom a tým pádom príliš zložitom projekte býva práve v tomto problém. Keďže ale projekt na vonok nezmení správanie (čo je podmienkou refaktoringu), sú tieto veci veľmi ťažko preukázateľné a to hlavne ľuďom, ktorí nie sú programátori (napríklad projektoví manažéri).

Ja sám mám z vykonaných zmenách na projekte veľmi pozitívny pocit. Projekt je čistejší, hlavne webová časť je očistená od logiky, ktorá je teraz ja jednom mieste. Prístup do databáze je takisto oddelený od logiky, čo prináša niekoľko výhod, ktoré som už spomínal. Class model je takisto o niečo vylepšený. Je pravda, že projekt sa o niečo zväčšil a pribudlo niekoľko nových tried, na druhej strane je ale kód oveľa prehľadnejší. Som presvedčený o tom, že keby k projektu pred mesiacom sadol niekto cudzí a dostal by úlohu doprogramovať nejakú funkcionalitu mal by s tým väčšie problémy ako dnes. Toto sú samozrejme skoro nič nehovoriace informácie, preto sa poďme pozrieť na to ako sa aplikované zmeny prejavili na metrikách. V nasledujúcej kapitole zhodnotím a porovnám hodnoty namerané pred a po aplikovaní refaktoringu.

6.4.1 Metriky

Rovnakým spôsobom ako na začiatku odmeriame rovnaké hodnoty teraz, po aplikácii refaktoringu a uvidíme ako sa zmeny prejavili.

celkový počet súborov	88
celkový počet riadkov zdrojového kódu	9189
celkový počet neprázdnych riadkov zdrojového kódu	6397
priemerný počet riadkov na jeden súbor	104

Tab. 7. Hodnoty hovoriace o celkovej veľkosti projektu

celkový počet súborov	51
celkový počet riadkov zdrojového kódu	4584
celkový počet neprázdnych riadkov zdrojového kódu	3060
priemerný počet riadkov na jeden súbor	88

Tab. 8. Hodnoty projektu *Business*

celkový počet súborov	35
celkový počet riadkov zdrojového kódu	4569
celkový počet neprázdnych riadkov zdrojového kódu	3322
priemerný počet riadkov na jeden súbor	130

Tab. 9. Hodnoty webového projektu *WebUI*

Vidíme, že sa nám zvýšil celkový počet súborov v projekte (Tab. 7.). Toto je spôsobené tým, že sme pridali *DataAccess* triedy pre niektoré biznis entity, vytvorili biznis triedy a ich biznis kolekcie pre entity, pre ktoré neexistovali a pracovalo sa s nimi v administračnom rozhraní pomocou priameho prístupu do databáze. Celkový počet riadkov v projekte sa ale znížil, tak ako aj priemerný počet riadkov na jeden súbor. Je to výsledok toho že sme odstránili množstvo duplicitného kódu. Projekt

Business od pôvodného stavu trochu narástol. Vznikli nové triedy na prístup do databáze (menný priestor *DataAccess*) a množstvo logiky bolo presunutej z prezentačnej vrstvy. Naopak vo webovom projekte *WebUI* počet riadkov kódu a počet súborov klesol (Tab. 9.), čo sme mohli aj čakať. Boli odstránené dve nepoužívané stránky a všetka logika bola presunutá do príslušných biznis tried. Použité boli refaktoringy *Extract method, Move method* a niekde aj *Parametrize method.* Nešlo o nič viac len o očistenie prezentačnej logiky od biznis logiky. Samotnej prezentačnej logiky sa refaktoring nedotkol aj keby si to zaslúžil. Kód je v dosť zlom stave a pýtalo by sa spraviť celkový redesign webovej časti. Ako som pri predstavení aplikácie spomenul, projekt slúžil na naučenie sa programovať v prostredí .NET a podľa toho kód aj vyzerá. Keďže ale Visual Studio.NET nedokáže merať metriky na webovej stránke, ďalším úpravám tohto projektu som sa nevenoval. Jediným výsledkom je teda zníženie počtu riadkov a tým sprehľadnenie a očistenie kódu.

Projekt *Business* okrem zmeny rozsahu pridaním nových tried a presunu logiky z prezentačnej vrstvy prešiel viacerými zmenami. Teraz sa pozrieme na hodnoty metrík (Tab. 10.), ktoré sme pomocou Visual Studia.NET merali na začiatku.

Trieda	**udržovateľnosť**	**cykl. zložitosť**	**úroveň dedičnosti**	**class coupling**	**počet riadkov**
PriceListColumnType	100	0	1	0	0
ExtraBedType	100	0	1	0	0
HotelCollection	100	1	2	2	1
CategoryCollection	100	1	2	2	1
DestinationCollection	100	1	2	2	1
PricelistRowCollection	100	1	2	2	1
NewsletterCollection	100	1	2	2	1
PricelistColumnCollection	100	1	2	2	1
CountryCollection	100	1	2	2	1
BusinessSystem	100	1	1	0	1
PricelistCollection	100	1	2	2	1
OrderCollection	100	1	2	2	1
ArticleCollection	100	1	2	2	1
OrderItemCollection	100	1	2	2	1
ExtraPaySystem	97	2	1	1	3
ExtraBed	95	2	3	1	2
BusinessObject	94	4	1	0	5
User	93	5	2	1	7
Country	93	3	3	1	3
Newsletter	93	7	1	0	8
PricelistValue	93	11	1	3	11
CommonBusinessObject	93	9	2	1	12
Hotel	92	5	3	2	6
Room	92	7	2	1	10
Category	92	8	3	1	10

Destination	92	6	3	2	7
ExtraPay	91	9	3	4	14
Article	91	13	2	2	19
PricelistColumn	91	14	3	2	17
OrderItem	91	10	2	2	14
Order	90	33	2	6	42
Pricelist	82	23	3	10	36
PricelistRow	76	14	2	3	35
PricelistValueCollection	73	6	2	6	8
CountrySystem	69	12	2	11	47
HotelSystem	69	11	2	13	50
DestinationSystem	69	15	2	12	52
UserSystem	69	3	2	7	11
PricelistSystem	68	70	2	28	224
CategorySystem	65	18	2	12	90
NewsletterSystem	65	10	2	11	45
ArticleSystem	61	22	2	12	116
OrderSystem	48	15	1	15	121
priemer	88	8,81	2	4,46	24,11
min	48	0	1	0	0
max	100	70	3	28	224

Tab. 10. Záverečné hodnoty metrík

Z predchádzajúcej tabuľky (Tab. 10.) je vidieť, že sa refaktoring pozitívne podpísal aj na týchto metrikách. Týka sa to hlavne tried, ktorých sa vykonané zmeny dotkli. Prvou metrikou je index udržovateľnosti, ktorej priemerná hodnota sa zvýšila. Za tento nárast môže refaktoring biznis kolekcií, refaktoring hierarchie tried a následné zmeny triedy *PricelistSystem*, ktorej hodnota sa zvýšila z 60 na 68. Priemerná hodnota tejto metriky sa zvýšila z pôvodných 85 na súčasných 88. Ďalšou a dôležitou metrikou je cyklomatická zložitosť. Zase sa jednalo o triedu *PricelistSystem,* ktorú sme na základe vysokej zložitosti refaktorovali. Z pôvodnej hodnoty 110 sme sa dostali na 70, čím bola aj ovplyvnená priemerná hodnota tejto metriky, ktorá je teraz 8,8 v porovnaní s 11,3 na začiatku. Ostatné metriky, už nie sú také zaujímavé. Po úprave hierarchie tried sa logicky zodvihla hodnota úrovne dedičnosti z 1,5 na súčasnú hodnotu 2. *Class coupling* sa v podstate nezmenil a počet riadkov som už okomentoval vyššie.

Pozrieme sa ešte na výstup analýzy programu FxCop (Tab. 11.). Chybami, ktoré boli výstupom tohto programu sme sa takisto zaoberali a predvádzali si refaktoring v praxi.

kritické chyby	0
chyby	62
kritické varovania	5
varovania	10

Tab. 11. Záverečný výstup z programu FxCop

Z prechádzajúcej tabuľky vidím, že aj tu je nejaké zlepšenie. V kapitole 6.3.4 sme šli cielene po opravách týchto chýb za účelom predviesť si refaktoring. Oprava všetkých chýb alebo aspoň čo najväčšieho počtu by taktiež posunula projekt, čo sa kvality týka ešte o kúsok ďalej.

Na prvý pohľad na dosiahnuté výsledky nejde o veľké zlepšenie, keď ale zoberieme do úvahy to, aké veľké úpravy sme spravili a na koľkých miestach, uvedomíme si, že ide o zlepšenie podstatné. Komplexnejší refaktoring, ktorý by bol aplikovaný na celý projekt by stav projektu niekoľko násobne zlepšil, o tom nie je pochýb. Refaktoring spojený s nástrojmi na to učenými a so sledovaním metrík je veľmi mocná technika, čo sme si ukázali v praxi a určite by sa dalo ísť ešte ďalej. Napríklad sústrediť sa na optimalizáciu kódu a sledovať ďalšie a ďalšie metriky, spolu s refaktoringom využiť návrhových vzorov alebo využiť možnosti ktoré poskytuje .NET 3.5. Na to však v tejto práci bohužiaľ nebol čas ani priestor. Nasleduje posledná kapitola práce a tou je záver, v nej zhodnotím splnenie požiadaviek na diplomovú prácu.

7 Záver

Predmetom diplomovej práce zoznámiť sa s problematikou refaktoringu objektovo orientovaných aplikácií a aplikovať znalosti na reálnu aplikáciu. V úvode práce som vysvetlil pojem refaktoring ako aj jeho základné princípy, výhody a nevýhody. Zodpovedal som na základné otázky akými sú, prečo refaktorovať, čo refaktorovať, kedy refaktorovať a kedy naopak nerefaktorovať. Ďalej som v teoretickej rovine vysvetlil základné a používané vzory refaktoringu v kapitole nazvanej *Katalóg refaktoringov*. Tento katalóg však nie je kompletný, nie je to v rozsahu, ktorý je určený na diplomovú prácu možné. Ďalej som sa v skratke popísal platformu .NET a nástroje, ktoré sú na podporu automatického refaktoringu na tejto platforme k dispozícii. V predposlednej kapitole som sa venoval použitím získaných znalostí na reálnej aplikácii, na ktorej som ukázal, čo sa refaktoringom dá dosiahnuť. Popísal som niekoľko krokov skvalitnenia aplikácie s použitím niektorých refaktoringov. V závere kapitoly som zhodnotil dosiahnuté výsledky.

Počas písania práce som zistil, že je daná téma veľmi rozsiahla a zasahuje aj do iných oblastí ako napríklad návrhové vzory. Nebolo teda možné v práci uviesť všetko, myslím si však, že som obsiahol to podstatné a tým som ako rozsahom tak aj obsahom práce splnil požiadavky diplomovej práce.

Literatúra

[1] Fowler, M.: Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.

[2] Gamma, E., R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, 1995.

[3] Cwalina, K., B. Adams: Framework Design Guidelines, Addison-Wesley, 2005.
ISBN 0321246756.

[4] http://en.wikipedia.org/wiki/Cyclomatic_complexity

[5] http://en.wikipedia.org/wiki/Microsoft_.NET#Microsoft_.NET

Zoznam príloh

Príloha 1. CD obsahujúce elektronickú podobu diplomovej práce vo formátoch .pdf, .doc a .docx a kompletné zdrojové texty aplikácie pred a po refaktoringu.

Adresárová štruktúra CD:

- adresár *Dokumenty* obsahuje text diplomovej práce
- adresár *SourceCode_before* obsahuje zdrojové texty pôvodnej aplikácie
- adresár *SourceCode_after* obsahuje zdrojové texty aplikácie po refaktoringu