Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informační systémy a technologie

Refaktoring v PHP

DIPLOMOVÁ PRÁCE

Student : Bc. Jiří Martišek

Vedoucí : Ing. Jan Mittner

Oponent : doc. Ing. Alena Buchalcevová, Ph.D.

2013

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 30. 11. 2013 . . . . . .

Jiří Martišek

Poděkování

Děkuji tímto svému vedoucímu Ing. Janu Mittnerovi za ochotu vést moji práci, za mnoho cenných připomínek a také za nezanedbatelné množství času, které si pro konzultace dokázal nalézt.

Abstrakt

Tato práce se zabývá návrhem metodiky refaktoringu webových aplikací v PHP. Hlavním cílem práce je navrhnout tuto metodiku a ukázat její praktickou aplikaci na reálném projektu.

V první kapitole vymezuji cíle, omezení a očekávané přínosy práce. Ve druhé pak uvádím výsledky průzkumu existující literatury na podobná témata. Ve třetí kapitole pak komentuji přístup některých metodik vývoje softwaru k problematice refaktoringu.

Ve čtvrté kapitole uvádím obecně známé pachy a přidávám další, obvyklé v PHP. Také si všímám specifik refaktoringu v PHP.

V páté kapitole shrnuji požadavky na metodiku, předpoklady jejího použití, její principy a obecné vlastnosti. Dále uvádím doporučený postup refaktoringu webové aplikace, jehož jednotlivé části jsou okomentovány dále v kapitole. Jedná se především o způsob identifikace oblastí k refaktoringu, jejich prioritizace a následně praktik pro samotný refaktoring. V závěru uvádím tipy pro zavedení předkládané metodiky do praxe.

V šesté kapitole ukazuji praktickou aplikaci metodiky na reálném projektu. Komentuji jednotlivé kroky provedené dle metodiky a následně postřehy získané při vlastním refaktoringu.

Klíčová slova

refaktoring, PHP, kvalita kódu

Abstract

This thesis deals with a methodology for PHP web applications refactoring. The main objective is to propose such a methodology, and to apply it on a real project.

The first chapter contains objectives, restrictions and contributions of the thesis. The research of the existing literature on similar topics follows in the second chapter. The third chapter comments on selected software development methodologies' approach to refactoring.

The fourth chapter describes known code smells, and adds some new ones appearing particularly in PHP. Specifics of PHP refactoring are also described.

The fifth chapter deals with the methodology proposal itself. It sums up requirements, deployment assumptions, the methodology principles and general characteristics. The next part contains the recommended procedure of PHP web applications refactoring with its parts commented farther on. These parts are mainly code smells identification techniques, prioritization and finally best practices. The chapter ends with this methodology deployment recommendations.

The sixth chapter shows a practical usage of the proposed methodology when refactoring a real web application. The performed steps are explained in terms of the methodology. The chapter closes with observations gained during this refactoring.

Keywords

refactoring, PHP, code quality

Obsah

1 Úvod

Internet hraje v našich životech stále větší roli. Nejmladší generace už si možná život bez internetu nedovede ani představit. Stále důležitější oblastí vývoje softwaru se stávají webové aplikace. Běžný uživatel se s nimi setkává na každém kroku, a tak není divu, že roste i počet webových vývojářů. Nejrozšířenějším jazykem vývoje webu je PHP[1] [W3Techs, 2013] a v poslední době je mu věnována velká pozornost.

Čemu ovšem není věnováno příliš pozornosti, to je psaní kvalitního kódu a jeho údržba. V kombinaci s poměrně častou nezkušeností vývojářů PHP (pro mnohé je to jejich první programovací jazyk) to pak vede k nedostatkům v jejich produktech.

Tato práce přispívá ke zvýšení povědomí o tomto problému a především nabízí praktické řešení.

1.1 Vymezení tématu práce

V této práci se zabývám refaktoringem kódu v jazyce PHP v kontextu webových aplikací. Refaktoringem se myslí změny struktury kódu za účelem jeho kvalitativního zlepšení, přičemž se nemění jeho vnější chování [Fowler, 1999].

Konkrétně navrhuji metodiku takového refaktoringu s ohledem na stávající principy a praktiky refaktoringu. V praktické části práce tuto metodiku aplikuji na konkrétní webovou aplikaci. Metodikou rozumím souhrn metod a postupů pro realizaci určitého úkolu, podle [Buchalcevová, 2009]. Této jednoduché definice se budu v této práci držet. Je ovšem třeba zdůraznit, že se nejedná o metodiku vývoje informačních systémů, tato metodika se zaměřuje pouze na refaktoring.

V celé práci používám slovo *refaktoring*, v české literatuře se ovšem také někdy používá slovo *refaktorizace*. Obojí vychází z anglického *refactoring* a význam je totožný.

[1] PHP – PHP: Hypertext Preprocessor

Dále se lze v literatuře setkat s pojmem *mikrorefaktoring*, což je pokus o odlišení označení pro jeden typ restrukturalizace kódu od označení pro celý proces zkvalitňování kódu neboli sérii takových restrukturalizací. V této práci používám jednoduše pouze pojem *refaktoring*, jelikož z kontextu vždy vyplyne, zda je tím myšlen celkový proces, či pouze jednotlivý typ zkvalitnění (restrukturalizace) kódu. Výjimečně se lze také setkat s mírně zavádějícím pojmem *refactoring patterns*, kterým ale mohou být označeny také doporučené postupy (v této práci *Praktiky*).

1.2 Důvod výběru tématu

Toto téma jsem si vybral, jelikož je mi tato problematika blízká z praxe. Všiml jsem si, že knih o refaktoringu je mnoho, ovšem téměř žádné práce se nevěnují refaktoringu webových aplikací v PHP. Navíc chybí návod (nebo tzv. „kuchařka“), podle kterého by mohli vývojáři k této problematice přistupovat. Rozhodl jsem se tento prostor zaplnit v této práci.

Navíc je třeba uvést, že refaktoring se dle mých osobních poznatků podceňuje (zvláště platící zákazník pro něj často nemívá pochopení, jelikož mu to nepřináší novou funkcionalitu, ale stojí ho to peníze). Chtěl bych touto prací poukázat na jeho důležitost.

1.3 Cíle práce

Hlavním cílem práce je navrhnout a prakticky ověřit metodiku refaktoringu PHP kódu webových aplikací, která by byla prakticky využitelná a aplikovatelná i ve stávajících projektech.

Dílčím cílem je prozkoumat a zohlednit dostupné poznatky o refaktoringu a existující metodiky o vývoji softwaru. V neposlední řadě využiji také svých poznatků z praxe.

Dalším dílčím cílem práce je ukázat aplikaci této metodiky na projektu z praxe. K tomu využiji webovou aplikaci Doučka.cz, jejímž jsem hlavním autorem a která s léty vývoje a právě nedostatkem refaktoringu ztratila na přehlednosti.

1.4 Předpoklady a omezení práce

V práci se zaměřím pouze na refaktoring webových aplikací psaných v jazyce PHP. K aplikaci patří většinou i databáze, o jejich refaktoringu ale existují jiné vhodné zdroje (viz

dále). Refaktoringem front-endu (typicky HTML[2], CSS[3], JavaScript,…) se v této práci také nezabývám, jelikož to přesahuje rámec tématu této práce.

1.5 Struktura práce

Struktura práce je následující. Nejprve zmíním důležitou literaturu, poté prozkoumám vztah existujících metodik vývoje softwaru k refaktoringu a navrhnu vlastní metodiku. V další části práce pak představím projekt Doučka.cz a popíšu praktickou aplikaci vytvořené metodiky při jeho refaktoringu. Na konci zhodnotím aplikovatelnost metodiky v praxi a uvedu příležitosti k jejímu dalšímu rozvoji.

1.6 Výstupy práce a očekávané přínosy

Výstupem práce tedy bude funkční a aplikovatelná metodika refaktoringu v PHP, včetně ukázky její aplikace. Práce bude přínosem pro PHP vývojáře, kteří dosud nemají explicitně definované postupy a praktiky refaktoringu. Aplikováním této metodiky budou moci dosáhnout efektivnějšího vývoje.

1.7 Vlastní přínos

Dostupné zdroje zpracovávají refaktoring vždy velmi podobným způsobem. Zaměřují se na katalog typů problematických míst v kódu a katalog způsobů zkvalitnění kódu (stručně je uvádím i v této práci). Neexistuje ale metodika, podle které by bylo možno při refaktoringu webových aplikací postupovat.

Mým přínosem v této práci je tedy zaplnění tohoto prostoru. Návrhem metodiky poskytnu vývojovým týmům „návod“, jak nasadit refaktoring ve svých projektech. Zmiňuji i nejrůznější aspekty jako např. rizika refaktoringu, což z této práce činí zajímavý souhrnný zdroj informací o refaktoringu pro praxi.

Ukázka praktické aplikace metodiky pak může být prospěšná všem, kteří preferují kromě teorie také praktické příklady.

[2] HTML – HyperText Markup Language
[3] CSS – Cascading Style Sheets

2 Rešerše literatury

V této části uvádím několik prací různých zaměření: a) práce o refaktoringu obecně, b) práce o refaktoringu v PHP, c) práce o refaktoringu databází, d) práce o kvalitním kódu, e) práce o návrhu metodiky.

Literaturu zde uvedenou lze doporučit k prostudování před aplikací metodiky navrhované v této práci.

2.1 Práce o refaktoringu obecně

Refactoring: improving the design of existing code [Fowler, 1999]

Prací o refaktoringu obecně je poměrně dost. Všechny se ale odkazují na tuto knihu, která se stala zásadní publikací pro tuto oblast. Zavádí pojem refactoring, popisuje principy a praktiky refaktoringu a poskytuje rozsáhlý katalog jednotlivých mikro-refaktoringů. U každého popisuje přesný postup, jak jej bezpečně provést. Celou kapitolu také věnuje testování, mj. popisuje důvody pro jeho nezbytnost při refaktoringu.

Tato kniha je stěžejní pro jakýkoliv refaktoring. Je do velké míry nezávislá na jazyce, předpokládá se pouze použití objektově orientovaného jazyka (vzhledem k většině popisovaných refaktoringů). Samotné příklady jsou psány v Javě, ovšem lze je dobře převést i do PHP.

Vymezení vůči této práci:
Tato kniha neobsahuje konkrétní návod (metodiku), spíše popisuje principy a postupy a nabízí také již zmiňované katalogy. Méně zkušený čtenář může tápat, jak má v praxi postupovat u konkrétního projektu. Obsahuje mnoho užitečných praktik, které přebírám také do navrhované metodiky. Na rozdíl od této práce se vůbec nezabývá webovými aplikacemi ani PHP – chce být obecná s mírným příklonem k jazyku Java.

Refactoring to patterns [Kerievsky, 2004]

Fowler [1999] se v celé knize odvolává na návrhové vzory dle knihy [Gamma et. al., 1994]. Ovšem Kerievsky jde v tomto ohledu dále a klade na návrhové vzory mnohem větší důraz.

Naopak nevěnuje tolik prostoru refaktoringům, které nemají s návrhovými vzory co do činění. Tuto knihu zde uvádím proto, že bývá také často citována.

Vymezení vůči této práci:

Ani tato kniha neobsahuje konkrétní návod (metodiku). Stejně jako předchozí kniha obsahuje spíše katalog, byť tentokrát zaměřený na návrhové vzory. A stejně jako předchozí kniha se vůbec nezaobírá specifiky webových aplikací.

Working effectively with legacy code [Feathers, 2005]

Tato kniha také pojednává o refaktoringu, ovšem z velmi praktického hlediska. Adresuje problémy, které mohou programátorovi nastat poté, co dostane na starost údržbu kódu, jehož autorem je někdo jiný. Třetí část knihy obsahuje již „tradiční" katalog refaktoringů, podobně jako [Fowler, 1999].

Ostatní části knihy popisují pravidla, principy a rady pro úpravy aplikace na vysoké úrovni („high-level"), mnoho z toho se dá také považovat za refaktoring (podle jeho definice).

Celkově je tato kniha velice užitečná vlastně pro každého programátora, jelikož každý se někdy dostane do situace, že musí pracovat s kódem, který napsal on sám před několika měsíci či roky.

Vymezení vůči této práci:

Tato kniha na rozdíl od obou předchozích poskytuje jak základní principy a praktiky, tak i jakýsi návod pro refaktoring, to vše v kontextu zastaralého (*legacy*) kódu. Opět to ale není uceleným návodem, jaký poskytuji v navrhované metodice. Navíc kvůli specifickému zaměření chybí mnohé praktiky a přístupy užitečné pro refaktoring za jiným účelem, než je převzetí zastaralého kódu.

2.2 Práce o refaktoringu v PHP

Pro PHP Refaktoring [Trucchia et.al, 2010]

Tato kniha je jedinou knihou o refaktoringu v PHP, kterou se mi povedlo nalézt. Vysvětluje principy a praktiky refaktoringu, zdůrazňuje význam testování, zabývá se také starým kódem a refaktoringem zapojujícím návrhové vzory. Většinu knihy ale tvoří opět katalog refaktoringů.

Pro PHP vývojáře ji lze doporučit možná více než Fowlera [1999], jelikož kniha obsahuje vše důležité, příklady jsou v PHP a kapitola o testování uvádí nástroje PHPUnit a Selenium.

Jelikož názvem je tato kniha nejpodobnější této práci, explicitně zmíním největší rozdíly. Cílem této práce je navrhnout a aplikovat metodiku pro refaktoring v PHP s ohledem na dodržování principů a pravidel refaktoringu. Neobsahuje katalog refaktoringů, nesnaží se zahrnout vše potřebné, spíše odkazuje na další zdroje.

Vymezení vůči této práci:

Tato práce jako jediná ze zde uvedených popisuje refaktoring v kontextu webových aplikací a PHP. Na druhou stranu, poměrně striktně se drží stejně struktury jako prvně citovaná kniha [Fowler, 1999], tedy tamní vymezení platí i zde.

2.3 Práce o refaktoringu databází

Refaktoring PHP kódu souvisí do velké míry s refaktoringem databází, jelikož mezi těmito dvěma vrstvami (aplikace a úložiště dat) je vzájemná provázanost. Změny v PHP kódu mohou vést k požadavkům na změny struktury databáze. Proto zde stručně zmíním i práce zabývající se refaktoringem databází.

Refaktoring databází [Paulavets, 2012], Agilní správa databáze [Kotyza, 2009]

Obě práce uvádějí katalog refaktoringů. Druhá z nich ovšem obsahuje poněkud širší pohled na tuto problematiku, jelikož navíc uvádí principy a praktiky, které lze využít.

Obě práce jsou vhodným zdrojem více informací, jelikož tato práce se problematikou refaktoringu databází nezabývá.

2.4 Práce o kvalitním kódu

Základem nutným pro refaktorování je ovšem ovládnout zásady pro psaní kvalitního kódu. Na toto téma lze doporučit např. knihu [Martin, 2009] nebo ještě obsáhlejší [McConnell, 2004]. Stručnější alternativou je webová prezentace předmětu Doporučené postupy v programování vyučovaného na Matematicko-fyzikální fakultě Univerzity Karlovy [Doporučené postupy v programování, 2009], která vychází převážně z druhé citované knihy.

S návrhem aplikací pak mohou pomoci návrhové vzory. Informace o nich jsou v češtině např. v [Pecinovský, 2007], v angličtině pak doporučuji velice čtivou knihu [Freeman, 2004],

případně lze prostudovat původní knihu [Gamma et.al., 1994], byť ta není tak čtivá, o to je ale podrobnější a preciznější. Ve všech lze nalézt i základní principy objektově orientovaného programování (OOP), ze kterých návrhové vzory vycházejí a kterých je radno se držet, aby vznikal kvalitnější kód. Je třeba se jich samozřejmě držet i při refaktoringu, protože ten přichází na řadu, když napsaný kód není dostatečně kvalitní, což je přirozené, i pokud máme principy OOP neustále na paměti.

Tato práce není primárně o tom, jak tvořit kvalitní design a psát kvalitní kód, byť jsou tato témata úzce spojena. Uvedené knihy se zase nezaměřují na to, jak postupovat při zkvalitňování již napsaného kódu. Výjimečně lze sice něco nalézt, ale nikdy to není metodika, natož návod, jak konkrétně postupovat.

2.5 Práce o návrhu metodiky

Poslední typ sem patří jen velmi okrajově, jelikož hlavním cílem této práce je navrhnout metodiku pro refaktoring v PHP. Přesto si myslím, že by zde neměl chybět.

Metodiky vývoje a údržby informačních systémů [Buchalcevová, 2005], Metodiky budování informačních systémů [Buchalcevová, 2009]

Tyto práce obsahují informace o charakteristikách metodik a rozšířených existujících metodikách. Přestože jsou tyto informace vztahovány k vývoji informačních systémů, lze se inspirovat i při tvorbě metodik zaměřených na menší oblasti vývoje, jako např. metodiky refaktoringu v PHP.

Na rozdíl od této práce se výše uvedené ani v nejmenším nezabývají refaktoringem, ani vývojem webových aplikací. Určitý průnik s touto prací je pouze v oblasti charakterizace a strukturace metodik.

2.6 Shrnutí

Nikde se mi nepovedlo nalézt práci zaměřenou na metodiku refaktoringu. Vždy je to spíše výčet principů a následně konkrétních praktik, zahrnuty bývají i rady, kdy co použít. Proto bych rád tento prostor zaplnil a poskytl vývojářům „kuchařku“, podle které mohou svůj PHP projekt refaktorovat.

3 Přístup vybraných metodik k refaktoringu

Inspirací k vytvoření metodiky mohou být kromě prací o refaktoringu také již existující metodiky vývoje softwaru. Záměrně píši „inspirací“, jelikož se mi nepovedlo nalézt metodiku zaměřující se přímo na refaktoring. V této kapitole tedy uvádím, zda a jak se o refaktoringu zmiňují vybrané metodiky vývoje softwaru.

Jelikož je navrhovaná metodika agilní (viz dále), zaměřil jsem se především na agilní metodiky vývoje softwaru. Jako kritérium výběru jsem použil výsledky průzkumu jejich rozšířenosti [Version One, 2012]. Na základě toho jsem vybral metodiky Scrum a Extreme Programming (XP), které jsou používány více než v 10% případů. Navíc jsem přidal metodiku OpenUP, která je zajímavá svou poměrně volnou a přizpůsobitelnou strukturou.

3.1 Scrum

V oficiální příručce k této nejrozšířenější agilní metodice [Schwaber, 2013] ani na oficiálním webu *scrum.org* nejsou žádné zmínky o refaktoringu. Není to ale příliš překvapivé, protože Scrum se zabývá spíše organizací práce než samotným kódem.

Existují ale knihy, které se zabývají agilním vývojem či přímo metodikou Scrum a které refaktoring explicitně uvádějí a výrazně doporučují jej nezanedbávat. Příkladem mohou být [Cohn, 2010] a [Chris, 2011].

3.2 Extreme programming [Beck, 2005]

Tato agilní metodika bere refaktoring jako nezbytnou součást vývoje. Stručně řečeno, vývojář nejprve napíše jednotkové testy (*unit tests*) a pak dopisuje kód tak, aby těmito testy prošel. Když už nedokáže přidat test, který by neuspěl, kód je hotov. S tím souvisí také

princip nejjednoduššího řešení, tzn. naprogramovat nejjednodušší možné řešení, které projde testy.

Vývojář tedy má řešení, které projde stávajícími testy, pak přidá další testy, ten neuspěje a cílem vývojáře je doplnit či přepsat stávající řešení tak, aby uspěly všechny testy. Z tohoto přístupu tedy vyplývá stálá potřeba refaktoringu při vývoji.

I při vývoji touto metodikou samozřejmě je někdy třeba provést větší refaktoring, ostatně jako v každém softwarovém projektu. Výhodou extrémního programování je absolutní nezbytnost mít vše podložené testy, takže refaktoringu se vývojáři nemusejí obávat, jelikož postupují po malých krocích a po každém z nich nechají proběhnout testy. Velice snadno totiž odhalí chyby, kterých se při refaktoringu mohli dopustit.

Pokud tým přebírá starý kód, který nemá jednotkové testy, pak je podle této metodiky třeba je dopsat dříve, než se začne s refaktoringem či opravou nějakých chyb.

3.3 OpenUP [OpenUP, 2012]

Tato metodika deklaruje refaktoring jak průběžně během vývoje, tak i při kontrole designu pro jeho vylepšení. Rozlišuje pojmy *code refactoring* a *design refactoring*. Zmiňuje několik důvodů pro refaktoring a několik způsobů nalezení problémových míst (míst k refaktoringu). Metodika navrhovaná v této práci je v tomto podstatně detailnější.

OpenUP také vyžaduje plnou sadu vývojových testů, aby bylo možné refaktorovat bezpečně.

3.4 Shrnutí

Metodiky vývoje softwaru jsou zaměřeny na vývoj na úrovni projektu, zatímco metodika navrhovaná v této práci funguje na nižší úrovni, pro specifickou část vývoje. Proto není možné se o ně opřít či je nějak rozšířit.

I tak je ale možné nalézt určitou inspiraci v Extreme Programming a OpenUP, především pro podkapitolu *Praktiky* (viz dále).

4 O refaktoringu v kontextu PHP

PHP je dynamický a také dynamicky typovaný programovací jazyk, což mimo jiné znamená určování typu proměnných až za běhu, nikoli při kompilaci. Spolu s různými vlastnostmi syntaxe jazyka to může vést ke specifickým problémům s kvalitou kódu, které se nevyskytují např. v Javě, kterou používá [Fowler, 1999] ve svém výkladu a příkladech. Některé zvláštní nebo neintuitivní vlastnosti jazyka dobře vystihuje [Munroe, 2012][4]. Proto se v této části zaměřím na některé tyto problematické vlastnosti a z nich vznikající pachy (včetně návrhu jejich řešení).

Výše uvedeným nijak nesnižuji kvalitu jazyka. I v PHP se dá psát čistý a kvalitní kód. Chci tím ale poznamenat, že PHP na druhou stranu umožňuje také psát kód velice nekvalitní, a to snáze než např. Java.

Situaci neulehčuje ani to, že vývojáři zpravidla nemají kontrolu nad konfigurací a verzí PHP na hostingu. Je tedy důležité dávat pozor na případné změny těchto parametrů na hostingu a vhodně reagovat. Proto je údržba běžící aplikace nutná, byť změny verzí nejsou časté a konfigurace PHP se na hostingu nemusí změnit nikdy.

4.1 Pachy v kódu obecně

Oblasti vhodné pro refaktoring označuje [Fowler, 1999] podle Kenta Becka jako *pachy v kódu* (*code smells*) a jelikož se toto označení ujalo, používám jej i já v této práci. Citovaná kniha nabízí seznam takových pachů, uvádím jej zde s doplněnými stručnými popisy. V posledním sloupci jsou pak navrhované refaktoringy. Pro více detailů a konkrétní postupy u jednotlivých refaktoringů doporučuji prostudovat citovanou knihu.

V *Tabulce 4-1* používám český překlad citované knihy [Fowler, 2003, přebal].

[4] Na tento článek také zajímavě reaguje Jakub Vrána: http://php.vrana.cz/php-a-fractal-of-not-so-bad-design.php

Tabulka 4-1: Seznam pachů (zdroj: [Fowler, 2003])

Pach	Popis	Obvyklé refaktoringy
Alternativní třídy s různými rozhraními	Metody dělající stejnou věc mají různou signaturu, což snižuje čitelnost.	Přejmenovat metodu Přesunout metodu
Komentáře	Komentář by neměl suplovat kvalitu kódu. Často se mu lze vyhnout refaktoringem dané oblasti. Komentář popisující CO kód dělá, nebývá v pořádku, vhodný je naopak komentář popisující PROČ.	Vyjmout metodu Zavést předpoklad
Datová třída	Třída obsahující pouze data je pravděpodobně příliš manipulována jinými třídami, místo aby tato funkcionalita byla zahrnuta v ní, tzn. u dat, se kterými se pracuje.	Přesunout metodu Zapouzdřit položku Zapouzdřit kontejner
Datové shluky	Více souvisejících datových položek je lépe sdružit do jednoho objektu.	Vyjmout třídu Zavést objekt pro parametry Zachovat celý objekt
Protichůdné změny	Nutnost různých změn v třídě z různých důvodů. Je příznakem, že toho třída dělá příliš mnoho.	Vyjmout třídu
Duplicitní kód	Totožný či velmi podobný kód na více místech aplikace. Znesnadňuje provádění změn.	Vyjmout metodu Vyjmout třídu Přesunout metodu výš Vytvořit šablonovou metodu
Chybějící schopnosti	Metoda třídy příliš často interaguje s jinou třídou, zpravidla kvůli jejím datům.	Přesunout metodu Přesunout položku Vyjmout metodu
Nevhodná důvěrnost	Jedna třída ví o druhé příliš mnoho. Porušuje se tím princip zapouzdření, zvyšuje provázanost a zvyšuje obtížnost změny.	Přesunout položku Přesunout metodu Změnit obousměrné propojení na jednosměrné Nahradit dědičnost delegováním Skrýt delegáta
Neúplná knihovní třída	Funkcionalita knihovní třídy je nedostatečná pro daný úkol v kódu a je třeba ji rozšířit.	Zavést cizí metodu Zavést místní rozšíření
Velká třída	Třída toho dělá příliš mnoho. Každá třída by měla mít jen jednu odpovědnost.	Vyjmout třídu Vyjmout podtřídu Vyjmout rozhraní Nahradit datovou položku objektem
Líná třída	Třída toho nedělá dost, není	Vložit třídu

	potřeba.	Zrušit hierarchii
Dlouhá metoda	Čím delší metoda, tím méně přehledná a hůře testovatelná.	Vyjmout metodu Nahradit dočasnou proměnnou dotazem Nahradit metodu objektem metody Rozložit podmínku
Dlouhý seznam parametrů	Příliš mnoho parametrů metody snižuje přehlednost; obecně platný přesný limit je ovšem těžké stanovit.	Nahradit parametr metodou Zavést objekt pro parametry Zachovat celý objekt
Zřetězené zprávy	Objekt volá jiný objekt, ten zase jiný objekt a tak dále. Výsledkem je vyšší provázanost objektů a tedy obtížnější změny.	Skrýt delegáta
Prostředník	Metody třídy-prostředníka volají metody jiné třídy v příliš velké míře, kdy už se vyplatí volat metody druhé třídy přímo.	Odstranit prostředníka Vložit metodu Nahradit delegování dědičností
Paralelní hierarchie dědičnosti	Speciální případ pachu *Rozptýlené úpravy*. Příznakem je, že po vytvoření podtřídy nějaké třídy je třeba vytvořit podtřídu nějaké jiné.	Přesunout metodu Přesunout položku
Primitivní obsese	Používání primitivních datových typů, přestože by bylo vhodnější vytvořit samostatnou třídu.	Nahradit datovou položku objektem Vyjmout třídu Zavést objekt pro parametry Nahradit pole objektem Nahradit kód typu třídou Nahradit kód typu podtřídami Nahradit kód typu stavem nebo strategií
Odmítnuté dědictví	Podtřída nepoužívá velkou část funkcionality svého předka. Může to znamenat nedostatky v hierarchii. Nemusí to ale být závažný problém, pokud ovšem zároveň neodmítá jeho rozhraní.	Nahradit dědičnosti delegováním
Rozptýlené úpravy	K provedení změny je třeba editovat kód na mnoha různých místech. To snižuje udržovatelnost aplikace a tím i efektivitu vývoje.	Přesunout metodu Přesunout položku Vložit třídu
Spekulativní obecnost	Způsob návrhu, kdy se příliš myslí na to, jaké požadavky na třídu/metodu by mohly v budoucnu přibýt. Výsledkem je zbytečná složitost, která se často ani nevyužije.	Zrušit hierarchii Vložit třídu Odstranit parametr Přejmenovat metodu

Příkazy switch	Příkazy switch jsou příznakem nedostatečného využití možností objektově orientovaného programování. Navíc často obsahují duplicitní kód.	Nahradit podmínku polymorfismem Nahradit kód typu podtřídami Nahradit kód typu stavem nebo strategií Nahradit parametr explicitními metodami Zavést objekt null
Dočasná položka	Atribut objektu, který je využíván jen někdy, což může být matoucí.	Vyjmout třídu Zavést objekt null

4.2 Pachy v kódu specifické pro PHP

Dále uvádím pachy vyskytující se poměrně často v PHP aplikacích a k nim korespondující refaktoringy. Některé jsou specifické pro PHP, některé lze (v případných obměnách) nalézt i v jiných jazycích. Ve srovnání s výše uvedenými pachy jsou tyto vesměs na nižší úrovni abstrakce, tzn. na úrovni konkrétních zápisů v kódu. Je to dáno tím, že výše uvedené pachy jsou platné obecně pro objektově orientované programování, zatímco následující pachy jsou specifické pro PHP a ty obecné rozšiřují.

Příliš mnoho PHP v HTML

Popis: V kódu HTML se nachází nejen PHP výpisy a jejich řídicí smyčky (např. při výpisu seznamu záznamů), ale také aplikační logika, výpočty apod.

Problém: Míchání prezentační vrstvy s vrstvou aplikační logiky s sebou přináší horší čitelnost a udržovatelnost. Při změně logiky je třeba zasahovat do vzhledu a opačně.

Refaktoring: Oddělit prezentační vrstvu a vrstvu aplikační logiky.

Příklad: Návrhový vzor MVC (model-view-controller) definuje *model* jako vrstvu aplikační logiky, *views* jsou prezentační šablony (do nich se vkládají jednoduché PHP výpisy) a *controller* obě tyto vrstvy spojuje a řídí interakci s uživatelem.

Nedůsledné dodržování typů proměnných

Popis: Tatáž proměnná je na různých místech kódu použita jako různý datový typ.

Problém: PHP je dynamicky typovaný jazyk, což znamená určování typu proměnné za běhu a implicitní konverze datových typů. Ovšem tyto konverze mohou mít jiný než intuitivní výsledek, což může vést k těžko odhalitelným chybám.

Refaktoring: Dodržovat typ proměnné. Je-li nutné typ změnit, udělat to explicitně. Proměnná může dělat jen jednu věc.

Příklad: *$a="6"; $a==" 6"* // porovnání vrátí *TRUE*
Zdůrazňuji mezeru v druhých uvozovkách (jelikož operátor == převádí operandy na čísla, pokud je to možné [Munroe, 2012]).

Porovnávání bez kontroly datového typu

Popis: Souvisí s předchozím, příčinou je opět dynamické typování. Operátor == kontroluje jen hodnoty, nikoli typ operandů, navíc může provádět implicitní konverzi.

Problém: Stejný jako výše. Implicitní konverze mohou mít neintuitivní výsledek, což může vést k těžko odhalitelným chybám.

Refaktoring: Používat operátor === místo ==, kde je to možné. Operátor === kontroluje kromě hodnoty také typ operandů. Proto zabraňuje implicitní konverzi při porovnávání.

Příklad: *„false“ == true* a *„false“ == 0* (ale *true != 0*)

Předávání parametru odkazem

Popis: Při předání parametru do funkce/metody odkazem se změna jeho hodnoty uvnitř funkce projeví i vně na proměnné, která do funkce vstupuje.

Problém: Snížená čitelnost, vyšší riziko chyby. V místě volání funkce/metody není na první pohled vidět, co se děje a že proměnná bude změněna.

Může se ovšem stát, že výhody převáží výše uvedené nevýhody. Pak lze parametr odkazem předat, ale je třeba to dobře zdokumentovat a ideálně i v místě volání nějak označit (přinejmenším komentářem).

Refaktoring: Předávat parametry hodnotou a výsledek vracet z funkce konstruktem *return*. V případě potřeby vrácení více hodnot lze vrátit objekt, případně pole (méně vhodné, viz pach *primitivní obsese*). Ve funkci/metodě je z hlediska čitelnosti vhodnou praktikou parametry vůbec nemodifikovat.

Příklad: PHP funkce *bool sort (array &$array [, int $sort_flags = SORT_REGULAR])*.[5]

[5] Tato funkce v oficiální PHP dokumentaci: http://us1.php.net/sort

Globální proměnné

Popis: Globální proměnná je proměnná definována vně funkce (oproti lokální proměnné, která je definována a přístupná pouze uvnitř jedné konkrétní funkce).

Problém: Používáním globálních proměnných se snižuje modularita kódu (jednotlivé části jsou více provázané), což má negativní vliv na čitelnost a znovupoužitelnost.

Refaktoring: Pokud možno nepoužívat globální proměnné. Proměnnou předávat parametrem a její novou hodnotu vracet standardně pomocí „return“. Je-li třeba vrátit z funkce více hodnot, je vhodné zvážit vrácení těchto hodnot v objektu, případně v poli.

Příklad:

Kód 4-1: Příklad globální proměnné (zdroj: dokumentace PHP[6])

```
<?php
$a = 1; /* globální proměnná */
function test()
{
    // volání lokální proměnné (která není totožná s globálním $a výše)
    echo $a;
}
test();
?>
```

Klíčové slovo global

Popis: Pokud chci ve funkci použít proměnnou „zvenčí“ (tedy globální proměnnou), kterou ovšem nepředám parametrem, je třeba ji na začátku funkce označit slovem klíčovým „global“.

Problém: Toto zásadním způsobem narušuje princip fungování funkcí, přidává to do nich vedlejší efekty. Pokud funkce může měnit hodnotu jiné proměnné, aniž by o tom volající věděl, může to vést k těžko očekávatelným důsledkům a záludným chybám. Navíc je pak obtížné takovou funkci testovat.

Refaktoring: Stejně jako v předchozím případě.

[6] Příklad převzat z dokumentace PHP: http://www.php.net/manual/en/language.variables.scope.php

Příklad:

Kód 4-2: Příklad použití klíčového slova *global* (zdroj: dokumentace PHP[7])

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    global $a, $b;

    $b = $a + $b;
}

Sum();
echo $b; // vypíše 3
?>
```

Potlačení chyb pomocí @

Popis: Napsáním znaku „@“ před libovolný výraz se potlačí chybové hlášení, které prováděním tohoto výrazu může vzniknout.

Problém: Toto není ošetření chyb, ale jejich potlačení, k chybám dochází i tak, jen nejsou vidět. Z toho vyplývá, že skript se může chovat jinak, než vývojář čeká – což ztěžuje ladění chyb. Vývojář by měl chybám předcházet, zejména testováním v kódu (např. před přístupem na určitý index v poli ověří, že tento index existuje).

Refaktoring: Odstranit znak „@“ a chybám předcházet (testováním v kódu, viz výše).

Příklad:

Kód 4-3: Příklad potlačení chyby (zdroj: dokumentace PHP[8])

```
// pokud index v poli neexistuje, nenahlásí žádnou chybu
$value = @$cache[$key];
```

Příkaz goto

Popis: Konstrukt goto umožňuje přesunout se ve vykonávání skriptu na jiné místo.

Problém: Výrazně snižuje modularitu kódu a tím i jeho čitelnost.

[7] Příklad převzat z dokumentace PHP: http://php.net/manual/en/language.variables.scope.php.
Více v dokumentaci PHP: [8] http://php.net/manual/en/language.operators.errorcontrol.php

Refaktoring: Nepoužívat. Vždy se dá kód napsat tak, aby goto nebylo třeba.

4.3 Shrnutí

V této kapitole jsem se zaměřil na specifika refaktoringu v kontextu PHP. Nejprve jsem vypsal pachy a jejich řešení podle [Fowler, 2003], dále jsem přidal pachy vyskytující se často specificky v PHP. Tento seznam lze jistě dále rozšířit, ale vybrané pachy považuji za nejdůležitější.

5 Návrh metodiky refaktoringu v PHP

Tato kapitola tvoří jádro práce. Popisuji zde navrhovanou metodiku. Kapitola je přehledně rozdělena do podkapitol pro snazší orientaci.

Abych mohl vůbec navrhnout metodiku, musím nejprve ujasnit požadavky na ni. Na jejich základě pak definuji tři základní principy, na kterých metodika stojí. Dále vyjmenovávám předpoklady nutné pro zavedení metodiky do praxe. Následuje soupis rolí.

V podkapitole *Postup při refaktoringu* uvádím algoritmus, podle kterého je vhodné postupovat. Každému z kroků algoritmu je věnována samostatná podkapitola, jelikož tyto kroky nejsou triviální.

5.1 Vymezení metodiky

Navrhovaná metodika se zabývá refaktoringem webových aplikací v PHP. Zaměřuje se na malé týmy či jednotlivce. Definuje postup od začátku do konce, jakýsi návod, přičemž ale nechává dostatek volnosti vývojářům pro rozhodování. Výhodou je, že takový refaktoring bude lépe uchopitelný jak pro méně zkušené v této oblasti, tak pro týmy jako celky, kterým tato metodika dodá určitou standardizaci (co se týče postupu, slovníku i praktik), aby byli všichni „na stejné vlně“.

Tato metodika se nezabývá refaktoringem databází. Tomu se věnují jiné práce (dvě z nich zmiňuji v kapitole *Rešerše literatury*). V praxi dojde při refaktoringu PHP často i k refaktoringu databáze, ovšem z hlediska návrhu této metodiky to není příliš podstatné (jak vyplyne ze zbytku této kapitoly). By bylo možné věnovat se refaktoringu databází více i bez návaznosti na PHP kód aplikace, motivací takového refaktoringu by bylo např. odstraňování redundancí či dokonce převod databáze do některé z definovaných normálních forem. To ale překračuje rámec této práce.

Dále také bylo zmíněno, že se metodika nezabývá refaktoringem front-endu[9] webové aplikace, který většinou používá jazyky HTML, CSS, JavaScript a další. Front-end je totiž, na rozdíl od databáze, na refaktoringu PHP kódu většinou nezávislý (při refaktoringu se nemění vnější chování). V praxi sice může k určitým změnám dojít v důsledku např. změny struktury PHP tříd a metod, ale ty budou mít za příčinu pouze malé změny ve front-endu. Téma refaktoringu front-endu by bylo zajímavým námětem na jinou akademickou práci.

Předkládaná metodika by mohla být s určitými změnami aplikována na vývoj webových aplikací v jiných jazycích či na refaktoring softwaru obecně (s určitými omezeními). Ovšem tato práce se cíleně omezuje pouze na refaktoring PHP kódu webových aplikací.

5.2 Požadavky na metodiku

Na základě literatury o refaktoringu, zkušeností z vlastní praxe i ze spolupráce s jinými vývojáři jsem stanovil tyto požadavky, které by měla navrhovaná metodika splňovat.

Definovat postupy (praktiky) pro určení, kdy a co refaktorovat

Má-li metodika fungovat jako návod, musí definovat postupy, které je možno následovat krok za krokem.

Minimalizovat riziko zanesení chyb

Refaktoring by samozřejmě neměl zanést do aplikace nové chyby. Navrhovaná metodika to zohledňuje, byť nevyžaduje tak striktně automatizovanou sadu testů pro všechny části kódu (viz dále). Patří sem i požadavek na používání správy verzí (což by měla být v projektech tak jako tak běžná praktika) – toto zajistí možnost návratu do posledního funkčního stavu v případě, že s refaktoringem přibudou nové chyby, které se obtížně hledají.

Maximalizovat efektivitu refaktoringu

Refaktoring je sice důležitá činnost, ale často zůstává nepochopen ze strany managementu. Proto je velmi vhodné refaktorovat efektivně jak z hlediska času, tak z hlediska subjektivně vynaloženého úsilí programátorů. Navíc když vývoj „jde od ruky“, podporuje to motivaci a disciplínu programátorů (ve srovnání se stavem, kdy se vývojář dlouho „zasekne“ na nějakém problému či dokonce dlouze hledá chyby).

[9] Front-end – prezentační vrstva, viditelná uživatelům [autor]

Tento požadavek souvisí s předchozím, neboť čím více zanesených chyb, tím nižší je efektivita.

Odstranit problémy v kódu do té míry, kdy to vývojáři/týmu „stačí“

Refaktorovat by se mělo vždy s konkrétním cílem, ne jen tak pro samoúčelné zkrášlení kódu [Beck ve Flowlerovi, 1999]. Cílem vývoje je aplikace, která má funkcionalitu požadovanou zadavatelem. Refaktoring má usnadnit (ideálně i zrychlit) dosažení tohoto cíle. Samoúčelný refaktoring je do značné míry ztrátou času z hlediska toho, že by práce mohla být odvedena rychleji bez něj.

Standardizovat postup refaktoringu

Navrhovaná metodika by měla pomoci se zavedením určitého systému refaktoringu do týmů vývojářů, kde byl dosud refaktoring záležitostí spíše zkušeností a dovedností jedince než sdílenou praktikou. Výsledkem bude mimo jiné snazší následování praktik a principů doporučovaných v literatuře, opora pro zaškolení méně zkušených vývojářů a efektivnější komunikace v týmu ohledně refaktoringu.

Přizpůsobitelnost konkrétnímu projektu a situaci

Mezi různými projekty a vývojovými týmy jsou nemalé rozdíly. Navrhovaná metodika nebude beze zbytku vyhovovat všem. Bude ale možné přizpůsobit si metodiku pro konkrétní projekt, a to zejména díky velké míře nezávislosti jednotlivých praktik a možnosti úpravy životního cyklu refaktoringu (např. důvodů pro jeho provedení).

Konkrétnější tipy pro přizpůsobení metodiky budou zmíněny přímo v jejím popisu dále v této práci.

5.3 Obecné vlastnosti navrhované metodiky

Z požadavků na metodiku a základních principů vyplývají některé vlastnosti navrhované metodiky, které bych zde rád explicitně zmínil.

Metodika je agilní. S drobnými odchylkami splňuje požadavky agilního manifestu [Beck, 2001]. Tyto odchylky jsou dány tím, že se nejedná o metodiku vývoje jako takového, ale jeho části, refaktoringu. Např. komunikace se zákazníkem není nutná, ale může být výhodná do té

míry, že tým pak může předvídat příští požadavky na funkcionalitu a přizpůsobit tomu některé větší refaktoringy.

Navrhovaná metodika se snaží být jednoduchá. Zaprvé je vymezená oblast poměrně úzká (ve srovnání s celým průběhem projektu vývoje softwaru), takže v tomto může sloužit jako vhodný doplněk některé z existujících metodik. Zadruhé má být tato metodika praktická, což mimo jiné znamená, že její zavedení by nemělo být příliš náročné. Je použitelná jak pro vývojáře-jednotlivce, tak i pro vývojové týmy. Více podrobností o jejím nasazení je uvedeno dále v práci.

Hlavním zdrojem informací pro tuto metodiku je literatura o refaktoringu, moje vlastní praxe a zkušenosti jiných vývojářů. Z hlediska metodik je pak volnou inspirací metodika OpenUP [OpenUP, 2012].

5.3.1 Principy metodiky

Na základě předchozího textu ještě explicitně definuji 3 základní principy navrhované metodiky. Nepotřebují, myslím, další komentář.

minimalizovat riziko chyb

maximalizovat efektivitu refaktoringu i vývoje celkově

maximalizovat praktickou využitelnost metodiky

5.4 Předpoklady pro použití metodiky

Vývoj webové aplikace v PHP

Jak již bylo zmíněno, navrhovaná metodika se zaměřuje na vývoj webových aplikací v PHP, ovšem s určitými úpravami by bylo možné ji použít i jinde (což přesahuje rámec této práce).

Existující kód

Je samozřejmým předpokladem, že kód aplikace již existuje, jelikož se jedná o změny ve stávajícím kódu za účelem jeho zkvalitnění. To ale neznamená, že aplikace musí být hotová. Právě naopak, refaktoring by měl probíhat průběžně při vývoji, tedy často na kódu, který je starý jen pár hodin nebo dní. Více k tomuto viz podkapitola *Praktiky*.

Použití objektově orientovaného programování

PHP dává vývojářům na výběr, zda budou či nebudou programovat objektově. Kód v PHP může být objektově orientovaný či procedurální. Tato metodika pro jednoduchost předpokládá použití objektově orientovaného přístupu, ovšem s drobnými obměnami by bylo možné ji využít i při procedurálním přístupu.

Malý tým a dobrá komunikace

Dalším předpokladem je malý tým. Konkrétní velikost záleží mimo jiné na jejich zkušenosti a úrovni komunikace. Např. pokud pracují všichni v jedné místnosti, bude komunikace mnohem lepší než na dálku, a tak je možné použít metodiku pro větší tým. Nemá smysl zde definovat nějakou pevně danou maximální velikost týmu, ale pro představu uvedu, že by to nemělo být více než zhruba 8 vývojářů.

Vhodnost metodiky v konkrétní situaci

Nezbytná je rovněž vhodnost metodiky pro konkrétní projekt, vývojový tým a firmu. Tato metodika nemusí vyhovovat každému a je proto na individuálním posouzení každého týmu, zda a do jaké míry se bude této metodiky držet. Inspiraci při tomto rozhodování lze najít např. v [Buchalcevová, 2009], kde autorka zmiňuje faktory výběru metodiky pro vývoj.

Podpora metodiky celým týmem

Vývojář-jednotlivec může rozhodnout sám za sebe, zda využije tuto metodiku. Ovšem u týmů nastává možná komplikace v tom, že metodika musí být přijata ve stejném rozsahu všemi členy týmu. Pokud se někdo rozhodne postupovat jinak než ostatní (např. nerespektovat některé praktiky či postupy, zatímco ostatní na ně spoléhají), může dojít k nedorozuměním a snížení efektivity práce.

Znalosti a zkušenosti vývojářů

Při nasazení metodiky se předpokládá, že vývojáři mají dostatečnou úroveň znalostí a zkušeností, aby zvládli psát čistý a přehledný kód, který je snadno udržovatelný a rozšiřitelný. Pro dodatečné získání znalostí v tomto ohledu lze doporučit práce zmíněné v části Rešerše literatury: knihy [Martin, 2009] a [McConnell, 2004] a případně stručnější webovou prezentaci [Doporučené postupy v programování, 2009].

Další velmi doporučenou oblastí k prostudování jsou návrhové vzory. Vhodné knihy pro získání odpovídajících znalostí jsou např. [Pecinovský, 2007], [Freeman, 2004] a [Gamma et.al., 1994] (jsou také zmíněny v části Rešerše literatury).

Ovšem teoretické znalosti nejsou samy o sobě dostatečné, vývojář by měl mít tyto znalosti zažité a vyzkoušené v praxi. Pokud jsou v týmu méně zkušení jedinci, dá se také investovat do jejich vzdělání tím, že úzce spolupracují se zkušenějším vývojářem, např. formou pravidelných revizí kódu.

PHP je pro mnoho vývojářů jejich prvním programovacím jazykem, a proto jejich kvalita často není příliš dobrá (jelikož teprve nedávno začali s programováním, což je mnohem větší nevýhoda než přechod na nový programovací jazyk).

Je důležité umožnit méně zkušeným vývojářům, pokud už v týmu jsou, spolupracovat se zkušenějšími kolegy, tím se totiž zvyšuje celková kvalita týmu. I když by se mohlo zdát, že se to časově (finančně) nevyplatí, opak bývá často pravdou. Zkušenější vývojář bývá o mnoho efektivnější. V kontextu refaktoringu to může znamenat například přehlednější kód, a tudíž méně chyb, snazší testování a ladění nebo lepší orientace kolegů v jeho kódu (např. při přerozdělení práce).

Pro markantní zlepšení může přitom stačit půlhodinové školení nebo několika stránkový materiál o základech psaní kvalitního kódu. Samozřejmě, není možné se za tak krátkou dobu naučit psát kvalitní kód a kvalitně navrhovat design aplikace. Ale proto jsou právě vhodné pravidelné revize kódu se zkušenějším kolegou a občasné připomenutí a rozšíření naučených principů.

5.5 Role

Tato podkapitola vyjmenovává role vývoje webového projektu, které mohou přijít do styku s refaktoringem a rozhodováním o něm. U každé role je popsáno pouze to, jak ovlivňuje refaktoring, nikoli její další působení v projektu. Role jsou s výjimkou *Vedoucího týmu vývojářů* převzaty z metodiky OpenUP [OpenUP, 2012], kde se lze také dočíst více podrobností pro lepší pochopení každé z nich.

Jeden člověk může mít více rolí. V krajním případě může dokonce zastávat všechny role jeden člověk.

Iniciovat refaktoring může kdokoli, ovšem rozhodují o tom vývojáři, u velkých refaktoringů pak hlavní vývojář spolu s projektovým manažerem.

Vedoucí týmu vývojářů

Zastřešuje tým vývojářů, komunikuje za něj navenek, především s projektovým manažerem. Rozhoduje o (ne)provedení větších refaktoringů. Toto podle uvážení konzultuje s ostatními rolemi. Tato role nemusí být u velmi malých týmů (2-3 lidé) definována.

Podporuje a řídí také komunikaci mezi vývojáři, což je zásadní při refaktoringu, který zasahuje do kódu, jehož autorem je jiný z vývojářů.

Vývojář

Vývojář je role, která ve skutečnosti jako jediná provádí samotný refaktoring. Komunikuje primárně s *Vedoucím týmu vývojářů*, je ale podporována i komunikace s ostatními rolemi v případě řešení konkrétních problémů z jeho oblasti.

V týmu je vývojářů zpravidla více. O to důležitější je rozdělení práce (což má na starosti *Vedoucí týmu vývojářů*) a dostatečná komunikace. Ideální jsou malé týmy (ne více než 8 vývojářů), a to v jedné místnosti, pak je totiž nejsnazší osobní komunikace, která je také nejefektivnější [Buchalcevová, 2009].

Analytik

Komunikuje se zadavatelem a zpracovává jeho požadavky, má tedy největší přehled o tom, kam se projekt bude v nejbližší budoucnosti vyvíjet, což může být výhodné při rozhodování o větším refaktoringu.

Architekt

Architekt je zodpovědný za architekturu aplikace a zásadní technická rozhodnutí. Větší refaktoringy zasahující zásadním způsobem do struktury aplikace je tedy třeba konzultovat s ním.

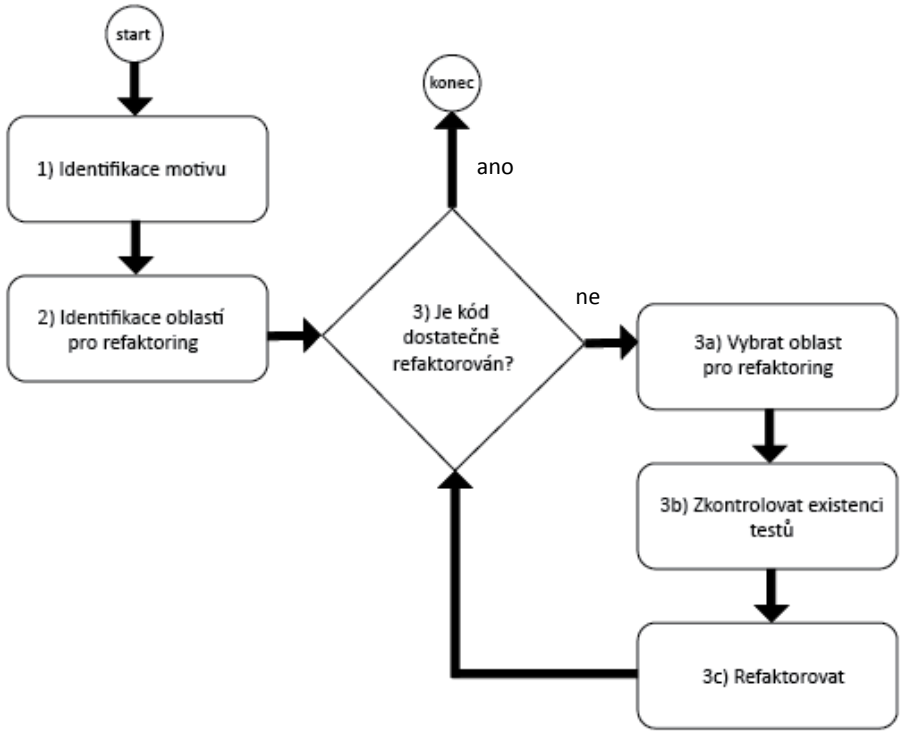
Komunikuje především s *Vedoucím týmu vývojářů* a s *Analytikem*.

Projektový manažer

Tuto roli zde uvádím proto, že v praxi jsou časté neshody vývojářů s managementem ohledně refaktoringu. To proto, že refaktoring nepřidává žádnou novou funkcionalitu a managementu se tak nezdá být efektivní.

Lze proto doporučit sjednocení představ projektového manažera a vývojářů o refaktoringu a jeho důležitosti v projektu. Je také možné stanovit včas pravidla a rozhodovací pravomoci v této věci. Konkrétní detaily jsou ale už velmi závislé na projektu, firmě, náhledu a zkušenostech konkrétních lidí, proto je v této metodice nebudu specifikovat.

5.6 Postup při refaktoringu



Obrázek 5-1: Postup při refaktoringu (zdroj: autor)

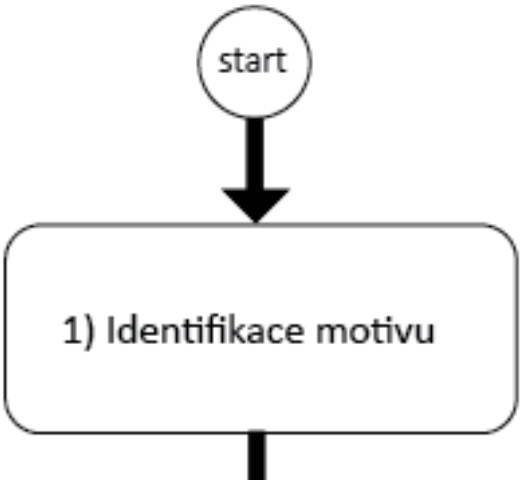
Jelikož tato metodika má sloužit také jako návod, uvádím na tomto místě algoritmus pro postup při refaktoringu v kontextu vývoje (viz také *Obrázek 5-1*):

1) **Identifikace motivu** (viz podkapitola *Motivy k refaktoringu*)
2) **Identifikace oblastí pro refaktoring** (viz podkapitola *Identifikace oblastí vhodných pro refaktoring*)

3) **Dokud není kód dostatečně refaktorován** (viz podkapitola *Cíle refaktoringu*)**, opakovat:**
    a) **Vybrat oblast pro refaktoring** (viz podkapitola *Výběr oblasti pro refaktoring*)
    b) **Zkontrolovat existenci testů** (více o testování viz praktika P3 *Testovat*)
    c) **Refaktorovat** (viz podkapitola *Praktiky* a literatura o refaktoringu)

5.7 Motivy k refaktoringu

Refaktoring by neměl být samoúčelný, tzn. vždy by měl sloužit nějakému účelu, vždy by pro něj měl být důvod. Jinak to je ztráta času, který by bylo možno vynaložit nějak jinak, efektivněji (Kent Beck ve [Fowler, 1999]). Např. pokud je třeba přidat funkcionalitu, pak je pro projekt jistě efektivnější pracovat na tomto úkolu, než provádět refaktoring, pro který není jiný důvod než „zkrášlit kód".



Obrázek 5-2: Identifikace motivu - krok 1 (zdroj: autor)

Naproti tomu např. [Shore, 2008] doporučuje provádět „velký refaktoring" pravidelně jednou týdně, i když dodává, že se to má týkat pouze kódu, na kterém vývojář aktuálně pracuje. Tato metodika se místo toho spoléhá na seznam motivů, které dle mého názoru ústí v to, že takto pravidelný refaktoring není potřeba. Zejména díky motivu *Splnění dílčího úkolu* a odpovídající praktice (viz dále).

Následuje výčet motivů, které mohou a které by měly iniciovat refaktoring (viz *Tabulka 5-1*). V této části vycházím většinou z [Fowler, 1999], která obsahuje mnoho užitečných rad, a vlastní praxe. Další zdroje jsou vyznačeny přímo v textu.

Každý motiv navíc charakterizuji jako *lokální* nebo *globální* podle toho, zda poměrně úzce vymezuje oblast kódu, ve které je refaktoring třeba. Např. první motiv, *Splnění dílčího úkolu*, je lokální, neboť po dokončení úkolu vývojář ví, na který kód se má z hlediska refaktoringu

zaměřit – je to ten, který právě napsal. A nehraje přitom roli, zda je všechen kód na jednom místě aplikace či nikoli. Naopak v případě motivu globálního typu je třeba hledat místa vhodná pro refaktoring v rámci celé aplikace nebo její velké části. Typickým příkladem globálního motivu je *Převzetí cizího kódu*.

Z předchozího textu by už měl být jasný účel tohoto dělení. V případě lokálního motivu není třeba hledat, kde vlastně refaktoring provést, je to poměrně přímočaré. Zajímavější je pak tato problematika u globálních motivů. Viz podkapitola *Identifikace oblastí vhodných pro refaktoring*.

Tabulka 5-1: Seznam motivů k refaktoringu (zdroj: autor)

Označení	**Název**	**Typ**
M1	Splnění dílčího úkolu	lokální
M2	Potřeba změn kódu při vývoji	lokální
M3	Přidání komentáře	lokální
M4	Přidání nové funkcionality	lokální
M5	Hledání chyby	lokální (globální)
M6	Revize kódu	lokální
M7	Potřeba optimalizace kódu	globální
M8	Klesající čitelnost a rozšiřitelnost kódu	globální (lokální)
M9	Převzetí cizího kódu	globální
M10	Příchod nového vývojáře do týmu	globální

M1. Splnění dílčího úkolu

Typ: lokální

Tento postup doporučuje [Cohn, 2010]. Vychází z toho, že refaktorovat kód velmi brzy (dá se říci ihned) po jeho napsání je nejjednodušší, jelikož vývojář jej má ještě v čerstvé paměti a přesně ví, proč co udělal.

Cílem je vylepšit kvalitu kódu pro jeho další používání v dalších dnech, za měsíc či za rok. Je v pořádku splnit dílčí úkol (přidat část funkcionality) bez ohledu na čitelnost a udržovatelnost kódu, ale není v pořádku jej pak takto nechat.

Tento motiv je nejblíže k onomu důvodu „zkrášlit kód“, který kritizuji v úvodu k této podkapitole. Ovšem zde to má smysl, neboť se přidává důvod popsatelný jako „zanechat po sobě kvalitní kód“.

Tento motiv je jeden z nejčastějších, jelikož se opakuje po každém dílčím úkolu. Vzhledem k jeho častému opomíjení jej přidávám také jako samostatnou praktiku, viz podkapitola *Praktiky*.

M2. Potřeba změn kódu při vývoji

Typ: lokální

Tento motiv nastává průběžně při vývoji. Příkladem může být extrakce kódu do metody nebo přejmenování proměnné. Vývojář si uvědomí, že mu kód na určitém místě nevyhovuje a rozhodne se pro refaktoring. Ten typicky netrvá příliš dlouho a postup k němu bude možné nalézt ve [Fowler, 1999].

V citované knize je také zmíněno *Pravidlo tří (The rule of Three)* od Dona Robertse. To říká, že pokud vývojář potřebuje použít (napsat) stejnou věc podruhé, napíše to znovu navzdory vznikajícímu duplicitnímu kódu, ale potřetí už by měl refaktorovat.

Pro rozhodování o tom, co by mělo být předmětem refaktoringu, jsou důležitým předpokladem znalosti a zkušenosti vývojáře. Mezi zásadní znalosti patří typy *pachů kódu (code smells)*. Jakmile vývojář narazí na nějaký *pach*, měl by refaktorovat, pokud to není z nějakého důvodu nevýhodné (jedním z takových důvodů může být příliš malá důležitost takového *pachu* v daném kontextu, na to se právě odkazuje výše zmíněné *Pravidlo tří*). Primární totiž není snaha o dokonalý kód, nýbrž snaha o maximalizaci efektivity vývoje.

M3. Přidání komentáře

Typ: lokální

Přidání komentáře může signalizovat příležitost pro refaktoring, jelikož kód není dostatečně čitelný a samovysvětlující. Může pomoci např. extrahovat metodu, a to i z jednoho řádku kódu, pokud by jinak potřeboval vysvětlující komentář.

Často se stává, že komentáře jsou v kódu proto, že je tento kód špatný. Zakrývají vlastně *pachy*. Vždy je lépe pokusit se nejprve kód refaktorovat. Komentáře ovšem mají své místo. Neměly by vysvětlovat, co se děje, ale jak je to uděláno a proč právě takto (je-li to třeba).

M4. Přidání nové funkcionality

Typ: lokální

Pokud je relativně obtížné přidat novou funkcionalitu a při lepší (kvalitnější) struktuře kódu by to bylo snazší, je vhodné refaktorovat a dosáhnout tak této lepší struktury, byť by to mělo znamenat pouze „úklid“ dané oblasti kódu. Úsilí vložené do refaktoringu se vyplatí také v tom, že vývojář lépe pochopí (připomene si), co se v dané oblasti děje, a o to rychlejší pak bude samotné přidání funkcionality.

M5. Hledání chyby

Typ: lokální (může být i globální)

Refaktoring může být variantou ladění kódu neboli hledání chyby. V některých situacích to může být efektivnější, obzvláště je-li oblast kódu, ve které se chyba pravděpodobně nachází, značně nepřehledná a kód nekvalitní.

M6. Revize kódu

Typ: lokální

Revize kódu (*code* review nebo *peer review*) je aktivita, při které napsaný kód prochází alespoň jeden další vývojář. Pohled jiného člověka může odhalit různé chyby, kterých si autor nevšiml. Tento nový pohled zároveň bývá motivem pro refaktoring, kdy se zjistí, že daný kód má jisté nedostatky a je možné jej napsat lépe.

[Fowler, 1999] doporučuje refaktorovat přímo při revizi, pokud je to proveditelné, a implementovat tak rovnou své připomínky a návrhy. Navíc to, jak uvádí, výrazně zlepšuje porozumění revidovanému kódu.

M7. Potřeba optimalizace kódu

Typ: globální

Při optimalizaci je třeba nejprve změřit výkon aplikace, aby bylo vidět, kde je problém a kterou oblast kódu je třeba optimalizovat. Pokud tamní kód není refaktorován, může být snazší jej nejprve refaktorovat a pročistit. Jednak se už tímto může kód zrychlit (např. vlivem pročištění závislostí) a jednak bude pak snazší provést samotnou optimalizaci. Ta už pak může být na úkor kvality kódu, nebude-li zbytí.

Jelikož mají někteří vývojáři tendenci psát „co nejrychlejší“ kód na úkor jeho kvality, zařadil jsem do této metodiky i praktiku *Optimalizovat až refaktorovaný kód* (viz podkapitola Praktiky).

M8. Klesající čitelnost a rozšiřitelnost kódu

Typ: globální (může být i lokální)

Tento motiv k refaktoringu nenastane z ničeho nic, ale zpravidla již delší dobu pozorují vývojáři klesající efektivitu vývoje, která v určitém okamžiku překročí pomyslnou hranici trpělivosti, která je samozřejmě velmi subjektivní, a některý z vývojářů iniciuje refaktoring. Na rozdíl od ostatních zde zmíněných motivů nelze tento okamžik příliš předvídat a tento je do velké míry závislý na zkušenostech vývojářů, projektových standardech apod.

M9. Převzetí cizího kódu

Typ: globální

Tímto tématem se obšírně zabývá [Feathers, 2005]. Pokud je třeba pochopit kód, se kterým ale vývojář dále pracovat nebude, pak přichází v úvahu tzv. *rychlý refaktoring* (viz příslušná praktika v podkapitole *Praktiky*). V opačném případě je vhodný standardní refaktoring jak za účelem pochopení (není-li kód kvalitní a dobře čitelný), tak za účelem dodržování projektových či firemních standardů.

M10. Příchod nového vývojáře do týmu

Typ: globální

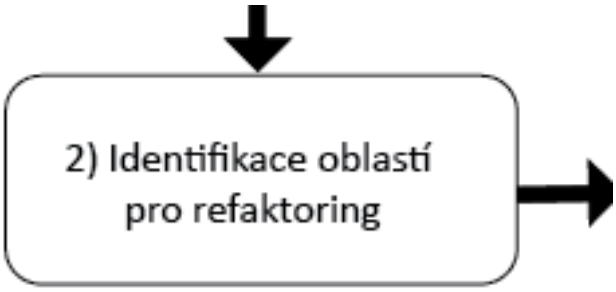
Pokud je projekt relativně malý a vývojový tým si udržuje přehled o kódu i přesto, že kód není příliš kvalitní a čitelný, pak nastává problém, když do týmu přijde nový vývojář. V tomto okamžiku (či ještě s určitým předstihem) je vhodná příležitost zrevidovat stávající kód a zpřehlednit jej.

Je jasné, že to nemusí být vždy proveditelné, zvláště pokud se dá očekávat, že by refaktoring zabral více času, než je únosné např. vzhledem k termínu odevzdání. Ovšem vždy by měla taková situace vést k důkladnému zamyšlení, zda není vhodná chvíle vylepšit kvalitu kódu.

5.8 Identifikace oblastí vhodných pro refaktoring

Pokud jde o refaktoring na základě jednoho z lokálních motivů, jsou oblast nebo oblasti kódu pro refaktoring většinou dostatečně jasně určené. Vývojář typicky narazí na některý

z pachů při vývoji. Často je pach tak silný (kód tak nekvalitní, nečitelný či nevhodně strukturovaný), že efektivita práce významně klesá. V takové situaci je důležité, aby to vývojář zaznamenal, zastavil vývoj, nasadil si příslovečný „refaktoringový“ klobouk (viz praktika *Dva klobouky*) a vylepšil kvalitu kódu.



Obrázek 5-3: Identifikace oblastí pro refaktoring - krok 2 (zdroj: autor)

V případě motivů globálních je o něco obtížnější najít v aplikaci místa největšího pachu. Je třeba prozkoumat kód aplikace jako celku. K tomu mohou pomoci metody uvedené v *Tabulce 5-2* a popisované dále v této podkapitole.

Je třeba ještě připomenout, že v tomto kroku nám nejde o odstranění pachů, ale pouze o jejich identifikaci. Vhodnou praktikou podle [Cohn, 2010] je poznamenat si nalezené pachy, v případě týmu pracujícího na jednom místě nejlépe na papír, jinak v elektronické podobě. Tento seznam pak po splnění roztrhat, příp. smazat. Podle citované knihy není dobré používat robustnější řešení, neboť takového seznamu je pak těžké se zbavit, zůstává nastálo, jelikož se plní dalšími a dalšími položkami.

Můj názor je, že není třeba odstranit všechny pachy (a zpracovat tak celý seznam), neboť u některých se to již nemusí vyplatit. Cílem je totiž maximalizace efektivity vývoje (je to jeden z principů této metodiky), nikoli co nejkvalitnější kód. To, zda se odstranění určitého pachu vyplatí či nikoli, už je na posouzení vývojáře či vedoucího týmu či celého týmu.

Důležité ale je pracovat nejprve na odstranění pachů, které nejvíce snižují efektivitu vývoje. O tomto pojednává podkapitola *Výběr oblasti pro refaktoring* (viz dále).

Tabulka 5-2: Metody k identifikaci oblastí vhodných pro refaktoring (zdroj: autor)

Označení	Název
I1	Sepsání známých problémů a nedostatků
I2	Nástroje pro statickou analýzu kódu
I3	Revize návrhu aplikace
I4	Manuální procházení kódu
I5	Profilování
I6	Kontrolní seznamy

I1. Sepsání známých problémů a nedostatků

Tato metoda je nasnadě. Pokud provádí refaktoring vývojář, který aplikaci již dříve, alespoň částečně, vyvíjel, je pravděpodobné, že si ihned poté, co se rozhodne pro refaktoring, vybaví oblasti aplikace, které je třeba refaktorovat. Ideálně by měl takový refaktoring probíhat průběžně (viz *Motivy k refaktoringu*), ovšem v praxi vidím, že tomu tak příliš často nebývá, zejména kvůli podceňování přínosů, což pak způsobí, že na refaktoring „nezbývá čas“.

I2. Nástroje pro statickou analýzu kódu

Existují nástroje, které provedou analýzu kódu bez toho, aby bylo třeba jej spustit (provést). Pro analýzu PHP kódu lze využít např. PHP Depend, PHP Mess Detector, PHP Code Sniffer nebo PHPLOC. Nebudu je zde více popisovat, neboť jejich orientační popis lze nalézt např. v [Hujer, 2012].

Z této analýzy lze následně vyčíst, které oblasti kódu jsou příliš komplikované, kde jsou porušovány standardy pro psaní kódu (možno i definovat vlastní) atd. Bližší informace o výstupech těchto analýz lze nalézt ve výše citované práci nebo na webových stránkách jednotlivých nástrojů.

Nelze striktně definovat parametry, které by určovaly pachy kódu, je třeba výstupy analýz projít manuálně, byť i s pomocí automatizovaných nástrojů, a posoudit jednotlivá místa individuálně. Může to být dost práce, ovšem není třeba odhalit všechny problémy v kódu, jen ty nejzávažnější.

I3. Revize návrhu aplikace

Zde se jedná především o kontrolu, zda je udržován vhodný návrh aplikace i s postupujícím vývojem. Typickým příkladem je hierarchie tříd, která s přibývajícími třídami a novými požadavky na funkcionalitu zpravidla přestává vyhovovat. Refaktoring je zde

nutností, ovšem snadno se může stát, že vývojář tuto potřebu včas nevycítí a dále rozšiřuje nevyhovující hierarchii namísto její změny.

Tyto problémy lze identifikovat při manuálním procházení kódu. Další pomůckou je zaměřit se na odpovědnosti a stávající interakce tříd [Feathers, 2005]. Také zde přicházejí na pomoc zásady objektově orientovaného programování, která popisuje např. [Freeman, 2004]. Příkladem za všechny může být to, že každá třída má mít jen jednu funkci (myšleno úkol či odpovědnost, nikoli samozřejmě *metodu*) – pokud to některá ze tříd v aplikaci porušuje, pak je jistě vhodné prozkoumat, zda by nebylo možné návrh vylepšit pomocí refaktoringu.

I4. Manuální procházení kódu

Tato metoda může vhodně doplňovat metody předchozí. Je ale nutné vědět, co hledat, na co se zaměřit. V první řadě jsou to samozřejmě pachy kódu. K tomu je potřeba je dobře znát a rozumět jim. Pachy v kódu vznikají s postupujícím vývojem vždy (*technický dluh*), ovšem je třeba je odstraňovat refaktoringem, jinak se další vývoj stane příliš neefektivním a nevýhodným [Shore, 2008].

Dále jsou to také prohřešky proti standardům a konvencím projektu. To nastává například, pokud do určitého okamžiku žádné takové standardy a konvence nebyly a každý vývojář měl mírně odlišný způsob zápisu a strukturace kódu. Nehledě na způsob vzniku může nevyhovující kód začít komplikovat další vývoj, tedy pach začne sílit. Nejpozději v tomto okamžiku je refaktoring zcela na místě.

Toto procházení kódu lze také uskutečňovat ve spolupráci s dalšími vývojáři jako formu revize kódu.

Je ale třeba zmínit, že pokud se jedná o refaktoring většího rozsahu, typicky s jedním z globálních motivů, pak nemusí být tato metoda výhodná, jelikož kódu může být mnoho a zabralo by se tak příliš mnoho času. Efektivnější jsou pak ostatní, zde uváděné metody.

I5. Profilování

Díky měření výkonu aplikace lze odhalit oblasti kódu, která je třeba optimalizovat. Tyto oblasti je ovšem vhodné před optimalizací refaktorovat, pokud kód ještě není dostatečně kvalitní. Zaprvé bude následná optimalizace jednodušší a zadruhé, a proto uvádím profilování v této podkapitole, je možné, že se refaktoringem problém odstraní.

Může se jednat například o zbytečně mnoho volání nějakých metod či zbytečné vytváření mnoha objektů namísto opakovaného používání jednoho, už vytvořeného.

I6. Kontrolní seznamy

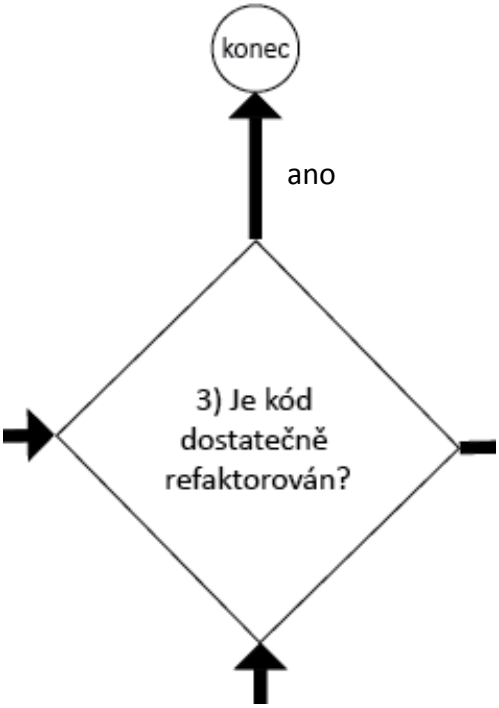
Kontrolní seznamy (anglicky *checklists*) pro návrh a implementaci aplikace mohou také poukázat na nedostatky dosavadního vývoje. Kontrolní seznamy si může vývojový tým vytvořit sám, nebo je možné použít již hotové (a případně je upravit).

Příkladem mohou být kontrolní seznamy metodiky OpenUP. [10][11]

5.9 Cíle refaktoringu

V případě lokálního motivu řeší vývojář zpravidla jeden či několik málo pachů. Cílem pak je tyto pachy zcela eliminovat či značně redukovat do té míry, kdy se již nevyplatí se jimi zaobírat (z hlediska celkové efektivity vývoje).

V případě globálního motivu je pachů zpravidla více. Snaha o dokonalý kód (eliminovat veškeré identifikované pachy) není na místě, opět kvůli maximalizaci efektivity vývoje. Ovšem stanovení hranice, kdy se refaktoring ještě vyplatí, je bohužel značně neuchopitelné. Zůstane proto opět na individuálním posouzení vývojáře (s případnou konzultací s vedoucím týmu či s ostatními vývojáři).



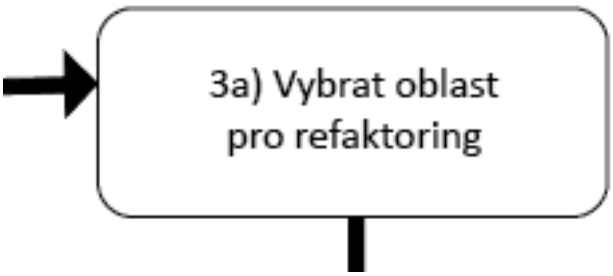
Obrázek 5-4: Rozhodnutí, zda bylo dosaženo cíle refaktoringu - krok 3 (zdroj: autor)

10 http://epf.eclipse.org/wikis/openup/practice.tech.evolutionary_design.base/guidances/checklists/design_68980812.html

11 http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/checklists/implementation_A5578577.html

5.10 Výběr oblasti pro refaktoring

Poté, co vývojář identifikuje pachy v kódu aplikace podle podkapitoly *Identifikace oblastí vhodných pro refaktoring*, je třeba vybrat jeden a na ten aplikovat refaktoring s cílem jej eliminovat či alespoň co nejvíce redukovat. S ohledem na maximální celkovou efektivitu vývoje nemusí být nejlepším řešením pach zcela odstranit, jelikož potřebné úsilí se někdy již nemusí vyplatit. Více viz předchozí podkapitola *Cíle refaktoringu*. Pro jednoduchost budu ale v dalším textu psát jen o „odstranění pachu“.



Obrázek 5-5: Výběr oblasti pro refaktoring - krok 3a (zdroj: autor)

Dle mého názoru zde nelze stanovit exaktní kritéria. Dostupná literatura také žádné postupy neobsahuje. Proto poskytnu alespoň několik tipů, na co se zaměřit, na základě vlastních zkušeností. Finální výběr pak zůstává na vývojáři, takže jsou zde opět důležité jeho znalosti, zkušenosti a také přehled o aktuální situaci ve vývoji.

Sledovat poměr „cena/výkon“

Cenou se zde rozumí především úsilí a čas, které je třeba vložit do provedení refaktoringu (odstranění pachu). Do úsilí je dále vhodné zahrnout rizika s refaktoringem spojená (např. nejsou-li k dispozici testy, riziko může být značné). Problémem je, že cenu refaktoringu může být předem těžké odhadnout.

Výkonem se rozumí přínos, který bude refaktoring a odstranění pachu mít. Bude to zpravidla lepší čitelnost a rozšiřitelnost kódu. Bohužel nelze takový přínos exaktně měřit, o to větší důraz je tedy kladen na úsudek vývojáře.

Zohlednit závislosti jednotlivých pachů

Může se stát, že jednotlivé pachy se do určité míry překrývají. Pak je výhodné zvolit takové pořadí refaktoringů, které vyžaduje menší celkové úsilí (eliminace jednoho pachu může redukovat či dokonce eliminovat další).

Dát vyšší prioritu větším (náročnějším) refaktoringům

Oblast výrazně nižší kvality kódu bude pravděpodobně vývojáře do jisté míry odpuzovat od refaktoringu. Ovšem zpravidla je výhodné začít důležitějšími refaktoringy, než se zaobírat těmi menšími, méně podstatnými. Důvodů je několik. Zaprvé je s tímto přístupem jistota, že se i na takovou oblast kódu dostane (při blížících se termínech by se jinak mohlo stát, že by na tuto oblast nezbyl čas). S tím souvisí i lepší odhad časové náročnosti refaktoringu, jelikož vývojář má „to nejhorší za sebou“. Není také výjimkou, že takový kód má přesah do dalších částí aplikace a úpravy, které je v něm třeba provést, budou přínosné i pro jiné oblasti aplikace; příkladem může být vytvoření nových tříd zapouzdřujících a opakovaně zpřístupňujících určitou funkcionalitu.

Ne vždy je ale tento přístup výhodný. Pokud je takový refaktoring závislý na refaktoringu jiných částí (viz předchozí tip o závislosti jednotlivých pachů), je třeba věnovat se nejprve jim. Také je nutno zvážit, zda takto náročný refaktoring má dobrý poměr „cena/výkon“, viz výše.

Udržovat subjektivní pocit pokroku

[Fowler, 1999] velmi stručně zmiňuje, že jedna z důležitých věcí při refaktoringu je viditelný postup a z toho pramenící pocit uspokojení. Zní to možná zvláštně, ale vývojáři jsou také lidé a pocit postupu je důležitý pro jejich vnitřní motivaci. A ta je zase důležitá pro efektivnější práci.

U malých refaktoringů, které netrvají déle než desítky minut, je přísun tohoto uspokojení většinou zajištěn (pokud ho nepřekoná pocit frustrace způsobený zanesením chyb apod., proto je důležité následovat níže popsané praktiky). U větších refaktoringů je velmi žádoucí rozložit je na menší kroky.

Pointou tohoto tipu je hlídat si svůj subjektivní pocit pokroku refaktoringu a pokud je příliš nízký a motivace klesá, může být vhodné zařadit refaktoring se znatelnějším pokrokem, byť méně účelný.

Opravovat prohřešky proti konvencím projektu

Pokud vývojář nalezne kód odporující konvencím projektu a tento pach není závislý na jiném (viz výše), může být vhodné a poměrně snadné dát odstranění tohoto pachu vysokou prioritu. Koneckonců, jednou by tyto změny musely být tak jako tak provedeny (záleží tedy samozřejmě na striktnosti těchto konvencí a vedoucího vývojářů).

Mám zde na mysli především jmenné konvence a formátování kódu. Pokud konvencím nevyhovuje např. délka metod, bude takový zápach na odstranění náročnější.

5.11 Praktiky

V této části vycházím opět většinou z [Fowler, 1999] a vlastní praxe. Další zdroje jsou vyznačeny přímo v textu. V *Tabulce 5-3* opět nejprve uvádím jejich seznam.

Tabulka 5-3: Seznam praktik (zdroj: autor)

Označení	Název
P1	Dva klobouky
P2	Dělat jen jednu věc
P3	Testovat
P4	Refaktorovat s ohledem na testovatelnost kódu
P5	Refaktorovat podle doporučeného postupu
P6	Minimalizovat velikost kroků
P7	Testovat po každém kroku
P8	Neladit chyby vzniklé refaktoringem, ale zmenšit velikost kroků a provést je postupně
P9	Refaktorovat ihned po dokončení menšího úkolu
P10	Rozhodovat o (ne)provedení refaktoringu podle jeho předpokládaného rozsahu
P11	Dát refaktoringu vysokou prioritu
P12	Používat systém na správu verzí
P13	Optimalizovat až refaktorovaný kód
P14	Refaktorovat i cizí kód, je-li to třeba
P15	Porozumět cizímu kódu rychlým refaktoringem
P16	Refaktorovat v páru

P1. Dva klobouky

Přidávání nové funkcionality a refaktoring jsou dvě různé činnosti a neměly by se nikdy míchat. Fowler k popisu této zásady využívá metaforu Kenta Becka o dvou kloboucích – vývojář vždy musí vědět, který klobouk má na hlavě. A samozřejmě nemůže mít na hlavě oba najednou. Na druhou stranu, lze je střídat i každých pár minut, je-li to vhodné.

S tím souvisí praktika *Dělat jen jednu věc*.

P2. Dělat jen jednu věc

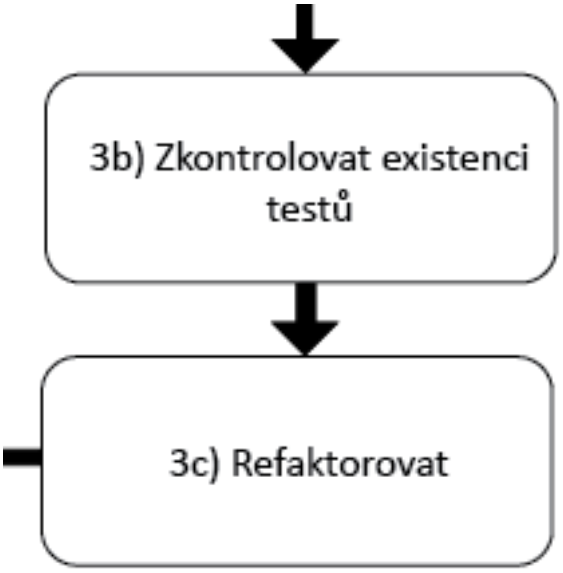
Pro efektivitu práce je zásadní dělat v daném okamžiku jen jednu věc [Feathers, 2005]. Pokud např. při refaktoringu vývojář přichází na další věci, které by bylo vhodné

refaktorovat, nemá se do nich pouštět rovnou. Správným postupem je poznamenat si je a dokončit aktuální refaktoring.

Dovést aplikaci do stavu, kdy opět běží bez chyb (všechny testy doběhnou úspěšně), přitom nemusí znamenat dokončení, jelikož postup většiny refaktoringů se skládá z více menších kroků, mezi kterými je třeba testy spouštět (a tedy se předpokládá, že uspějí). Dokončením se tedy myslí ukončení aktuálního refaktoringu provedením všech kroků dle katalogu.

P3. Testovat

Veškerá literatura o refaktoringu, se kterou jsem se doposud seznámil, důrazně radí mít psát automatizované testy. Testy proto, aby se minimalizovalo riziko zanesení chyby během refaktoringu. Automatizované proto, aby bylo co nejjednodušší a nejpohodlnější je spouštět, protože jen tak je šance, že je bude vývojář spouštět během refaktoringu (i vývoje) dostatečně často.



Obrázek 5-6: Kontrola existence testů a refaktoring - kroky 3b a 3c (zdroj: autor)

V praxi vývoje webových aplikací se ovšem často setkávám s tím, že vývojáři testy nepíší. Zdá se jim to jako neefektivní způsob vývoje, přestože příručky o vývoji řízeném testy tento postup doporučují a prosazují. Problém je v tom, že tento způsob se vývojáři zdá zpravidla nejprve pomalý a musí si na něj zvyknout. Ovšem v dlouhodobém měřítku se projeví zvýšení produktivity způsobené tím, že chyby jsou odhaleny v průměru mnohem dříve [Beck, 2005].

Vzhledem k tomu, že tato metodika klade důraz na použití v praxi, nespokojím se jen s konstatováním, že je třeba psát automatizované testy. Implementace takového přístupu vyžaduje jistou dávku učení a disciplíny a ne každý vývojář je ochoten se takovéto nové věci naučit a přijmout ji za svou. Proto nebudu s tímto stavem bojovat nebo jej přehlížet, ale přizpůsobím mu tuto metodiku. Z hlediska principů, na kterých tato metodika stojí, jde o kompromis mezi principem minimalizace chyb a praktické využitelnosti metodiky.

V zájmu praktické využitelnosti slevím z požadavku na automatizaci testů a budu považovat také manuální testování za vhodný způsob, jak testovat. Manuálním testováním v kontextu vývoje webových aplikací myslím znovunačtení stránky v prohlížeči a vizuální kontrolu výsledků. Velmi často tento přístup stačí a nemusí být ani pomalejší než automatizované testování. Jeho nevýhodou však je, že testuje aktuální část aplikace (načítaná stránka) pouze v přítomném okamžiku a ne už v budoucnu. Tedy pokud zanese vývojář někam chybu později a tato chyba se projeví na této stránce, bude obtížné to při vývoji odhalit, jelikož tato stránka už později nebude (manuálně) testována. V případě automatizovaných testů platí, že co je jednou testováno, bude testováno i při budoucích úpravách – v tomto ohledu jsou bezesporu efektivnější.

V PHP lze k automatizovanému testování využít např. rozšířené nástroje PHPUnit (pro jednotkové testování) a Selenium (pro funkční testování).

Tato metodika tedy nechává výběr způsobu testování na vývojářích (s vědomím toho, že manuální testování je méně bezpečné). Ale důležité je vůbec nějak testovat, a to po každém kroku refaktoringu (viz příslušná praktika). Doporučením je testovat alespoň důležitá a často používaná místa v kódu aplikace automatizovaně.

P4. Refaktorovat s ohledem na testovatelnost kódu

Pokud budou používány automatizované testy, je třeba psát kód s ohledem na jeho dobrou automatickou testovatelnost. Při refaktoringu je na to třeba brát zřetel.

Pokud kód, který se chystáme refaktorovat, není dobře testovatelný, pak je třeba refaktorovat jej do této podoby pouze s manuálním testováním, i pokud v projektu provádíme testování automatizované (protože to v dané situaci není dost dobře možné). Prioritou při takovém refaktoringu by tedy mělo být dostat kód do stavu, kdy je možné jej automaticky testovat, a pak se teprve pustit do dalšího refaktoringu.

Dopsáním testů do existujícího kódu, který byl převzat od jiných vývojářů a který neobsahuje testy, se obšírněji zabývá např. [Feathers, 2005].

P5. Refaktorovat podle doporučeného postupu

Pro většinu refaktoringů lze nalézt doporučený postup, nejlepší referencí je v tomto právě [Fowler, 1999] s poměrně obsáhlým katalogem refaktoringů a jejich postupů. Je vhodné se vždy takovým postupem řídit. Postupy jsou totiž navrženy tak, aby minimalizovaly riziko

zanesení chyb, a to jak rozložením na minimální kroky, tak častým spouštěním testů. To je pro vývoj důležité a je to také jedním ze základních principů této metodiky.

P6. Minimalizovat velikost kroků

Tato praktika souvisí s tou předchozí. Čím menší kroky vývojář při refaktoringu dělá, tím menší pravděpodobnost je, že udělá chybu. Jedním krokem se myslí přesun mezi stavy během refaktoringu, ve kterých aplikace prochází testy.

Často se zdá snaha o minimalizaci velikosti kroků zbytečná nebo nudná, proto lze pochopit tendenci vývojářů dělat více kroků najednou. Pokud vše funguje, je to v pořádku. Ale pokud se příliš často objevují při refaktoringu chyby, je lépe zmenšit velikost kroků. Oprava chyb totiž nakonec může být dražší (z hlediska času a tedy i peněz) než si práci zdánlivě zpomalit zmenšením kroků.

Některé refaktoringy mohou trvat několik hodin, dní či dokonce měsíců [Beck, 2005]. A u těch je rozdělení do malých kroků nutností. Nelze pracovat několik hodin, aniž by vývojář otestoval, zda vše funguje.

P7. Testovat po každém kroku

Po každém z těch minimálních kroků je třeba výsledek otestovat. Pokud funguje, lze provést další krok. Pokud ne, je velká šance na odhalení chyby po pár vteřinách přemýšlení či při prvním pohledu zpět do kódu.

P8. Neladit chyby vzniklé refaktoringem, ale zmenšit velikost kroků a provést je postupně

Tuto praktiku radí Kent Beck ve [Fowler, 1999] a souvisí to s praktikou *Minimalizovat velikost kroků*. Pokud udělá vývojář více kroků najednou a teprve poté testuje, je těžší odhalit případné chyby. Snad každý vývojář bude mít tendenci chybu hledat a opravit (ladit). Beck ale důrazně doporučuje raději vrátit provedený refaktoring do posledního funkčního stavu a zmenšit velikost kroku. Důvodem je to, že pokud chyba nastane při malém kroku, mnohem snáze se hledá než ta, která vznikne provedením několika kroků najednou. Takto může vývojář hledáním chyby strávit 5 vteřin, 5 minut nebo 5 hodin – nelze to předem odhadnout. Naproti tomu lze říci, že vrácení provedených kroků, na kterých vývojář pracoval např. 1 hodinu a jejich následné provedení postupně zabere napodruhé mnohem méně času, např. 10 minut.

Dlouhodobě je tato strategie časově výhodnější, jak uvádí Beck. Ovšem je k tomu třeba velké disciplíny, protože tendence hledat chybu ihned je opravdu velká.

P9. Refaktorovat ihned po dokončení menšího úkolu

Tuto praktiku popisuji a zdůvodňuji výše v podkapitole *Motivy*, ovšem dle mého názoru je natolik významná a přitom opomíjená, že ji zmiňuji i na tomto místě mezi praktikami.

P10. Rozhodovat o (ne)provedení refaktoringu podle jeho předpokládaného rozsahu

Malý refaktoring, který nebude trvat dlouho, provádí vývojář na základě vlastního rozhodnutí, které nemusí s nikým konzultovat. Takový refaktoring bude mít pravděpodobně ve výsledku pozitivní vliv na jeho celkovou produktivitu.

Refaktoring většího rozsahu už by měl konzultovat s vedoucím týmu, popřípadě i s jeho dalšími členy. Větší refaktoring již totiž zabere poměrně dost času a nemusí to být vhodné např. vzhledem k termínu odevzdání.

Záměrně zde přesně nespecifikuji, kde je hranice mezi malým a větším refaktoringem, to je individuální podle situace. Pro představu lze říci, že pokud během jednoho dne na 7 hodin práce vyjde např. hodina refaktoringu a tento je typicky rozložen během dne, pak je to malý refaktoring, za který je zodpovědný sám vývojář. Naopak větší refaktoring se dá do určité míry specifikovat tím, že se kód aplikace upravuje na mnoha místech a obsahuje více „katalogových“ refaktoringů (podle katalogu refaktoringů ve [Fowler, 1999].

P11. Dát refaktoringu vysokou prioritu

V praxi se velmi často na refaktoring nedostane (nebo jen v omezené míře), protože na něj „není čas“. V překladu to znamená, že priority vývojářů jsou jinde, typicky u vývoje nové funkcionality.

Někdy to může být opravdu výhodnější, zvláště když se s kódem v určité části aplikace nebude v nejbližší době pracovat a blíží se termín odevzdání (který se ostatně blíží vždy). Pak by si měl vývojář zapsat danou oblast kódu do svého seznamu úkolů. Jakmile se později ukáže, že absence refaktoringu brzdí další vývoj, měl by se k němu vrátit. Je ovšem možné, že se to vůbec nebude třeba. Z praktického hlediska je pak v pořádku netrávit jím čas a zaměřit se na oblasti k refaktoringu s lepším poměrem „cena/výkon“.

Ovšem jakmile se s kódem dále pracuje, doporučuji refaktoring provést, a to co nejdříve. Je to nejefektivnější a vynaložený čas se s dalším vývojem pravděpodobně vrátí. Toto jde jistě proti zvykům nejednoho vývojáře, ovšem zde je třeba zdůraznit princip maximalizace efektivity. Využitelnost této metodiky v praxi tím jistě neklesne. Dle mých osobních zkušeností je třeba si tento způsob zkrátka vyzkoušet a vývojář brzy pochopí jeho výhody a bude se moci zodpovědněji rozhodovat, zda refaktoring provést či odložit.

P12. Používat systém na správu verzí

Tato praktika je dnes již při vývoji softwaru běžná. Uvádím ji zde proto, že vývojářům umožňuje vrátit se k předchozímu funkčnímu stavu aplikace, pokud se refaktoringem zanese do aplikace chyba. Případně je pak možné vrátit změny provedené v rámci refaktoringu, který ale nebyl úspěšný (např. se zjistilo, že jeho dokončení je příliš neefektivní nebo že nevede do lepší situace, než byla ta původní).

P13. Optimalizovat až refaktorovaný kód

Mnoho vývojářů má tendenci psát „efektivní“ kód, který poběží rychleji. Bohužel to bývá na úkor jeho čitelnosti. Přitom tento přístup nemá příliš dobré výsledky.

Mnohem lepší je nestarat se o výkon (rychlost) aplikace a potřebnou funkcionalitu, dokud není funkcionalita hotova a kód refaktorován. Teprve pak je výhodné pustit se do optimalizací.

Ve skutečnosti totiž pomalou aplikaci zpomaluje malá oblast kódu, která se dá měřením odhalit a zaměřit se na ni. Pokud se vývojář snaží optimalizovat vše, co píše, dělá drtivou většinu této práce zbytečně, právě proto, že jen u malého zlomku kódu ve skutečnosti hrají nějaké optimalizace roli. Navíc při průběžné snaze o optimalizace je výsledkem hůře čitelný kód, což poté zase zpomaluje vývoj.

A optimalizovat refaktorovaný kód je většinou mnohem jednodušší a efektivnější.

P14. Refaktorovat i cizí kód, je-li to třeba

[Beck, 2005] doporučuje, aby byl kód vlastněn všemi. Navíc říká, že kdo uvidí příležitost zlepšit kvalitu kódu, musí to udělat, i když není autorem toho kódu. Toto je jedna ze zásad extrémního programování.

Tato metodika se spokojí s tím, že vývojář je primárně zodpovědný za refaktoring svého kódu, ovšem je-li to třeba, může zasáhnout i do kódu napsaného někým jiným. Důvodem je to, že jinak může refaktoring zůstat neúplný („bylo by skvělé změnit ještě něco v tom druhém modulu, ale to už není můj kód, takže bohužel“), což není tak efektivní jako jít za hranice svého kódu. Ovšem k tomu je samozřejmě třeba dobrá komunikace mezi programátory, zvláště mezi autory obou částí kódu. Tyto případy jsou ideální příležitostí pro refaktoring v páru, což opět doporučuje extrémní programování s tím, že to je efektivnější [Beck, 2005], ovšem tato metodika to, s ohledem na stávající praxi vývoje webových aplikací, nevyžaduje.

Nepřebírám zde Beckovu zásadu společného vlastnictví a refaktoringu kódu beze zbytku, jelikož se domnívám, že pro její úspěšné použití je třeba používat metodiku extrémního programování důsledněji, nelze si z něj vzít jen tuto jednu praktiku a vyvíjet podle jiné metodiky vývoje softwaru. Navíc se dá říci, že refaktoring svého kódu je efektivnější než refaktoring kódu cizího.

Pokud je vlastnictví kódu v projektu striktně rozděleno a nelze editovat cizí kód, je třeba o to více komunikace v týmu. Lze doporučit komunikaci formou společných diskuzí nad kódem s tím, že vývojář iniciující refaktoring vysvětlí své požadavky autorovi kódu a společně se doberou řešení. Jak je z tohoto zřejmé, vyšší nároky na komunikaci předpokládají, že vývojáři sdílí jednu místnost či alespoň budovu – komunikace na dálku totiž nikdy nebude tak efektivní, viz opět [Buchalcevová, 2009].

P15. Porozumět cizímu kódu rychlým refaktoringem

[Feathers, 2005] navrhuje zajímavou metodu pro pochopení cizího kódu. Nazývá ji *scratch refactoring*, v českém překladu [Feathers, 2009] *rychlé refaktorování*. Jde o to, že vývojář provádí na cizím kódu refaktoring, čímž jej lépe pochopí. Klíčové ovšem je, že pro kód nemusí mít testy a nemusí jej koneckonců ani často spouštět, jelikož jediným účelem refaktoringu je kódu porozumět, nikoli vylepšit strukturu programu. Jinými slovy, změny provedené refaktoringem vývojář neuloží. Zdá se to jako neefektivní způsob, ovšem Feathers jej obhajuje. Díky tomu, že tento druh refaktoringu nevyžaduje takovou opatrnost a preciznost, nezabírá tolik času, a tedy se opravdu může vyplatit.

Ještě doplním, že [Fowler, 1999] také navrhuje refaktoring pro lepší pochopení cizího kódu, ovšem už nemluví o neukládání provedených změn, což je jistě také alternativa.

Dle mého názoru záleží především na tom, jaké máme s daným kódem další záměry – přebíráme-li jej a budeme jej dále editovat a rozšiřovat, pak se pravděpodobně vyplatí klasický refaktoring. V opačném případě bude stát za zkoušku rychlý refaktoring. Samozřejmě to ale nelze dogmaticky určit, záleží na situaci a posouzení vývojáře.

P16. Refaktorovat v páru

Tuto praktiku doporučuje Beck ve [Fowler, 1999]. Programování v páru obecně (ne jen při refaktoringu) je jedna z praktik extrémního programování [Beck, 2005]. Výhodou je, že zatímco jeden refaktoruje, druhý jej hlídá podle hesla „víc očí víc vidí“, mohou (a měli by) spolu probírat aktuální postup a v neposlední řadě dohlížející vývojář může podpořit druhého v dalším refaktoringu, či naopak jej včas zastavit, jelikož při refaktoringu bývá někdy těžké rozpoznat, kdy je na čase s refaktoringem přestat.

Předpokládám, že tato praktika bude pro refaktorující vývojáře jedna z těch náročnějších, jelikož programování v páru si zatím nezískalo velkou popularitu, zejména kvůli jeho diskutabilní efektivitě. V obou citovaných knihách lze nalézt více informací a snad i motivaci tento přístup vyzkoušet.

5.12 Nástroje pro refaktoring v PHP

Refaktoring mohou usnadnit různé nástroje. V PHP ovšem není jejich nabídka automatizovaných refaktoringů příliš velká, jedná se především o přejmenování (proměnných, metod, tříd, souborů), méně často pak extrakci metod a další. Důvodem je to, že PHP je dynamický jazyk[12], a v takových jazycích je náročnější automaticky analyzovat kód, a tedy automatizovat refaktoring. Více informací o automatické analýze kódu dynamických programovacích jazyků a automatizaci refaktoringu v PHP lze nalézt např. v [Beňo, 2013].

V *Tabulce 5-4* uvádím seznam některých nástrojů pro automatizovaný refaktoring. Neuvádím ovšem přesný výčet refaktoringů, které podporují, jelikož by brzy tak jako tak nebyl aktuální. Místo toho raději u každého nástroje odkazuji na domovskou stránku, kde se

[12] http://en.wikipedia.org/wiki/Dynamic_programming_language

vývojář může o nástroji dozvědět vše potřebné a případně jej i vyzkoušet. Také rovnou v tabulce uvádím, zda je nástroj součástí integrovaného vývojového prostředí (IDE[13]) či je to samostatný nástroj (ovládaný z příkazového řádku).

Tabulka 5-4: Seznam některých nástrojů pro automatizovaný refaktoring (zdroj: autor)

Název	IDE vs. samostatný nástroj
Netbeans[14]	IDE
Eclipse PDT[15]	IDE
Zend Studio[16]	IDE
PHPStorm[17]	IDE
PhpED[18]	IDE
PHP refactoring browser[19]	samostatný nástroj
Scisr[20]	samostatný nástroj
Rephactor[21]	samostatný nástroj

Kromě automatizovaných refaktoringů lze také v některých IDE narazit na statickou kontrolu kódu, která upozorňuje na různé prohřešky proti vhodnému stylu psaní kódu (*best practices*), což napomáhá psaní kvalitnějšího kódu. Například NetBeans lze propojit s některými nástroji na statickou analýzu PHP kódu, v době psaní této práce to byl PHP Code Sniffer a PHP Mess Detector (opět zde odkazuji na stručný popis těchto nástrojů v práci [Hujer, 2012]).

5.13 Přínosy refaktoringu

Pro větší motivaci by měli vývojáři znát přínosy refaktoringu, proto je uvádím i na tomto místě. [Fowler, 1999] uvádí 4 důvody, proč refaktorovat:

1) Zlepšení designu existujícího softwaru

S vývojem se design aplikace nutně stává méně a méně přehledným. Bez dostatečného refaktoringu je stále těžší jej modifikovat a přidat novou funkcionalitu. Roste riziko chyb a

[13] IDE – Integrated Development Environment
[14] https://netbeans.org/features/php/editor.html
[15] http://projects.eclipse.org/projects/tools.pdt
[16] http://www.zend.com/en/products/studio/features
[17] http://www.jetbrains.com/phpstorm/features/
[18] http://www.nusphere.com/products/refactor_php.htm
[19] http://qafoo.com/blog/041_refactoring_browser.html
[20] http://iangreenleaf.github.io/Scisr/
[21] http://rephactor.sourceforge.net/

klesá efektivita vývoje. Aby byla aplikace udržovatelná, je třeba kód pravidelně „uklízet“ refaktoringem.

2) Snazší porozumění kódu

Kód je kvalitnější a také čitelnější. Dodržuje konvence (projektové či zvolené obecné) a neobsahuje duplicity. To všechno přispívá k tomu, že vývojář porozumí cizímu kódu (nebo i svému po určité době) rychleji a snáze. Navíc se pravděpodobně díky lepšímu porozumění dopustí méně chyb.

3) Snazší hledání chyb

V nekvalitním a nepřehledném kódu je snadné udělat chybu a naopak může být těžké ji odhalit. Díky přehlednějšímu kódu je snazší se chybám vyhnout. Zvláště v kontextu testování je třeba, aby kód byl dobře testovatelný (což lze označit za jeden z aspektů kvality kódu).

4) Rychlejší vývoj

Přestože refaktoring může být časově náročný, dlouhodobě se vyplatí. Krátkodobě je možné bez refaktoringu vyvíjet rychleji, ale vzrůstající složitost a nepřehlednost kódu bude vývoj čím dál více zpomalovat. Souvisí to se všemi předchozími body.

První tři body jsou poměrně jasné a těžko proti nim lze něco namítat. V posledním bodu je ale zádrhel. Bez zkušeností s refaktoringem může být těžké mu uvěřit. Proto může snadno vyvstat problém, kdy projektový manažer vyžaduje od vývojářů vývoje nové funkcionality a nemá pochopení pro refaktoring. Zde je třeba, aby mu vedoucí vývojářů vysvětlil výše zmíněné přínosy a přesvědčil jej o závislosti efektivity vývoje na refaktoringu. Je samozřejmě možné, že manažer bude i nadále trvat na svém. Fowler pak doporučuje nic mu neříkat a zkrátka refaktorovat, kdy je to třeba. Argumentuje tím, že odpovědností vývojářů je vyvíjet software co nejefektivněji. Proto pokud vědí, že refaktoring efektivitu zvyšuje, měli by jej používat, podobně jako jiné postupy a nástroje pro zvýšení efektivity.

5.14 Rizika refaktoringu

Existuje několik rizikových faktorů, které každý refaktoring obsahuje a které je třeba zvážit. V některých situacích mohou převážit nad přínosy refaktoringu, a pak je lépe se do refaktoringu nepouštět. V opačném případě je i tak vhodné mít je na paměti.

Nedostatek času

Když se blíží termín odevzdání či jiné časové ohraničení, může být vhodnější refaktoring odložit, aby bylo projekt možné v termínu stihnout. Ovšem nemělo by to sloužit jako výmluva, a už vůbec ne dlouhodobě. V krátkodobém horizontu se snížená kvalita kódu nemusí projevit, ovšem pokud se refaktoring opomíjí déle, bude se to téměř jistě projevovat na celkové efektivitě vývoje.

Nedostatečné či chybějící testy

Absence automatizovaných testů znamená větší riziko zanesení chyb. Proto je třeba brát na tuto skutečnost zřetel při rozhodování o tom, zda refaktoring provést. U webových aplikací nemusí být dopady zanesení chyby tak velké jako např. u softwaru pro řízení letového provozu, může tedy být toto riziko plně akceptovatelné.

Nezkušení vývojáři

Již jsem v této práci několikrát zmínil nároky na kvalitu vývojářů a jejich dovednosti psát kvalitní kód a rozumět alespoň základům refaktoringu. Jejich případná nezkušenost je totiž dalším rizikem, na které je třeba brát zřetel, a to nejen při rozhodování o (ne)provedení refaktoringu, ale i při samotném procesu refaktoringu, kdy například na méně zkušené vývojáře dohlížejí jejich zkušenější kolegové – pro to jsou vhodnou aktivitou již dříve zmiňované revize kódu.

Když by bylo lépe kód přepsat

Někdy je kód natolik nekvalitní a jeho refaktoring by vyžadoval tolik času, že je vhodné zamyslet se nad možností napsat jej znova. Nemusí to znamenat přepisovat celou aplikaci, ale jen část, např. jednu třídu či jen metodu.

5.15 Faktory od refaktoringu odrazující

Kromě výše uvedených rizik existují další faktory, které mohou vývojový tým od refaktoringu odrazovat. S nimi je třeba si poradit, jinak nebudou vývojáři k refaktoringu motivováni, nebudou vnitřně přesvědčeni o jeho prospěšnosti v konkrétní situaci či obecně. Obecnou radou je dodatečné vzdělání v této problematice a více předaných zkušeností od zkušenějších kolegů. V této části volně vycházím z [Tinkham, 2005].

Nedostatečné porozumění kódu aplikace

Nejjednodušeji se refaktoruje vlastní kód. Naopak kód, jehož autorem je někdo jiný a ve kterém se vývojář nevyzná, situaci příliš nezjednodušuje. Na druhou stranu, je vhodné použít refaktoring právě pro lepší porozumění kódu, viz praktika *Rychlý refaktoring*.

Nedostatečné pochopení refaktoringu

Pokud vývojář v refaktoringu do určité míry tápe, může k němu mít samozřejmě jistý odpor, ať už proto, že přesně neví, jak postupovat nebo co refaktorovat, nebo proto, že neocení jeho přínosy. Je vhodné si tyto záležitosti ujasnit hned na začátku projektu a shodnout se na společných předpokladech a prioritách. Pokud to není z nějakého důvodu možné (zvláště v případě zavádění této metodiky do existujícího projektu), lze takovému vývojáři doporučit studium příslušné literatury (viz výše).

Pokud vývojáři neocení přínos refaktoringu, nebudou mít samozřejmě ani motivaci jej provádět. Lze je samozřejmě do jisté míry motivovat zvnějšku (odměnami za dodržování pravidel a tresty za jejich porušování), ale tato motivace nebude nikdy tak dobrá jako ta vnitřní. Tu může vývojář mít, jen pokud pochopí a ocení přínosy refaktoringu a nemá tak pocit, že to dělá jen proto, že musí.

Krátkodobé zaměření

Refaktoring se vyplácí převážně v dlouhodobém horizontu. Proto pokud se vývojáři zaměřují jen na jeho krátkodobé efekty, může je to demotivovat. Je opět třeba dbát na dostatečné vzdělání vývojářů v tomto směru. Lze jít také cestou stanovení pravidel, při jejichž dodržování se vývojáři sami po určité době přesvědčí o výhodnosti refaktoringu.

Žádná nová funkcionalita

Jednou ze základních charakteristik refaktoringu, vyplývající přímo z definice, je to, že není přidávána žádná nová funkcionalita. To může vývojáře také demotivovat, zvláště pokud je systém špatně nastaven a projektový manažer nebo zadavatel vyvíjejí tlak na přidávání nové funkcionality bez ohledu na potřebu časové rezervy pro refaktoring. Tento jev je bohužel častý a z pozice vývojáře není jednoduché mu předcházet. Lze zde navrhnout jednak snahu o lepší komunikaci s nadřízenými (aby pochopili, že refaktoring v dlouhodobém horizontu efektivitu vývoje zvyšuje) a jednak zahrnutí refaktoringu tak jako tak s tím, že jej vývojář nebude nikde explicitně zmiňovat (bude jej brát doslova jako nezbytnou součást vývoje požadované funkcionality).

Strach ze zanesení chyb do aplikace

V aplikaci se mohou objevit oblasti kódu, které jsou komplikované a velmi špatně pochopitelné. V důsledku toho se s nimi velmi špatně pracuje. Je-li třeba přidat novou funkcionalitu v této oblasti, často to skončí tak, že ji vývojář k původnímu kódu nějak „přilepí“, což samozřejmě kódu na kvalitě (a bezchybnosti) nepřidá.

Strach vývojáře ze zanesení chyb do aplikace dále zvyšuje absence automatizovaných testů. Tyto dva problémy v kombinaci mohou vést k silnému odporu vývojářů kód dále jakkoliv upravovat (včetně refaktoringu).

Nelze než opět doporučit psaní automatizovaných testů a testovatelného kódu. Další užitečnou metodou jsou již mnohokrát zmiňované revize kódu, které zvyšují čitelnost kódu. Variantou je také technika z extrémního programování – programování (nebo refaktoring) v páru [Beck, 2005], kdy je o něco obtížnější udělat chybu.

5.16 Zavedení metodiky

Jedním ze základních principů této metodiky je její dobrá použitelnost v praxi. K tomu patří také její snadné zavedení do projektu (nového či již probíhajícího). Z toho důvodu je tato metodika koncipována tak, aby se dala aplikovat i nekompletní – např. lze implementovat jen některé praktiky. Největší efekt ale bude mít, bude-li implementována celá.

Výhodou je, že menší či větší část této metodiky již vývojové týmy často aplikují (neboť také vycházím z praxe). Pokud ne, nic jim nebrání zavádět metodiku postupně takovou rychlostí, jaká jim bude vyhovovat.

Úspěšné zavedení této metodiky má jisté předpoklady. Zde se volně inspiruji prací [Mittner, 2011]. Důležitým předpokladem je přijetí metodiky vývojovým týmem. K tomu je třeba, aby jí vývojáři porozuměli, a to bez větší časové investice. Proto je jednoduchá. Dále je pro motivaci vývojářů důležité, aby viděli její přínosy. Ty spočívají především v přehledném a uceleném pohledu na refaktoring v PHP, který je zároveň relativně jednoduchým návodem.

Metodika musí být také veřejně dostupná, což tato práce splňuje. Do budoucna bude ale vhodnější vytvořit pro ni oficiální webovou stránku a případně další podpůrné materiály, např. seznam praktik volně ke stažení. Lze také využít nástroj Eclipse Process Framework Composer[22], jak pro svou metodiku navrhuje také [Mittner, 2011].

Dále je vhodné používat nástroje usnadňující refaktoring. Viz výše v podkapitole *Nástroje pro refaktoring v PHP*.

5.17 Shrnutí

V této kapitole jsem popsal navrhovanou metodiku. Pro praktickou aplikaci je důležitý algoritmus uvedený v podkapitole *Postup při refaktoringu*, a to včetně dalších podkapitol, ve kterých jsou jednotlivé kroky popsány podrobněji.

Popsal jsem také přínosy a rizika refaktoringu, stejně jako faktory, které mohou vývojáře od refaktoringu odrazovat. Dle mého názoru je důležité, aby byli vývojáři s těmito záležitostmi obeznámeni, jelikož jen tak se mohou zodpovědně rozhodovat o provedení a rozsahu refaktoringu.

[22] http://www.eclipse.org/epf/

6 Ukázka aplikace metodiky

Cílem této části je předvést praktickou aplikaci navrhované metodiky na reálném projektu. Dílčích cílů je několik. Zaprvé je to zmapovat postup podle algoritmu doporučovaného metodikou. Druhým dílčím cílem je uvést a okomentovat skutečný průběh refaktoringu, a to na vhodné úrovni detailu. Posledním dílčím cílem je analyzovat aplikaci a porovnat a zhodnotit výsledky analýzy před a po refaktoringu. Analýza bude provedena primárně nástrojem PDepend[23], více viz podkapitola *Identifikace oblastí vhodných pro refaktoring* dále v této kapitole.

6.1 Webová aplikace Doučka.cz

Webový projekt Doučka.cz zprostředkovává doučování mezi studenty a lektory. Student ve webové aplikaci vyplní formulář a jeho objednávka se uloží do databáze. Lektoři, kteří se zaregistrují a vyplní povinné údaje, si mohou takovou objednávku převzít, pokud se jim líbí. Student má také možnost vybrat si konkrétní lektory, v takovém případě se objednávka pošle pouze jim.

V aplikaci se také řeší zejména správa lektorských účtů, dobíjení tzv. kreditu (pro placení zprostředkovacího poplatku), rušení doučování, která neproběhla (včetně vracení kreditu lektorovi), změny parametrů objednávek (před i po převzetí lektorem) a hodnocení lektorů studenty. Projekt funguje již od roku 2008, doména doucka.cz ovšem vznikla až v červenci 2013 jako rozcestník pro celkem 9 měst. Každé fungovalo a nadále funguje samostatně na vlastní doméně[24], ovšem webová aplikace existuje pro všechny tyto domény pouze jedna, a ta je právě předmětem této praktické aplikace navrhované metodiky.

Vývoj aplikace probíhal průběžně po celou dobu. Autorem kódu aplikace jsem já, s výjimkou knihoven zahrnutých ve zvláštním adresáři (viz dále).

[23] http://pdepend.org/
[24] doucovanipraha.cz, doucovanivbrne.cz, doucovaniostrava.cz, doucovanivplzni.cz, doucovaniolomouc.cz, doucovaniliberec.cz, doucovanibudejovice.cz, doucovanihradec.cz, doucovanipardubice.cz

6.1.1 Struktura

Aplikace je postavena na Zend Frameworku[25] ve verzi 1.11. Zend Framework staví na návrhovém vzoru MVC[26]. Tento vzor odděluje aplikační logiku (model) od prezentační (views), kdy tyto dvě vrstvy spojuje controller.

Dále uvádím strukturu adresářů aplikace.

Kód 6-1: Struktura aplikace Doučka.cz (zdroj: autor)

```
application
    modules
       admin
            controllers
            views
        default
            controllers
        lector
            controllers
            views
    JM
    model
cache
config
languages
    cs
    sk
library
    SES
    simpledom
    smsconnect
    Zend
logs
public
    css
    icons
    images
    js
tests
```

[25] http://framework.zend.com/
[26] model-view-controller, viz např. http://cs.wikipedia.org/wiki/Model-view-controller

Struktura je poměrně jasná, okomentuji ji tedy jen stručně. Složka *application* obsahuje veškerou aplikační logiku (s výjimkou zahrnutých knihoven). Aplikace pracuje s celkem třemi moduly, *admin, lector a default.* Adresář JM ukrývá moje knihovní funkce, většina tříd je tam děděna od tříd Zend Frameworku. V adresáři *model* jsou pak třídy pro práci s daty (vztaženo k databázi).

Složka *cache*[27] obsahuje soubory napomáhající rychlejšímu běhu aplikace, *config* pak obsahuje soubor s konfigurací aplikace. V adresáři *languages* jsou uloženy překlady řetězců použitých v aplikaci, to kvůli překladu do dalších jazyků. V současné době existuje neúplný překlad do slovenštiny (ovšem projekt na Slovensku zatím nefunguje).

Ve složce *library* jsou pak externí knihovny, v pořadí uvedeném výše slouží k odesílání e-mailů přes SES[28], práci se strukturou HTML (DOM[29]) a posílání SMS zpráv. Poslední složka, *Zend*, obsahuje Zend Framework.

Složka logs obsahuje logy[30], neboli soubory, kam se ukládají záznamy o běhu aplikace. Složka tests obsahuje testy.

Ve složce public je pak front-end aplikace, konkrétně CSS[31] a JavaScript, dále také obrázky a ikony.

6.1.2 Vhodnost metodiky pro tento projekt

Předpoklady pro použití metodiky, které uvádím výše v příslušné podkapitole, jsou splněny. Je třeba explicitně poznamenat, že i při refaktoringu zůstávám jediným vývojářem, čímž se situace na jednu stranu zjednodušuje o komunikaci v týmu, na druhou stranu je to ochuzení o možnost revizí kódu a refaktoringu v páru.

6.2 Postup dle metodiky

6.2.1 Motiv a cíle

Primárním motivem pro refaktoring je v tomto případě příchod nového vývojáře do týmu (v metodice motiv M10). Dílčím motivem je pak klesající čitelnost a udržitelnost kódu (M8).

[27] cache - http://cs.wikipedia.org/wiki/Cache
[28] SES – Amazon Simple Email Service; http://aws.amazon.com/ses/
[29] DOM – Document Object Model; http://cs.wikipedia.org/wiki/Document_Object_Model
[30] log – http://cs.wikipedia.org/wiki/Log
[31] CSS – Cascading Style Sheets; http://cs.wikipedia.org/wiki/Kask%C3%A1dov%C3%A9_styly

Oba tyto motivy spolu souvisejí, ovšem prvně jmenovaný je tím rozhodujícím impulsem k refaktoringu.

Z hlediska obou uvedených motivů je hlavním cílem zvýšit srozumitelnost designu a kódu aplikace do té míry, že i nový vývojář se bude schopen snadno zorientovat a pokračovat ve vývoji. Dílčím cílem zde je odstranit nejzávažnější problémy zjištěné při statické analýze kódu. Druhým dílčím cílem je, aby každá třída měla jen jednu odpovědnost.

6.2.2 Identifikace oblastí vhodných pro refaktoring

K identifikaci jsem použil metody uvedené v příslušné podkapitole dříve v této práci. Nejprve jsem si sepsal problémová místa, o kterých jsem věděl.

Pomocí nástroje PDepend jsem zjistil místa se zásadními problémy. Pro identifikaci oblastí nekvalitního kódu jsem se zaměřil především na metriky NPath[32], CCN2[33] a WMC[34]. První dvě uvedené jsem využil pro identifikaci příliš komplexních metod, WMC pak pro nalezení příliš komplexních tříd. Zajímavé je, že tyto komplexní třídy obsahovaly většinu metod identifikovaných zbylými dvěma metrikami. Je to ovšem logické, jelikož PDepend počítá WMC dané třídy jako součet CCN2 jejích jednotlivých metod.

Využil jsem také nástrojů PHP Mess Detector[35] a PHP Code Sniffer[36]. Ovšem v aktuální situaci mi nepomohly tolik jako PDepend, jelikož úroveň detailu byla příliš velká, oba se zaměřují především na dodržování konvencí v kódu. To je sice také potřebné, ale bylo výhodnější věnovat se nejprve větším a rozsáhlejším refaktoringům, podle podkapitoly *Výběr oblasti pro refaktoring*.

Následovala revize návrhu aplikace. Sepsal jsem odpovědnosti tříd modelu, controllerů a také těch ve vlastní knihovně v adresáři JM. Výsledný seznam uvádím v příloze A. Při této metodě jsem zároveň využil také metodu manuálního procházení kódu, byť jsem neprocházel jiný kód než ten výše uvedených tříd. Tato revize odpovědností tříd byla velice přínosná a držel jsem se jí při rozdělování velkých tříd na menší s jedinou odpovědností.

[32] http://phpmd.org/rules/codesize.html#npathcomplexity
[33] CCN – Cyclomatic Complexity Number; http://pdepend.org/documentation/software-metrics/cyclomatic-complexity.html
[34] WMC – Weighted Method Count; http://pdepend.org/documentation/software-metrics/weighted-method-count.html
[35] http://phpmd.org/
[36] http://pear.php.net/package/PHP_CodeSniffer/

Další dvě metody identifikace problémových oblastí, profilování a kontrolní seznamy, jsem neuplatnil, jelikož předchozí metody mi již tak poskytly dostatečný seznam oblastí pro refaktoring. Je ale možno je využít v budoucnu.

Výsledkem této identifikace bylo zejména několik problémových míst, kde byl refaktoring třeba především. Zajímavé bylo, že na tato místa ukazovalo více metrik statické analýzy kódu a také že to byla často místa, která jsem si dokázal vybavit i po paměti. V příloze B pak uvádím metody s nejvyššími hodnotami zmiňovaných metrik a jejich novými hodnotami po refaktoringu.

6.2.3 Výběr pachů k odstranění

Na základě výše uvedené identifikace jsem získal seznam problémů, které bylo třeba refaktorovat či „opravit“. Dalším dílčím cílem v této fázi bylo vybrat, čím se zabývat. V tomto případě jsem se rozhodl jít cestou prioritizace. Seřadil jsem nalezené problémy podle kritérií (tipů) uvedených v příslušné podkapitole výše.

Zásadním faktorem při tomto řazení byly závislosti jednotlivých problémů. Většina z nich byla na sobě závislá jen velmi mírně (prakticky jen tím, že se týkaly shodných oblastí kódu). Ovšem například zeštíhlování controllerů záviselo na potřebě vytvořit speciální třídy pro skládání e-mailových zpráv a pro odesílání samotných e-mailů (které se do té doby často odesílaly přímo v controlleru). Snažit se nejprve zeštíhlit controller bez věnování pozornosti odesílání e-mailů z aplikace by bylo neefektivní.

Dále jsem podle další zásady výběru pachů k odstranění zařadil obtížnější, rozsáhlejší problémy (pokud nebyly závislé na nějakých dalších). Zde jsem se držel výsledků statické analýzy.

Další průběh jsem již více neplánoval a počítal jsem s tím, že na další problémové oblasti narazím průběžně.

6.2.4 Testování

Refaktoring je třeba mít pojištěn testováním. Jak jsem uvedl výše, v praxi často automatické testy chybějí, u webových aplikací psaných v PHP to podle mých zkušeností je spíše pravidlem než výjimkou. Ovšem je třeba testovat alespoň nějak, nikdy není možné se spoléhat na to, že jsem refaktoring provedl správně, aniž bych si to ověřil.

V tomto případě jsem se spolehl nejčastěji na manuální testování, jelikož většinou stačilo znovu načíst stránku a případné chyby byly okamžitě odhaleny. Pokud se takový test skládal z více úkonů (načíst stránku, udělat určitou operaci, např. vyplnit formulář, odeslat jej a zkontrolovat výsledek), pak mi pomohla aplikace Selenium ve formě doplňku pro prohlížeč Firefox, která po nadefinování provedla akce mnohonásobně rychleji.

Nástroj PHPUnit jsem využil podstatně méně, ovšem v určitých místech (metodách) jsem byl opravdu rád, že jsem nemusel kód testovat manuálně.

S testováním souvisí i snaha o dobře testovatelný kód. Což mimo jiné znamená dbát na kratší metody a menší provázanost mezi třídami. Pokud je metoda dostatečně jednoduchá, stačí ji otestovat po napsání jednou manuálně a pak už je jasné, že funguje. To se samozřejmě týká jen některých metod, ale chci na tom ilustrovat, že psát testovatelný kód je velmi výhodnou praxí nejen při jednotkovém testování.

6.2.5 Průběh refaktoringu

Při popisu průběhu refaktoringu se zaměřím na zásadní věci a nebudu zbytečně zabíhat do podrobností, to není záměrem této práce.

Začal jsem tím, že jsem přesunul či úplně smazal některé soubory (a jejich funkcionalitu přesunul jinam).

Poté jsem si sepsal odpovědnosti jednotlivých tříd modelu a také odpovědnosti controllerů obecně. Většina z nich měla již před refaktoringem na starosti jen jednu věc. Ty, které jich měly více, uvádím v příloze A spolu s naznačeným řešením této situace. Tento soupis odpovědností mi pomáhal v počátečních fázích, kdy jsem se vypořádával s příliš velkými třídami. Například třídu *Orders*, která se starala původně o vše kolem objednávek doučování a která měla přes 1000 řádků kódu (bez komentářů), jsem rozdělil na tyto menší třídy: *OrderDetail* (poskytuje informace o konkrétní objednávce), *OrderInserter* (vkládá nové objednávky a zajišťuje vše, co je s tím spojeno), *OrderUpdater* (aktualizuje data o objednávce) a *OrdersUpdater* (aktualizuje data o více objednávkách najednou). Třídě *Orders* tak zůstala pouze odpovědnost získávat data o objednávkách (pro potřeby výpisů různých seznamů apod.). Tímto rozdělením jsem dosáhl výrazně vyšší přehlednosti, byť jednotlivé metody poté ještě vyžadovaly další refaktoring.

Samostatnou kapitolou byly controllery. Soupis jejich původních odpovědností uvádím taktéž v příloze A. Na základě této analýzy jsem vytvořil třídy *Mailer* (pro odesílání e-mailů), *MailMessageComposer* (pro skládání e-mailových zpráv), *FlashMessageComposer* (pro skládání delších tzv. flash zpráv[37]) a *LogMessageComposer* (pro skládání delších zpráv k logování). Také jsem přesunul části kódu do tříd modelu, případně do tříd formulářů (typicky to byla validace a dodatečné ošetření vstupních dat).

Po rozkladu příliš rozsáhlých tříd jsem se řídil především výsledky statické analýzy. Zaměřil jsem se na metody s nejvyššími hodnotami NPath a CCN2. Výsledky ukazují tabulky v příloze B. Vedlejším efektem kontroly těchto hodnot byl subjektivní pocit pokroku, který se dostavoval nejen po značném zjednodušení určité oblasti, ale právě i se snižováním naměřených hodnot jednotlivých metrik. Je ovšem třeba podotknout, že cílem nebylo hodnoty těchto metrik snížit. To bylo jen prostředkem ke kvalitnějšímu kódu. Tyto metriky obecně považuji za výborný ukazatel problémových míst, ovšem s tím, že je třeba vyvarovat se snižování těchto hodnot za každou cenu, jde především o zvyšování kvality kódu. Tedy čísla je možné zmenšit například vyjmutím části kódu do samostatné, nově vytvořené metody, ovšem pokud je extrahovaný kód nepřehledný, je třeba v refaktoringu pokračovat, přestože se hodnoty metrik dostaly do únosných mezí. Dále jsem také prozkoumal view skripty a extrahoval z nich většinu aplikační logiky, která se mohla odehrávat v controlleru či v modelu. Někdy jsem ale dospěl k závěru, že je lépe logiku nepřesouvat, jelikož by to znamenalo zanesení další složitosti. Záleželo tedy značně na aktuální situaci, přestože obecný princip je zřejmý, a to, mít ve view skriptech co nejméně aplikační logiky, jelikož tyto by měly sloužit jen k zobrazení dat uživateli.

Průběžně jsem také narážel na nekonzistence v různých konvencích – pokud lze jednu věc zapsat různými způsoby, je třeba si vybrat jeden a ten používat, což bohužel ve stávající aplikaci nebylo pravidlem. Například pokud lze získat určitá data pomocí různých metod (nebo jen parametrů jedné metody) jako objekt nebo jako pole. Při následném použití těchto dat v kódu se pak používala různá notace a vedlo to k horší čitelnosti kódu a snazšímu zanesení chyb. Při refaktoringu jsem měl skvělou příležitost tyto konvence sjednotit, jelikož jsem měl nejlepší celkový přehled o kódu.

[37] flash zpráva - zpráva zobrazená uživateli, zpravidla o výsledku právě provedené operace

6.2.6 Postřehy z průběhu refaktoringu

Při refaktoringu metod se ukázalo jako nezbytné si nejprve ověřit, zda je metoda v aplikaci skutečně používána a zda jsou používány všechny její parametry. Nezřídka jsem totiž objevil metodu, kterou jsem mohl bez jakéhokoli efektu smazat (či smazat některý z jejích parametrů). Toto byly pozůstatky kódu třeba i několik let staré, na které se během vývoje „zapomnělo“ při změně funkcionality na jiných místech. Při tomto mi byla velmi užitečným pomocníkem funkce *Find usages* (lze přeložit jako *Najdi použití*). Výsledkem je každopádně ušetření práce, jelikož smazat kód je samozřejmě jednodušší a rychlejší, než jej refaktorovat.

Když jsem se setkal s nějakou komplexní metodou, ve které jsem se příliš dobře nedokázal zorientovat, postupoval jsem po opravdu malých krocích. Prvním z nich bylo přeskládání kódu (pokud to bylo možné kvůli závislostem v kódu) tak, aby byly související věci pohromadě. Důvodem pro tento krok je to, že po přeskupení se objeví příležitosti vyjmout takový související kód do zvláštní metody (a případně jej dokonce přesunout do jiné třídy).

Jedním z ukazatelů na možnost takového přeskupení byla příležitost zkrátit životnost proměnné (počet řádků mezi jejím prvním a posledním použitím). Pokud do ní byla přiřazena hodnota na začátku metody, ale použita byla poprvé až na konci, pak bylo nasnadě přesunout přiřazení hodnoty těsně před toto první použití. Těmito změnami se kód často zpřehlednil, přestože tyto změny nestály příliš mnoho mentálního úsilí.

Jedním ze základních refaktoringů je vyjmutí kódu do samostatné metody. Během mé práce jsem si uvědomil, že toto je výhodné nejen pro zkrácení původního kódu. V takto vytvořené metodě lze totiž extrahovaný kód často zásadním způsobem zjednodušit, ať už proto, že je přehlednější (jelikož je jej méně), nebo proto, že je možno využít omezeného rozsahu metody (zásadní výhodou je vrácení hodnoty v určitém místě bez nutnosti dojít na konec, na rozdíl od původního kódu).

Osvědčilo se mi také eliminovat konstrukci *new* pro vytvoření nové instance třídy. Takové vytvoření třídy totiž vytváří v kódu novou závislost (objekt je předáván v proměnné), zatímco pokud lze objekt získat voláním určité metody, tato starost odpadá. V aplikaci jsem to realizoval pomocí třídy, která uchovává vytvořené instance všech tříd modelu (ve smyslu návrhového vzoru MVC, viz výše) a při volání třídy s daným jménem vrátí existující instanci (případně ji nejprve vytvoří, pokud dosud neexistuje).

Také se mi osvědčilo používat více statické metody, a to tam, kde není důležitý stav objektu. Například při skládání e-mailových zpráv. Zjednodušení bylo opět v tom, že ubyla starost s vytvářením objektu a jeho předáváním ve speciální proměnné. Tento přístup nemusí být výhodný v každé situaci, ale pro účely refaktorované aplikace posloužily velmi dobře.

Účinným způsobem, jak zvýšit bezpečnost aplikace a zároveň zabránit duplicitám, je definovat si úroveň, pod kterou již budu předpokládat, že proměnné na vstupu jsou ošetřeny (proti případným útokům podstrčením nesprávných hodnot). V této aplikaci nechávám ošetření vstupních dat od uživatele na controllerech a ve třídách modelu již předpokládám, že vstupní proměnné není třeba testovat. Při refaktoringu jsem objevil bezpočet míst, kdy byla proměnná kontrolována jak v controlleru, tak pak i ve volané třídě modelu.

Pokud stojím před refaktoringem metody, které je těžké porozumět, pak může pomoci sepsat si postupně vše, co tato metoda dělá. Dále je vhodné prověřit, zda kroky musejí následovat v tomto pořadí. Pokud ne, může pomoci kroky přeskládat do větších logických celků (například podle volaných tříd). V tomto popisu je pak již snazší se zorientovat a použít jej pro případnou extrakci metod (každý krok, který zapíšu, by ideálně měl být zajištěn jednou metodou, tedy pokud jej zajišťuje v původní metodě 10 řádků, ukazuje to, že tento kód je pravděpodobně možné vyjmout do nové metody).

Některé controllery byly v refaktorované aplikaci poměrně dlouhé. Rozhodl jsem se ale držet zásady štíhlého controlleru, která říká, že controller pouze zajišťuje interakci s uživatelem a o veškeré operace s daty se starají třídy modelu. Soupis odpovědností controllerů vyplývajících z tohoto přístupu uvádím v příloze A, stejně jako odpovědnosti, které jsem přesunul jinam, a tím controllery úspěšně zeštíhlil.

Co se týče praktik uvedených v této metodice, nejzajímavější pro mě byla snaha dělat vždy jen jednu věc. Často jsem měl nutkání udělat „odbočku“ a při refaktoringu určitého problému opravit i jiné věci, na které jsem narazil. Pomohlo ale vést si seznam takových odboček a nejprve vždy dokončit aktuální refaktoring. Zaprvé to bylo ve výsledku časově efektivnější, mimo jiné proto, že odstranit jeden nedostatek na více místech je efektivnější, než odstraňovat tyto nedostatky postupně v průběhu času. Zadruhé se tak snížilo riziko chyb – jelikož po chvíli už si člověk často nepamatuje na všechno a snáz tak něco přehlédne.

Další důležitou praktikou bylo postupovat v malých krocích. V několika situacích jsem se přesvědčil o tom, že pustím-li se do větších kroků, snadno pak něco přehlédnu či dokonce ztratím přehled. Souvisí to také s předchozím postřehem. Pokud někde udělám „odbočku“, dělám vlastně více věcí najednou.

Duplicity jsou jedním z největších problémů ve vývoji a také jasným pachem, který ukazuje na potřebu refaktoringu. Duplicity nejsou téměř nikdy v pořádku. Důvodem je to, že při případné změně je nutné upravit kód na více místech, což vede k vyššímu riziku chyb. Také to zpravidla znamená, že vývojář vynaložil zbytečné úsilí psaním kódu, který už v nějaké podobě někde existuje. Odstraněním duplicit se navíc kód značně zjednoduší a zpřehlední. Často je tomu tak proto, že dvě metody (či obecněji oblasti kódu) jsou si velmi podobné, ale ne stejné (mohou se lišit například jen v použitých konvencích). A právě sjednocením těchto rozdílů se vyjasní účel kódu, což výrazně napomůže dalšímu refaktoringu.

Osvědčilo se mi také vrátit se při zjištěné chybě k poslednímu funkčnímu stavu, pokud ovšem tato chyba nebyla jednoduše odstranitelná během několika desítek vteřin, a refaktorovat v menších krocích (viz jedna z praktik předkládané metodiky). Většinou stačilo využít možnosti vývojového prostředí vrátit zpět poslední provedené akce, ale dvakrát jsem využil také systému na správu verzí.

6.3 Zhodnocení průběhu

Hlavním cílem praktické části této práce bylo ukázat aplikaci navrhované metodiky v praxi. V této kapitole jsem se držel metodiky s omezeními danými tím, že jsem pracoval samostatně a nikoli v týmu. Dodržet jednotlivé části postupu navrhovaného metodikou nebylo obtížné. Musel jsem si ale zvyknout na některé praktiky, například na již výše zmiňované dělání jen jedné věci v jeden okamžik.

Co se mi osvědčilo, to popisuji v předcházející podkapitole. Zde znovu vyzdvihnu jen soupis odpovědností tříd a statickou analýzu kódu. Obojí bylo výborným ukazatelem na problémová místa a mohu obě tyto metody jen doporučit.

Během refaktoringu se objevovaly další a další oblasti, kterým bylo třeba věnovat pozornost. Není tedy třeba plánovat refaktoring do detailu hned na začátku. Je třeba pouze

identifikovat problémová místa, začít refaktorovat a v průběhu pružně reagovat na nové požadavky.

6.4 Shrnutí

V této kapitole jsem představil projekt Doučka.cz a předvedl praktickou aplikaci navrhované metodiky na reálném projektu, tedy při refaktoringu této aplikace. Zaměřil jsem se postupně na jednotlivé části algoritmu uvedeného v metodice. Okomentoval jsem samotný průběh refaktoringu a stručně se zmínil i o výsledcích statické analýzy kódu s tím, že podrobnější výsledky jsou k nalezení v příloze B. V závěru jsem pak uvedl i praktické tipy získané při práci, které mohou být inspirací dalším vývojářům.

7 Závěr

Hlavním cílem této práce bylo navrhnout a prakticky ověřit metodiku refaktoringu PHP kódu webových aplikací, která by byla prakticky využitelná a aplikovatelná i ve stávajících projektech. Za tímto účelem jsem nejprve prozkoumal existující literaturu o refaktoringu, jak obecně, tak i v PHP, byť zde byly zdroje velice omezené. Zaměřil jsem se také na metodiky vývoje software. Vzhledem k požadavku na praktickou použitelnost metodiky jsem se z velké části opíral o svou vlastní praxi ve vývoji webových aplikací v PHP.

Dále jsem uvedl pachy v kódu, které jsou běžně známé v oblasti refaktoringu, a přidal jsem k nim několik nejčastěji se vyskytujících v PHP.

Na základě výše uvedeného a analýzy požadavků na metodiku jsem definoval nejprve tři základní principy navrhované metodiky: minimalizace rizika chyb, maximalizace efektivity refaktoringu i vývoje a maximalizace praktické využitelnosti metodiky. Dále jsem definoval také předpoklady pro použití metodiky a role aktérů, kteří se přímo či nepřímo na refaktoringu podílejí.

Stěžejním prvkem metodiky je navržený postup při refaktoringu. Jeho jednotlivé části popisuji v práci podrobněji. Jedná se o identifikaci motivu refaktoringu a oblastí pro refaktoring, dále pak výběr jedné z problémových oblastí a samotný refaktoring. Pro efektivnější průběh refaktoringu při minimalizaci rizika chyb doporučuji dodržovat praktiky, které uvádím dále.

Také se zmiňuji o nástrojích vhodných pro refaktoring. Důležitá je také zmínka o přínosech a rizicích refaktoringu, jelikož dobrá informovanost a zkušenosti vývojářů v této oblasti jsou důležitým předpokladem úspěšného refaktoringu.

Na závěr teoretické části uvádím několik poznámek k zavedení metodiky v praxi.

V praktické části této práce ukazuji praktickou aplikaci metodiky na vlastním projektu Doučka.cz. Kromě zdůraznění postupu dle metodiky také popisuji postřehy a náměty získané v průběhu refaktoringu.

Tím jsem tedy splnil hlavní cíl této práce s pomocí splnění dílčích cílů, uvedených v úvodu.

Mým vlastním přínosem je zejména metodika samotná, jelikož refaktoring v PHP nebyl zatím touto formou sepsán. Zásadní charakteristikou této metodiky je její použitelnost v praxi a také její jednoduchost a snadná zvládnutelnost ze strany vývojářů. Zde jsem vycházel především ze své vlastní praxe.

K obecně známým pachům v kódu přidal několik dalších, které jsou specifické pro PHP, čímž jsem rozšířil průzkum problematiky refaktoringu PHP aplikací a do budoucna snad inspiroval další autory k hlubšímu zkoumání této oblasti.

Výslednou metodiku lze upravit či rozšířit i pro použití v jiných jazycích či na jiných typech projektů. Její jednotlivé části jsou sice propojeny, ale v případě potřeby je lze nezávisle upravovat, například přidávat nové praktiky či způsoby identifikace oblastí vhodných pro refaktoring.

Tato metodika mi byla velmi vhodným pomocníkem při refaktoringu projektu Doučka.cz. Doufám, že pomůže i dalším vývojářům v refaktoringu jejich projektů.

Terminologický slovník

Termín (zkratka), ustálený český překlad	**Význam**	**Zdroj**
Cascading style sheets (CSS), česky kaskádové styly	jazyk pro formátování (X)HTML a XML	http://cs.wikipedia.org/wiki/Kask%C3%A1dov%C3%A9_styly
code review, česky revize kódu	procházení již napsaného kódu za účelem jeho zkvalitnění	autor
code smell, česky pach v kódu	problémové oblasti v kódu vhodné pro refaktoring	[Fowler, 1999]
cron	pravidelně automaticky spouštěný skript	autor
Cyclomatic complexity number (CCN2)	softwarová metrika, jejíž hodnota označuje počet možných větvení v kódu	autor, založeno na http://pdepend.org/documentation/software-metrics/cyclomatic-complexity.html
design pattern, česky návrhový vzor	obecně použitelné řešení pro obvyklý problém v objektově orientovaném vývoji	autor
flash message	zpráva zobrazená uživateli, zpravidla o výsledku právě provedené operace	autor
front-end	prezentační vrstva aplikace	autor
HyperText Markup Language (HTML)	značkovací jazyk, který se používá pro webové stránky	autor
JavaScript	skriptovací jazyk, který se používá zejména pro programování front-endu	[Mittner, 2011]
log	záznam, typicky o aktuálním dění v aplikaci, ukládá se do souboru či databáze	autor
model-view-controller (MVC)	návrhový vzor oddělující prezentační vrstvu (view) od datové vrstvy (model), jejichž interakci a interakci aplikace s uživatelem řídí controller	autor
NPath	softwarová metrika, jejíž hodnota označuje počet možných acyklických průchodů metodou	autor, založeno na http://phpmd.org/rules/codesize.html#npathcomplexity

object-oriented programming (OOP), česky objektově orientované programování	způsob vývoje software, kdy se využívá tříd (objektů) a jejich vlastností	autor, založeno na http://en.wikipedia.org/wiki/Object-oriented_programming
PHP: Hypertext Preprocessor (PHP)	jazyk, který se používá zejména pro programování webových aplikací	autor
refaktoring	změny struktury kódu za účelem jeho kvalitativního zlepšení, přičemž se nemění jeho vnější chování	[Fowler, 1999]
Structured query language (SQL)	databázový dotazovací jazyk	autor
Uniform resource locator (URL)	jednoznačná adresa umístění webové stránky (či jiné entity) na internetu	autor
view script	skript v prezentační vrstvě MVC, zpravidla obsahuje HTML doplněné o PHP	autor

Literatura

BECK, Kent et al. *Agile manifesto* [online]. 2001 [cit. 2013-10-20].
Dostupné z: http://agilemanifesto.org/

BECK, Kent a Cynthia ANDRES. *Extreme programming explained: embrace change*. 2nd ed. Boston, MA: Addison-Wesley, 2005, xxii, 189 p. ISBN 03-212-7865-8.

BEŇO, Maroš. Real-World Refactoring of Dynamic Language Code. Praha, 2013. Diplomová práce. České vysoké učení technické v Praze.

BUCHALCEVOVÁ, Alena. *Metodiky vývoje a údržby informačních systémů: kategorizace, agilní metodiky, vzory pro návrh metodiky*. 1. vyd. Praha: Grada, 2005, 163 s. ISBN 80-247-1075-7.

BUCHALCEVOVÁ, Alena. *Metodiky budování informačních systémů*. Vyd. 1. Praha: Oeconomica, 2009, 205 s. Vysokoškolská učebnice (Oeconomica). ISBN 978-80-245-1540-3.

COHN, Mike. *Succeeding with agile: software development using Scrum*. Upper Saddle River, NJ: Addison-Wesley, c2010, xxviii, 475 p. Addison-Wesley signature series. ISBN 03-215-7936-4.

Doporučené postupy v programování. *Department of Distributed and Dependable Systems MFF UK* [online]. [2009] [cit. 2013-10-19].
Dostupné z: http://d3s.mff.cuni.cz/teaching/programming_practices/#outline

FEATHERS, Michael C. *Údržba kódu převzatých programů*. Vyd. 1. Brno: Computer Press, 2009, 367 s. ISBN 978-80-251-2127-6.

FEATHERS, Michael C. *Working effectively with legacy code*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, c2005, xxi, 434 p. ISBN 01-311-7705-2.

FOWLER, Martin. *Refactoring: improving the design of existing code*. Boston: Addison-Wesley, 1999, xxi, 431 s. ISBN 0-201-48567-2.

FOWLER, M. Refactoring - zlepšení existujícího kódu. Vyd. 1. Praha: Grada, 2003, 394 s. ISBN 80-247-0299-1.

FREEMAN, Eric a Elisabeth FREEMAN. *Head first design patterns*. 1st ed. Sebastopol: O´Reilly, 2004, xxxvi, 638 s. ISBN 05-960-0712-4.

GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995, 395 s. ISBN 02-016-3361-2.

HUJER, Martin. *Kontinuální integrace při vývoji webových aplikací v PHP*. Praha, 2012.
Dostupné z: http://isis.vse.cz/zp/103424. Bakalářská práce. Vysoká škola ekonomická v Praze.

KERIEVSKY, Joshua. *Refactoring to patterns*. Boston: Addison-Wesley, 2005, xxvi, 367 s. ISBN 03-212-1335-1.

KOTYZA, David. Agilní správa databáze. Praha, 2010.
Dostupné z: http://isis.vse.cz/zp/78026. Diplomová práce. Vysoká škola ekonomická v Praze.

KRUCHTEN, Philippe. *The rational unified process: an introduction*. 3rd ed. Upper Saddle River: Addison-Wesley, 2004, xviii, 310 s. ISBN 03-211-9770-4.

MASSONI, Tiago. *Introducing Refactoring to Heavyweight Software Processes* [online]. 2003 [cit. 2013-10-19]. Dostupné z: http://www.cin.ufpe.br/~tlm/download/article-refactoring-RUP.pdf

MCCONNELL, Steve. *Code complete*. 2nd ed. Washington: Microsoft Press, c2004, xxxvii, 914 s. ISBN 07-356-1967-0.

MITTNER, Jan. Metodika pro vývoj webových aplikací. Praha, 2011.
Dostupné z: http://isis.vse.cz/zp/97949. Diplomová práce. Vysoká škola ekonomická v Praze.

MUNROE, Alex. PHP: a fractal of bad design. Fuzzy notepad [online]. 2012 [cit. 2013-11-04].
Dostupné z: http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/

OpenUP [online]. 2012 [cit. 2013-10-19]. Dostupné z: http://epf.eclipse.org/wikis/openup

PAULAVETS, Anastasiya. Refaktoring databází. Praha, 2012.
Dostupné z: http://isis.vse.cz/zp/105719. Bakalářská práce. Vysoká škola ekonomická v Praze.

PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

RATIONAL SOFTWARE. *Rational Unified Process: Best Practices for Software Development Teams* [online]. 1998 [cit. 2013-10-19]. Dostupné z:
http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

SCHWABER, Ken a Jeff SUTHERLAND. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game* [online]. 2013 [cit. 2013-10-19]. Dostupné z:
https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf

SIMS, Chris a Hillary JOHNSON. *The elements of Scrum*. Foster City, CA: Dymaxicon, 2011. ISBN 978-098-2866-917.

TINKHAM, Cem KANER a MCGEE. Refactoring: Code smells. Center for Software Testing Education & Research [online]. 2005 [cit. 2013-11-02].
Dostupné z: http://www.testingeducation.org/pt/Refactoring-smells.pdf

TRUCCHIA, Francesco a Jacopo ROMEI. Pro PHP refactoring with test driven design. New York: Distributed to the book trade by Springer Science Business Media, LLC, c2010, xxi, 335 p. Expert's voice in open source. ISBN 14-302-2727-3.

Version One: 7th annual state of agile development survey. VERSION ONE: Agile made easier [online]. 2012 [cit. 2013-10-31]. Dostupné z: http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf

W3Techs: Usage of server-side programming languages for websites. W3TECHS: Web technology surveys[online]. 2013 [cit. 2013-10-31]. Dostupné z:
http://w3techs.com/technologies/overview/programming_language/all

Seznam obrázků a tabulek

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

A Příloha A: Odpovědnosti tříd

V této příloze uvádím seznam odpovědností různých tříd a zároveň třídu, do které jsem tuto odpovědnost při refaktoringu přesunul (pokud nikam, naznačuji to pomlčkami).

Níže uvedené *Tabulky A-1, A-2, A-3* a *A-4* obsahují pouze třídy, které měly před refaktoringem více než jednu odpovědnost, jelikož to je důležité pro praktickou ukázku v mezích této práce.

Všechny konkrétně jmenované třídy v pravém sloupci jsem nově vytvořil právě za účelem lepšího rozdělení odpovědností mezi třídami.

V ideálním případě by měla každá třída mít pouze jednu odpovědnost. Přestože jsou některé třídy uvedeny v pravém sloupci vícekrát (u více odpovědností), jsou si tyto odpovědnosti natolik podobné, že je lze pojmenovat souhrnným názvem. Záleží totiž na úrovni detailu (granularitě) pohledu. Při příliš detailním náhledu by vzniklo příliš mnoho malých, tzv. „líných“ tříd. Zvolil jsem tedy takovou úroveň, která byla dle mého názoru nejvhodnější.

A.1 Třídy modelu

A.1.1 FairDeals

Zde je třeba upřesnit, že tato třída má své jméno z českého označení „férovky“, kterým jsme pojmenovali případy, kdy si lektor převezme doučování, ovšem zprostředkovací poplatek zaplatí dodatečně, čímž mu u nás na Doučka.cz vzniká dluh. „Férovka“ má symbolizovat, že jsme féroví jak my, tak lektor.

Tabulka A-1: Odpovědnosti třídy FairDeals před refaktoringem (zdroj: autor)

Odpovědnost	**Pokud je v této třídě nevhodná, do které třídy ji přesunout?**
získává data o férovkách z databáze	---
posílá varování neplatičům (včetně skládání zprávy)	Mailer
splácí férovky uživatele (pokud to kredit dovolí)	Credit
označí lektory jako nespolehlivé (dlouho nesplatili férovku, nebudou si už nikdy moci vzít další)	UsersUpdater
zjistí, zda je doučování s daným ID nesplacená férovka	OrderDetail
zjistí celkový dluh uživatele za férovky	UserDetail

A.1.2 Orders

Tabulka A-2: Odpovědnosti třídy Orders před refaktoringem (zdroj: autor)

Odpovědnost	**Pokud je zde nevhodná, do které třídy ji přesunout?**
získává data z databáze o objednávkách	---
skládá SQL pro dotazy na databázi	OrdersSql
získává data z databáze o konkrétní objednávce	OrderDetail
vkládá do databáze nové objednávky	OrderInserter
aktualizuje volné i převzaté doučování (včetně převzetí objednávky lektorem)	OrderUpdater
převádí profilovou objednávku na neadresnou	OrderUpdater
aktualizuje více doučování najednou	OrdersUpdater
posílá e-maily o zlevnění doučování, včetně tvoření zprávy	Mailer, MailMessageComposer
skládá a odesílá e-mail o nové objednávce	MailMessageComposer

A.1.3 Users

Tabulka A-3: Odpovědnosti třídy Users před refaktoringem (zdroj: autor)

Odpovědnost	**Pokud je zde nevhodná, do které třídy ji přesunout?**
získává data z databáze o lektorech	---
získává data o konkrétním lektorovi	UserDetail
přidává nové lektory	UserInserter
navýší a odečte lektorovi kredit	Credit
aktualizuje data o lektorovi	UserUpdater
maže lektory	UserUpdater

A.1.4 Controller (obecně)

Tabulka A-4: Odpovědnosti tříd controllerů před refaktoringem (zdroj: autor)

Odpovědnost	**Pokud je zde nevhodná, do které třídy ji přesunout?**
volá metody tříd modelu	---
inicializuje formuláře	---
ošetřuje proměnné na vstupu (předané z formulářů nebo parametrem v URL)	---
upravuje data z formulářů, dodatečně je validuje	do tříd příslušných formulářů
přiřazuje data z modelu to proměnných view skriptů	---
skládá a odesílá maily	Mailer, MailMessageComposer
loguje	do tříd modelu, kde se logované operace provádějí
zadává flash zprávy pro zobrazení	---
skládá text flash zpráv	FlashMessageComposer
přesměrovává	---

B Příloha B: Výsledky statické analýzy kódu

B.1 Npath

V *Tabulce B-1* jsou uvedeny metody s nejvyššími hodnotami metriky NPath, konkrétně ty, které měly tuto hodnoty před refaktoringem větší než 500. Tabulka je řazena sestupně podle této hodnoty před refaktoringem.

Křížky znamenají, že metoda byla zcela odstraněna. Snížení hodnot jsem dosáhl částečně úpravami přímo v dané metodě, částečně extrakcí do nových metod. Při této extrakci jsem ale nevytvořil žádné metody s hodnotou metriky Npath větší než 200 nebo CCN2 větší než 10 (nejednalo se tedy o přesunutí problému jinam).

Tabulka B-1: Metody s hodnotou NPath před refaktoringem vyšší než 500 (zdroj: autor)

		PŘED				PO			
soubor třídy	**název metody**	**ccn**	**ccn2**	**ncloc**	**npath**	**ccn**	**ccn2**	**ncloc**	**npath**
lector/IndexController	myDataAction	26	27	177	5849088	8	8	59	128
default/IndexController	doLoginAction	17	22	94	765000	5	6	40	24
AssignedOrdersController	cancel	12	19	87	14400	7	8	48	160
lector/IndexController	doTakeOrder	13	16	69	10800	8	9	42	72
Orders	changeParam	13	18	66	7680	6	8	40	72
lector/IndexController	uploadFiles	14	18	73	3584	3	5	13	9
LectorsController	indexAction	6	6	33	3125	1	1	30	1
AssignedOrdersController	indexAction	6	6	32	3125	1	1	27	1
DeletedUsersController	indexAction	6	6	33	3125	1	1	30	1
default/IndexController	newPassword2Action	7	10	30	2500	4	7	29	20
AssignedOrdersController	paramchangeAction	8	10	38	2500	4	4	24	5
AssignedOrdersController	detailAction	12	16	120	880	10	10	82	48
Orders	assignOrder	9	10	59	802	6	7	30	32
FinanceController	indexAction	8	8	35	750	x	x	x	x
Users	getSqlForGet	12	12	59	576	2	2	23	2
lector/IndexController	returnCreditAction	10	10	57	576	8	8	43	144

B.2 CCN2

Zde platí stejná poznámka jako k tabulce pro NPath. Do *Tabulky B-2* byly zahrnuty metody s hodnotou CCN2 větší než 10. Řazena je sestupně podle této hodnoty před refaktoringem. Některé metody jsou samozřejmě uvedeny i v předchozí tabulce, jelikož měly vysoké hodnoty obou metrik.

Tabulka B-2: Metody s hodnotou CCN2 před refaktoringem vyšší než 10 (zdroj: autor)

		PŘED				PO			
soubor třídy	**název metody**	**ccn**	**ccn2**	**ncloc**	**npath**	**ccn**	**ccn2**	**ncloc**	**npath**
lector/IndexController	myDataAction	26	27	177	5849088	8	8	59	128
default/IndexController	doLoginAction	17	22	94	765000	5	6	40	24
AssignedOrdersController	cancel	12	19	87	14400	7	8	48	160
Orders	changeParam	13	18	66	7680	6	8	40	72
lector/IndexController	uploadFiles	14	18	73	3584	3	5	13	9
AssignedOrdersController	detailAction	12	16	120	880	10	10	82	48
lector/IndexController	doTakeOrder	13	16	69	10800	8	9	42	72
Credit	getMessagesForCharge	14	15	84	360	x	x	x	x
Subjects	orderSubjectsToString	10	14	30	150	4	6	18	10
Orders	getOrdersFromDb	10	13	45	252	6	8	30	30
Orders	getOrdersFromDbSql	11	13	66	144	1	1	27	1
Users	getSqlForGet	12	12	59	576	2	2	23	2
Credit	chargeCreditFromMails	12	12	64	264	3	3	21	10
NewOrdersController	searchajaxAction	12	12	73	52	3	3	30	4
AssignedOrdersController	setUpFlashMessage	11	12	45	281	x	x	x	x
Subjects	getLevelName	11	11	24	100	11	11	25	100
Orders	getMinExpectedEarning	11	11	33	25	1	1	7	1
Orders	getOrderData	6	11	50	192	4	6	42	16
lector/IndexController	profileOrderDetail2Action	10	11	59	336	7	8	25	96
lector/IndexController	profileTopAction	9	11	42	75	7	9	41	27