REINGENEERING A MODERNIZACE SOFTWAROVÝCH SYSTÉMŮ

Jan Pavlovič, Lukáš Kokrment, Marek Cikánek
Fakulta informatiky, Masarykova univerzita, {xpavlov, xkokrmen, xcikanek}@fi.muni.cz

ABSTRAKT:
V souvislosti se vzrůstajícím počtem stárnoucích softwarových systémů vzrůstá i potřeba na jejich úpravu a modernizaci. Hlavním cílem reengineeringu je modernizace aplikace při zachování maximální možné funkcionality stávajícího systému. Článek představí základní postupy a metodiky, jak řešit tuto komplikovanou problematiku.

KLÍČOVÁ SLOVA:
Reengineering, modernizace, softwarové systémy

ÚVOD

Pro dnešní společnost je typický především nezadržitelný technologický rozvoj. Tento rozvoj se samozřejmě dotýká i oblasti informačních technologií. Moderní doba si vynucuje neustálé obměňování a modernizaci existujících softwarových systémů, což představuje mnohdy nemalý problém pro jejich provozovatele i uživatele. Provozovatelé těchto systémů jsou dříve nebo později postaveni před poměrně zásadní rozhodnutí, zda existující systém zmodernizovat nebo vytvořit zcela znovu (a akceptovat s tím souvisejících rizika).

Disciplína, která se těmito otázkami souhrnně zabývá, se nazývá software reengineering (softwarové reinženýrství) [3], [13]. Vzhledem k obecnosti problematiky reengineeringu je velmi těžké podat dostatečně obecný, ucelený koncept, který by bylo možné využít na širší spektrum konkrétních případů. Je to především z důvodu rozdílnosti jednotlivých systémů, rozdílných požadavků na tyto systémy, rozdílnému aplikačnímu zaměření a také rozdílnosti prostředí, ve kterých tyto systémy pracují. Můžeme se však pokusit popsat určité společné techniky a metodiky, které nám celý tento proces do značné míry zpřehledňují a zjednodušují.

Vývoj rozsáhlých softwarových systémů není snadná záležitost. Při tomto procesu musíme brát v úvahu celou řadu faktorů nutných k úspěšné realizaci projektu, kterými jsou zejména:

- Technické aspekty zahrnující počítačovou infrastrukturu a softwarové vybavení.
- Netechnické aspekty dané strukturou organizace vyvíjející daný systém a jejími ekonomickými možnostmi.
- Znalosti z oblasti specifikace požadavků na softwarový systém, jeho analýzy, návrhu, implementace, testování a také instalace u zákazníka.
- Lidské zdroje schopné aplikovat výše uvedené znalosti při vývoji systému.
- Řízení spjaté s vývojem samotného systému umožňující efektivní využití všech výše uvedených faktorů.

Náročnost vývoje způsobuje, že v současné době klesá počet rozsáhlých systémů, které jsou vyvíjeny zcela od počátku. Oproti tomu zároveň pozvolně narůstá počet *legacy* (zděděných) systémů, které nadále zůstávají v provozu [9].

Provozovatelé *legacy* systémů se jich nechtějí zříci převážně z následujících důvodů:

- *Legacy* systémy bývají většinou velmi rozsáhlé. Původní vývojáři systému většinou nebývají již k dispozici a tak neexistuje nikdo, kdo by celý systém důkladně znal. Původní specifikace požadavků na systém také bývá často po letech vývoje a používání nenávratně ztracena. Většina dokumentace bývá zastaralá nebo

nekonzistentní a jediným zdrojem informací se proto stává současný kód systému. Kompletní přepsání systému je tedy velmi obtížné.

- Zděděné systémy představují pro provozovatele zdroj příjmů, nové projekty představují riziko neúspěchu.
- Zákazník je ochoten zaplatit pouze za novou funkcionalitu a nikoliv za vývoj nového systému. Obecně také platí, že je-li zákazník spokojen se stávající verzí systému, nemá jej potřebu zásadním způsobem měnit.
- Softwarové firmy nemají dostatek času ani vývojářů, aby mohly uskutečnit renovaci úspěšného systému. To platí i v případě, že současný stav systému zabraňuje jeho dalšímu vývoji.

Praktický důsledek je, že tyto systémy zůstávají velmi dlouho v provozu. Typickým příkladem může být řízení letového provozu v USA, kde se používají běžně softwarové a hardwarové systémy staré více než 20 let a nikdo se je neodvažuje vyměnit za nové.

CÍLE REINGENEERINGU

Symptomy degradace systému vedoucí k rozhodnutí pro reingeneering (případně pro vytvoření a nasazení zcela nového systému) představují zejména časté chyby (výpadky) v systému, velmi složitá struktura programu a logických toků, vysoké nároky na systémové prostředky nebo na údržbu systému, pevně kódované parametry, nedostatečná dokumentace, chyby ve specifikaci návrhu a další.

Nahrazení původního systému zcela novým nelze velmi často z mnoha důvodů jednoduše realizovat – riziko spojené s nahrazením osvědčeného systému (s mnoha sice nepříjemnými, ale známými nedostatky) novým a neověřeným bývá totiž příliš velké. Obecně tedy vzrůstá snaha o „recyklování“ existujícího SW, o jeho úpravu do strukturované, lépe udržovatelné podoby a přichází na řadu reengineering. Mezi hlavní cíle a přínosy softwarového reengineeringu patří obecně:

- Snížení rizika – při klasickém softwarovém vývoji existuje vysoká míra rizika. Mohou se vyskytnout problémy se samotným vývojem systému, personální problémy nebo problémy při specifikaci požadavků.
- Snížení nákladů – náklady na reengineering bývají většinou významně nižší než náklady na vývoj nového systému.

Z hlediska provozu a údržby systému pak mezi cíle reingeneeringu patří zejména zjednodušení údržby, zvýšení výkonnosti, zvýšení spolehlivosti a další konkrétní cíle v závislosti na požadavcích a záměrech provozovatele (přechod na jinou platformu, použití nových technologií, změny GUI, apod.)

PRINCIPY SOFTWAROVÉHO REINGENEERINGU

Abychom si mohli proces reengineeringu popsat trochu podrobněji, ukážeme si nejdříve jaké abstraktní úrovně lze nalézt v klasickém modelu softwarového vývoje (obrázek 1). Každá abstraktní úroveň odpovídá určité fázi vývoje a zároveň definuje systém na určité úrovni detailu.

- *Konceptuální úrove*ň představuje nejvyšší úroveň abstrakce. Je zde popsán koncept systému – tzn. důvod pro jeho existenci. Na této úrovni jsou funkční charakteristiky systému popsány pouze velmi obecně.
- *Na úrovni požadavk*ů jsou již funkční charakteristiky systému popsány detailně. Na těchto prvních dvou úrovních však ještě nejsou zmíněny vnitřní detaily systému.
- *Úrove*ň *návrhu* popisuje charakteristiky jako například architektonický návrh systému, systémové komponenty, rozhraní mezi komponentami, algoritmy a datové struktury.

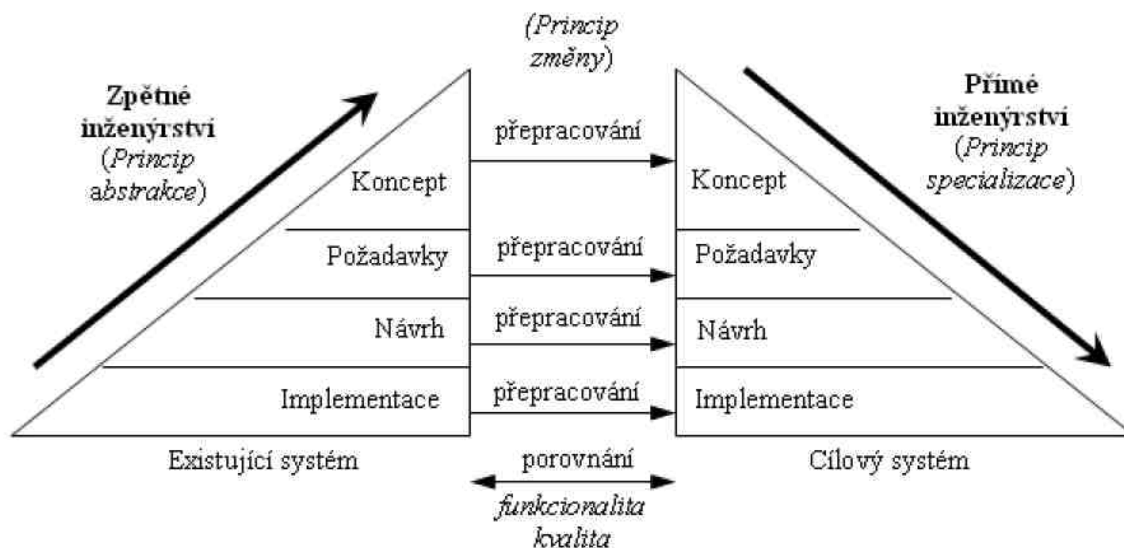
- Nejnižší úroveň je *úroveň implementační. Na* této úrovni se popis systému zaměřuje na implementaci daných charakteristik, přičemž zápis je realizován ve vybraném programovacím jazyce.

Obrázek 1: Abstraktní úrovně softwarového vývoje

Na obrázku 2 pak můžeme vidět, jak se proces reengineeringu dotýká jednotlivých abstraktních úrovní. Na obrázku je možné vidět tři základní principy procesu reengineeringu:

- *Princip abstrakce* představuje postupné zvyšování úrovně abstrakce systému. Reprezentace systému je vytvářena postupnou výměnou existujících detailních informací více abstraktními. Princip vytváří formu reprezentace systému, která zvýrazňuje vybrané charakteristiky systému pomocí skrývání informací o charakteristikách jiných. Tento postup směrem „zdola-nahoru" se nazývá *zpětné inženýrství a je s* ním asociována řada pomocných podprocesů, nástrojů a technik.
- *Princip změny* znamená provedení jedné nebo více změn do reprezentace systému při zachování stejné úrovně abstrakce zahrnující přidání, odebrání a změnu informace, ale ne funkcionality.
- *Princip specializace* představuje postupné snižování úrovně abstrakce v reprezentaci systému, která je způsobená výměnou existující informace v systému nějakou detailnější informací. Tento proces se nazývá přímé inženýrství a zahrnuje psaní nového kódu a postupné zpřesňování některých procesů souvisejících s vývojem.

Pro změnu charakteristiky systému je potřeba provést zásah na odpovídající úrovni abstrakce (tzn. na té, kde je explicitně zmíněna informace o této dané charakteristice). Pro překlad existujícího programového kódu do jiného programovacího jazyka není potřeba využívat zpětného reengineeringu, protože změna probíhá na implementační úrovni. S tím, jak se zvyšuje úroveň abstrakce, se mění povaha pozměňovacích zásahů do systému a zároveň se mění počet a povaha zásahů využívajících zpětného inženýrství. Pro obnovení specifikace požadavků na systém musejí být využity techniky zpětného inženýrství na úrovni návrhu a programového kódu, které umožní získat funkční charakteristiky systému.



Obrázek 2: Obecné schéma procesu reingeneeringu

ZÁKLADNÍ TECHNIKY

V této kapitole si uvedeme a vysvětlíme některé základní techniky, které lze použít v průběhu realizace reengineeringu určitého softwarového systému.

Zpětné inženýrství

Hlavní cíle zpětného inženýrství představují zejména identifikace komponent systému a jejich vzájemných vazeb, vytvoření reprezentace systému v jiné formě (na vyšší úrovni abstrakce), inspekce systému a obnova dokumentace [6]. Zpětné inženýrství se tedy zabývá pouze inspekcí systému, nikoliv jeho změnami a restrukturalizací. Dvě kvalitativně odlišné reprezentace systému vytvořené zpětným inženýrstvím jsou obnovení návrhu *(design recovery)* a obnovení dokumentace *(redocumentation).* Obnovení dokumentace má za úkol odvodit sémanticky ekvivalentní popis na stejné úrovni abstrakce. Obnovení návrhu odvozuje sémanticky ekvivalentní popis na vyšší úrovni abstrakce.

Zpětné inženýrství se většinou neomezuje pouze na obnovení dokumentace. Často je nutné se zajímat o to, proč jsou určité záležitosti řešeny určitým způsobem, jaký je význam zkoumaného fragmentu kódu a podobně. V případě přímého inženýrství je obvyklý postup od vyšších úrovní abstrakce k nižším úrovním implementací. Během tohoto procesu často dochází ke ztrátě informace (znalost detailu neznamená automaticky hlubší poznání systému). Při implementaci se téměř vždy ztratí informace, proč bylo zvoleno právě toto řešení detailu a jaký je význam detailu v celku. Jestliže chceme projít opačným směrem k vyšší hladině abstrakce, musíme ztracenou informaci rekonstruovat. Východiskem pro rekonstrukci je především poslední zachovalý zdrojový kód programu a stávající dokumentace.

Proces pochopení systému při zpětném inženýrství lze podpořit také specializovanými softwarovými nástroji.

Restrukturalizace

Restrukturalizaci lze popsat jako činnost, která se snaží o zlepšení srozumitelnosti existujícího kódu a usnadnění jeho případné změny. Funkce systému se při restrukturalizaci nemění. Na implementační úrovni většinou restrukturalizace představuje změnu struktury kódu beze změny jeho sémantiky. Za restrukturalizaci je považováno i nové naprogramování modulu poté, co byl revidován návrh.

Restrukturalizaci lze často realizovat s použitím automatizovaných nástrojů. Tyto nástroje mají velkou cenu i v jednoduché podobě, kterou představují například programy usnadňující vnímání programu (reformátovací programy).

S restrukturalizací se v posledních letech setkáme jako s činností, jejímž cílem je konverze existujícího softwaru do modulární podoby s vysokou mírou znovupoužitelnosti jednotlivých modulů. Náklady vložené do této činnosti se nevrátí pouze v úsporách při údržbě, ale i při tvorbě nových systémů. Restrukturalizace sama o sobě však nedokáže transformovat špatně navržený systém na dobrý. Většinu změn, jako například vylepšení systému a odstranění chyb, je stále nutné řešit manuálně. Více informací o restrukturalizaci lze nalézt v [14].

Přímé inženýrství

Pod tímto pojmem si můžeme představit běžný proces vývoje systému v oblasti softwarového inženýrství. To znamená postupný přesun od shromažďování požadavků na systém, přes vysokoúrovňový návrh systému, k postupně nižším úrovním návrhu systému a implementaci. Přídavné jméno „přímé" před slovem inženýrství se může zdát možná zbytečné, nicméně je potřebné z hlediska snadného odlišení přímého inženýrství od zpětného inženýrství.

Refaktorizace

Refaktorizace představuje formu restrukturalizace na úrovni objektově orientovaného návrhu. Cílem refaktorizace je zvýšení obecné použitelnosti jednotlivých tříd a jasná definice třídních rozhraní. Princip refaktorizace spočívá ve skládání relativně jednoduchých transformací (přejmenování třídy, přejmenování metody, přesunu metody nebo atributu do jiné třídy, ...) do podoby komplexnějších, sémantiku zachovávajících, transformací (vytvoření komponent z částí třídy, zavedení návrhového vzoru most).

MOŽNÉ ZPŮSOBY REALIZACE REINGENEERINGU

V okamžiku, kdy přibližně víme, do jaké podoby chceme existující systém přepracovat, jsme postaveni před problém volby vhodného způsobu, jak to provést. Existují tři základní možné přístupy k realizaci reengineeringu. Jednotlivé přístupy se liší zejména velikostí a frekvencí úprav a každý přístup má své výhody i nevýhody.

Metoda velkého třesku

Při přístupu označovaném jako „velký třesk" je celý cílový systém nahrazen novým, který je výsledkem reingeneeringu, v jediném kroku.. Tento přístup je často využíván v případě projektů, které vyžadují vyřešení nějakého zásadnějšího problému, jakým může být například migrace na jinou systémovou architekturu.

Hlavní výhoda tohoto přístupu spočívá právě v tom, že systém je přenesen do jiného prostředí v jediném okamžiku. Nemusejí tedy být vyvíjena žádná rozhraní pro spolupráci starých a nových komponent a současně nemusejí být udržována smíšená prostředí pro běh starých a nových součástí.

Nevýhodou tohoto přístupu je, že výsledkem bývají monolitické systémy, které nemusejí být vždy vyhovující. V případě velkých systémů může být tento přístup příliš náročný z hlediska množství času a spotřebovaných prostředků. Riziko spojené s tímto přístupem je vysoké. Funkcionalita původního systému musí zůstat zachována, proto musí být nový systém schopný pracovat souběžně se starým systémem. Souběžná činnost obou systémů může být obtížně realizovatelná a nákladná. Hlavní problém pak představuje správa změn, která je také důsledkem použití pouze jediného kroku. Je velmi pravděpodobné, že v době od zahájení vývoje nového systému po jeho dokončení, budou ve starém systému prováděny změny, které musí být následně přeneseny do nového systému.

Inkrementální přístup
Tento přístup lze jinými slovy popsat také jako „postupnou výměnu“ systému. Jednotlivé části systému přepisovány a přidávány inkrementálně spolu s tím, jak vznikají nové verze systému pro uspokojení nových požadavků na systém. Proces reengineeringu rozděluje systém do nových částí v závislosti na existujících částech systému.

Výhodou tohoto přístupu je rychlejší vývoj systémových komponent a snadnější odhalení chyb z důvodu snadné identifikace nových komponent. Díky tomu, že jsou průběžně vyvíjeny nové verze systému, může zákazník snadno sledovat postupný vývoj a rychle odhalit případnou ztracenou funkcionalitu. Nepochybným přínosem je také izolace přepisovaných komponent od zbytku systému. Se změnou starého systému se lze tedy daleko lépe vypořádat. Je to z důvodu, že změny v komponentách, kterých nejsou právě přepisovány, se žádným způsobem nedotknou právě přepisovaných komponent.

Nevýhoda inkrementálního přístupu je zejména v tom, že vývoj systému trvá déle a během vývoje je potřeba uvolňovat více průběžných verzí, což vyžaduje náročnější správu konfigurací. Další nevýhodou je, že celková struktura systému nemůže být změněna. Změněna může být pouze struktura specifických částí komponenty, která je právě přepisována. Je potřebná důkladná identifikace jednotlivých komponent existujícího systému a pečlivé plánování nové struktury cílového systému. Tento přístup představuje menší riziko než metoda velkého třesku, protože s tím, jak začne práce na konkrétní komponentě systému, lze nebezpečí s tím spojené snadno identifikovat a monitorovat.

Evoluční přístup
Při „evolučním“ přístupu jsou, stejně jako v případě přístupu inkrementálního, části původního systému postupně nahrazovány nově přepracovanými částmi systému. V případě tohoto přístupu je ovšem výběr komponent prováděn s ohledem na jejich funkcionalitu, ne na strukturu existujícího systému. Cílový systém je vytvářen postupným přepisováním komponent systému v pořadí daném jejich funkcionální provázaností. Evoluční přístup umožňuje vývojářům zaměřit úsilí spojené s reengineeringem na identifikaci funkčních objektů bez ohledu na to, kde se tyto objekty fyzicky vyskytují v původním systému. Komponenty původního systému jsou nejdříve rozděleny na jednotlivé metody a ty jsou následně přesunuty do nových komponent.

Výhodami evolučního přístupu jsou modulární návrh a užší funkční zaměření jednotlivých komponent. Tento přístup lze dobře využít při konverzi systému do objektově orientované podoby.

Nevýhodou je, že podobné funkce musí být identifikovány v rámci celého existujícího systému a zahrnuty do stejné funkční jednotky. Mohou také vyvstat problémy s definicí rozhraní a degradací doby odezvy systému způsobené tím, že tento přístup upřednostňuje během procesu reengineeringu funkcionální hledisko před hlediskem architektonickým.

VYUŽITÍ SOFTWAROVÝCH METRIK PŘI REINGENEERINGU
Softwarové metriky mapují určité vlastnosti softwarového projektu na číselné (nebo jiné) vyjádření pomocí striktně definovaných, objektivních pravidel. Výsledky měření jsou poté použity k popisu, posouzení a nebo předpovědi charakteristik daného softwarového projektu. Měření jsou většinou prováděna za účelem získání základních informací, díky kterým mohou být lépe naplánovány a realizovány jednotlivé úkoly doprovázející vývoj systému.

Softwarové metriky podporují mnoho činností v procesu reengineeringu tím, že pomáhají odhalit ty části systému, kterých se bude proces úpravy pravděpodobně týkat. Pomáhají při formování a prvotním pochopení zděděného systému a mohou odhalit indicie vedoucí k odhalení návrhových vad, které představují překážku při změně a rozšiřování systému. Výpočet metrik lze provádět automatizovaně použitím vhodných nástrojů. Místo kompletního

procházení zdrojového kódu tak můžeme důkladněji prostudovat pro nás zajímavá místa indikovaná pomocí výsledků softwarových metrik. Softwarové metriky můžeme rozdělit do několika skupin.

Metriky pro odhad složitosti měří složitost (komplexitu) vybrané entity v systému. Dále prezentované metriky měří složitost tříd. Pojmem složitost části softwarového produktu nebo zdrojového kódu se většinou snažíme popsat, kolik úsilí stráví vývojář pochopením, napsáním nebo modifikací dané části produktu - kód, který je těžko čitelný, je pokládán za složitý. Protože přímé měření složitosti není možné (protože bychom museli nechat vývojáře číst zdrojový kód a měřit kolik času potřeboval pro jeho pochopení), používáme pro odhad složitosti softwarové metriky. Mezi metriky pro odhad složitosti patří např. počet řádků kódu [18], vážený součet metod [5], [7], [12] a počet metod [15].

Metriky zkoumající vazby mezi třídami zkoumají množství a charakter vazeb mezi jednotlivými třídami definovanými v kódu. Třída je svázána s jinou třídou, jestliže je na jiné třídě nějakým způsobem závislá, což nastává například při přístupu k proměnným nebo při volání metod dané třídy. Příkladem metrik spadající do této kategorie jsou např. abstraktní vazby mezi daty [19], [16], [17] nebo velikost výstupní množiny [5].

Třetí skupinou softwarových metrik jsou metriky zkoumající třídní kohezi. Koheze v rámci třídy definuje, do jaké míry spolu souvisejí entity (jako například atributy a metody) dané třídy. Koheze bývá často měřena pomocí zkoumání vztahů mezi metodami třídy, kde tyto metody přistupují ke stejným instančním proměnným. Užitečná metrika zkoumající tuto vlastnost je např. těsnost třídní koheze [2], [16], [17].

Metriky zkoumající třídní hierarchii jsou čtvrtou kategorií. Základním společným znakem pro objektově orientované systémy je dědičnost. Pomocí dědičnosti mohou být modelovány vztahy mezi objekty (jako například vazba *is-a).* Dědičnost bývá také často využívána jako nástroj pro znovupoužití již existujících tříd. Z těchto faktů vyplývá, že použití metrik pro měření vlastností stromu dědičnosti systému často přináší zajímavé výsledky. Jednoduché metriky související se stromem dědičnosti a s jeho strukturou jsou hloubka stromu dědičnosti [5], počet přímých potomků třídy [5] a počet odvozených tříd [23].

PŘÍPADOVÁ STUDIE

Jako případovou studii, na které byly demonstrovány postupy a procesy popsané v tomto článku, můžeme uvést projekt Kalkulátoru energetické náročnosti technologií pro čištění kontaminovaných médií (*Waste Site Energy Calculator*).

Projekt byl zahájen přibližně na počátku roku 2004 na základě dohody mezi Fakultou informatiky Masarykovy univerzity a americkými organizacemi US EPA a PPRC, které se zabývají ochranou životního prostředí. Účelem projektu je vytvořit aplikaci pro provozovatele čistících areálů určených k odstraňování různých druhů kontaminantů (pohonné hmoty, těžké kovy, těkavé látky, kyanidy,. . .) z různých médií (půda, vzduch, voda,. . .). Systém umožňuje na základě vstupů zadaných uživatelem stanovit energetickou náročnost rozličných technologií pro čištění těchto kontaminovaných medií (tzv. sanačních technologií) a v konečném důsledku pak umožnit vybrat tu technologii, která nejméně zatěžuje životní prostředí.

Technicky je celý systém realizován jako webová aplikace na platformě Java 2 Enterprise Edition. Jako Servlet kontejner je použit Jakarta Tomcat ve verzi 5.5. Definice jednotlivých technologií jsou uvedeny v XML souborech. Ke každé technologii zároveň existuje javová třída obsahující implementaci výpočetního vzorce pro danou technologii.

Cílem bylo jednak implementovat různé pozměňovací požadavky, které přicházely z americké strany a dále také pokusit se restrukturalizovat jádro systému a integrovat do projektu některý z běžně používaných aplikačních rámců pro tvorbu webových aplikací. Hlavními požadavky se tedy postupně staly:

- Přepracování návrhu systému do přehlednější podoby.
- Odstranění původního proprietárního webového rámce a nahrazení nějakým univerzálním webovým rámcem.
- Implementace pozměňovacích návrhů přicházejících během vývoje systému.
- Příprava na možnou budoucí separaci jednotlivých technologií do podoby samostatných komponent.

Detekce problémů
V úvodním kroku celého procesu reingeneeringu bylo nutné stanovit kritická místa v původní implementaci systému:

- Nedokonalé rozdělení do vrstev (zejména velké množství kódu v JSP stránkách).
- Chyby v objektovém návrhu (rozsáhlá centrální třída v systému připomínající staré monolitické programy, nevýstižné pojmenování určitých tříd, silná závislost mezi jednotlivými balíky, příbuzné třídy v různých balících).
- Proprietární webový rámec (přílišná integrace se zbytkem systému, nedeterministické chování v určitých situacích, těžkopádnost, neprůhlednost návrhu a obtížné pochopení nezaškoleným vývojářem, nesnadná rozšiřitelnost)
- Defekty na úrovni kódu (nesprávné pojmenování některých proměnných nedodržující konvence jazyka Java, „magické hodnoty“ v kódu, nepružný mechanismus indikace chyb, existence skrytých závislostí v kódu)

Analýza problémů
V tomto kroku bylo nutné analyzovat kritická místa odhalená v předchozím kroku a stanovit možné způsoby jejich odstranění:

Nedokonalé rozdělení do vrstev – výskyt velkého množství kódu v JSP stránkách snižuje přehlednost a udržovatelnost kódu. Kód by se měl v JSP stránkách vyskytovat co možná nejméně. Odstranění nadbytečného kódu z JSP stránek v původní aplikaci bylo možné dosáhnout následujícími úpravami:

- Převedením aplikační logiky z JSP stránek do samostatných tříd
- Náhradou některých často používaných programových konstrukcí (iterace přes prvky kolekce, ...) vyskytujících se v JSP stránkách značkami poskytovaných rámcem Struts.

Chyby v objektovém návrhu – chyby v objektovém návrhu představují jedny z nejvážnějších chyb, které lze v systému pozorovat. Tyto chyby mají většinou dalekosáhlé následky. V případě tohoto projektu je zde výskyt chyb tohoto rázu podmíněn především nedostatečnou původní analýzou a také nedostatečnou původní specifikací. Další entropie byla do systému zanášena postupným přidáváním funkcionality do již existujících tříd. Tento stav bylo možno řešit pouze velmi obtížně, zejména z důvodu, že systém musel být v průběhu přeměny stále v provozuschopném stavu. Změny v daném okamžiku musely být tedy pouze lokální pro určitou část systému. Kompletní přepsání systému by pravděpodobně vyžadovalo menší úsilí, nebylo však možné z důvodu, který je typický pro většinu zděděných systémů, tj. neexistence původní specifikace a funkcionality ukryté v systému (např. způsob výběru základní technologie pro čistící proces – *baseline technology*). Jako nejlogičtější přístup pro opravu původního návrhu byl tedy zvolen přístup založený na následujících úpravách:

- Dekompozicí rozsáhlých klíčových tříd na menší třídy
- Extrahováním nepatřičné funkcionality z tříd a jejím umístěním do samostatných tříd
- Striktnějším oddělením jednotlivých částí systému
- Reorganizaci struktury balíčků (jmenných prostorů pro jednotlivé třídy)

Proprietární webový rámec – účelem této části kódu bylo především zabezpečovat udržování hodnot jednotlivých prvků formulářů na stránkách jednotlivých technologií a dále

realizovat jejich následné automatické ukládání do proměnných tříd odpovídajících technologií. Jak bylo již uvedeno výše, byl tento mechanismus realizován poměrně těžkopádně a zbytečně komplikovaně. Původní řešení zejména způsobovalo problémy s integrací některých pozměňovacích návrhů přicházejících během vývoje systému. V některých případech integraci dokonce úplně zabraňovalo. Odstranění tohoto problému však bylo možné realizovat poměrně přímočarým řešením, a to nahrazením původního webového rámce nějakým univerzálním webovým rámcem. Vzhledem k povaze projektu padla nakonec volba na rámec Jakarta Struts, který nejlépe vyhovoval podmínkám stávající infrastruktury projektu. Z integrace webového rámce Jakarta Struts plynou značné výhody. Rámec Strus představuje robustní základ pro webovou aplikaci, poskytuje širokou škálu funkcionality, odstínění od některých problémů spojených s webovým vývojem a umožňuje striktnější zaměření se na vývoj aplikační logiky.

Defekty na úrovni kódu – problémy této kategorie snižují čitelnost a přehlednost kódu a znesnadňují odhalování chyb. Jejich odstranění značně usnadňuje budoucí rozšiřování systému. Výše uváděné problémy:

- „Magické hodnoty“ v kódu – tento termín označuje výskyt konkrétních řetězců nebo číselných hodnot přímo uvnitř zdrojového kódu. Tento problém lze řešit zavedením konstant a náhradou exaktních hodnot těmito symbolickými konstantami.
- Nepružný mechanismus indikace chyb – původně byla indikace chyb řešena pomocí návratových hodnot metod. Tento zastaralý a poměrně těžkopádný přístup vyžaduje explicitní kontrolu návratových hodnot a v případě výskytu chyby bývá poměrně obtížné vypátrat její pravou příčinu. Tento přístup byl nahrazen flexibilnějším přístupem využívajícím výjimky.
- Existence skrytých závislostí v kódu – tento problém lze částečně eliminovat použitím automatizovaného zpracování klíčových proměnných ve třídách technologií. V novém systému je toho dosaženo použitím anotací v kódu a mechanismu reflexe.

Výsledky reingeneeringu

Během reingeneeringu došlo v systému k významným změnám, které zlepšují čitelnost zdrojového kódu a usnadňují údržbu systému. Příkladem těchto změn je zavedení použití typovaných kontejnerových typů (generik) či náhrada některých proprietárních datových typů jejich ekvivalenty z nových verzí platformy J2SE. Mezi další vylepšení patří použití anotací, což jsou metainformace obsažené ve zdrojovém kódu, které v systému pomáhají automatizovat některé operace nad třídními proměnnými. Na aplikační úrovni bylo možné díky použitému webovému rámci zlepšit způsob lokalizace sytému. V současné době také probíhá práce na integraci genetických algoritmů do jádra systému, které umožní výběr optimálnějších vstupů pro jednotlivé technologie a tedy minimalizaci energetické spotřeby.

ZÁVĚR

Ukazuje se, že použití výše popsaných technik a metodik může významně ulehčit všechny fáze procesu reengineeringu existujícího softwarového systému. Na úrovni obnovení návrhu lze úspěšně použít automatické nástroje pro generování diagramů tříd na základě existujících zdrojových kódů. Na úrovni detekce problémů lze využít softwarové metriky, které nám pomohou identifikovat kritická místa v systému. Zde je však potřeba brát v potaz také to, že metriky mohou indikovat i falešné varování, takže je vždy nutné ověřit výsledky měření nahlédnutím do zdrojového kódu. Při analýze problémů lze využít návrhové vzory, které představují optimální řešení typických programových situací.

I pro změny na úrovni kódu lze použít některé automatizované nástroje. Celý proces je pak vhodné podpořit vytvořením automatizovaných testů, které nám umožní ověřit správnost renovovaného systému. Je také důležité si uvědomit, že reengineering je součástí životního

cyklu systému, tj. že každý systém se dříve nebo později dostane do stavu, kdy je potřebné přistoupit k reengineeringu. Tento okamžik však lze oddálit důsledným dodržováním správných objektově orientovaných přístupů v době návrhu systému.

Jedním ze základních pravidel pro úspěšnou tvorbu softwarových systémů je použití standardních řešení a frameworků. Proto bychom se měli ve fázi reengineeringu snažit migrovat systém k těmto řešením. Vývoj a údržba webového subsystému je pak totiž zajišťována širokou vývojářskou komunitou a nehrozí nebezpečí, že se použité řešení brzy stane nadmíru komplikované či dokonce nepoužitelné.

LITERATURA

[1] Bauer, M.:, Analyzing Software Systems by Using Combinations of Metrics, Proceedings of the ECOOP'99Workshop, June 1999.

[2] Bieman, J. a Kang, B.:, Cohesion and Reuse in an Object-Oriented System, Proceedings of the ACM Symposium on Software Reusability, April 1995.

[3] Bloch, J.: , Effective Java, Addison Wesley, June 2001.

[4] Tegarden, T. a Sheetz, S. a Monarchi, D.: , A Software Complexity Model of Object-Oriented Systems, Decision Support Systems vol. 13, 1995.

[5] Chidamber, S. a Kemerer, C.: , A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering vol. 20, June 1994.

[6] Chikofsky, E. a Cross, J.: , II. Reverse engineering and design recovery: A taxonomy, IEEE Software, January 1990.

[7] Churcher, N. a Shepperd, M.: , A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering vol. 21, March 1995.

[8] Demeyer, S. a Ducasse, S.: , Metrics, Do They Really Help?, HERMES Science Publications, Paris, 1999.

[9] Deursen, A. a Klint, P. a Verhoef, C.: , Research Issues in the Renovation of Legacy Systems, Springer-Verlag, 1999.

[10] Demeyer, S. a Ducasse, S. a Nierstrasz, O.: , Finding Refactorings via ChangeMetrics, Working paper, April 1999.

[11] Erni, K. a Lewerentz, C.: , Applying Design-Metrics to Object-Oriented Frameworks, IEEE Computer Society Press, 1996.

[12] Etzkorn, L. a Bansiya, J. a Davis, C.: , Design and Code Complexity Metrics for OO Classes, Journal of Object-Oriented Programming, 1999.

[13] Ducasse, S. a Demeyer, S.: , The FAMOOS Object-Oriented Reengineering Handbook, http://www.iam.unibe.ch/_famoos/handbook/, 1999.

[14] Galdiera, G. a Basili, V.: , Reusing existing software, Technical report CS-TR-2116, University of Maryland, College park, 1988.

[15] Henderson, B.:, OO Metrics: Measures of Complexity, PrenticeHall, 1996.

[16] Hitz, M. a Montazeri, B.: , Measure Coupling and Cohesion in Object-Oriented Systems, Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95), October 1995.

[17] Hitz, M. a Montazeri, B.: , A Measurement Tudory Perspective, IEEE Transactions on Software Engineering vol. 22, April 1996.

[18] Humphrey,W.: , Introduction to the Personal Software Process, SEI Series in Software Engineering. AddisonWesley, 1997.

[19] Li,W. a Henry, S.: , Maintenance Metrics for the Object Oriented Paradigm, IEEE Proceedings of the First International Software Metrics Symposium, May 1993.

[20] Lorenz, M. a Kidd, J.: , Object-Oriented Software Metrics: A Practical Approach, Prentice-Hall, 1994.

[21] Riel, A.: , Object-Oriented Design Heuristics, Addison-Wesley, May 1996.