Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Jan Kubr

Podpora metod specifikace požadavků

Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. RNDr. Jaroslav Král, DrSc.
Studijní program: Informatika
Studijní plán: Softwarové inženýrství

Studijní program: Softwarové systémy

Poděkování

Rád bych na tomto místě poděkoval vedoucímu své diplomové práce Prof. RNDr. Jaroslav Královi, DrSc. za vypsání zajímavého tématu, poskytnutí cenné literatury, doporučení a připomínky. Dále bych chtěl poděkoval RNDr. Janu Pavelkovi, CSc., Davidu Bílkovi, Andrew Kirknessovi a RNDr. Tomáši Rubačovi za zodpovězení anketních otázek; Hansi Thelosenovi a René Krikhaarovi, Ph.D. pak za cenné připomínky s anketou související. V neposlední řadě bych rád ocenil ochotu Ing. Jiřího Gryce ze společnosti AIT zapůjčit mi neveřejné manuály a další materiály o produktech Telelogic. Významné pro obsah této práce byly i moje pracovní zkušenosti, rád bych zde tedy vyjádřil vděčnost Ing. Jiřímu Starému ze společnosti Robert Bosch Praha, RNDr. Filipu Zavoralovi, Ph.D. z MFF UK a Siemu Vaessenovi z firmy Noterik, že mi umožnily tyto zkušenosti nabýt.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7. srpna 2006 Jan Kubr

Obsah

Název práce: Podpora metod specifikace požadavků
Autor: Jan Kubr
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. RNDr. Jaroslav Král, DrSc.
E-mail vedoucího: Jaroslav.Kral@mff.cuni.cz
Abstrakt: V současné době jsme svědky významné změny toho, jak vývojáři sbírají požadavky na software. Dvěma hlavními důvody jsou nástup servisně orientované architektury a silné zapojení uživatelů. Tato diplomová práce stručně shrnuje dosavadní vývoj na poli správy požadavků a prostředků specifikace požadavků. Poté se zaměřuje na servisně orientovanou architekturu a to, jak její další rozvoj může změnit způsob, jakým požadavky na rozsáhlé systémy sbíráme. Na základě ankety provedené pro účely této práce v softwarových společnostech a pracovních zkušeností autora pak práce formuluje doporučení, jak bychom měli ke specifikaci požadavků v současnosti a blízké budoucnosti přistupovat. Na důkaz, že přednesená doporučení jsou snadno realizovatelná v praxi, byla vytvořena malá softwarová architektura radám vyhovující.
Klíčová slova: Správa požadavků, Prostředky specifikace požadavků, Servisně orientovaná architektura, Zkušenosti firem, Agilní metody.

Title: Support of Requirements Specifications Methods
Author: Jan Kubr
Department: Department of Software Engineering
Supervisor: Prof. RNDr. Jaroslav Král, DrSc.
Supervisor's e-mail: Jaroslav.Kral@mff.cuni.cz
Abstract: There is a significant change in how we collect software requirements nowadays. The two main reasons are the emergence of service oriented architecture and the rising importance of the user involvement. This thesis briefly summarizes the past work in the requirements engineering field and gives an overview of today's requirements management software tools. It then focuses on Service oriented architecture and how its spreading can change requirements specification and management processes. Considering the survey which was conducted for this work among (not only) software companies and a case study based on author's experience, the thesis gives a recommendation on how we should collect requirements in the current situation and near future. To give a proof of concept, the recommendations were implemented in a real world application architecture.
Keywords: Requirements management, Requirements specification tools, Service oriented architecture, Companies experience, Agile methods.

1 Úvod

Pro začínajícího vývojáře se může specifikace požadavků zdát snadným úkolem. Vždyť pouze stačí zjistit, co budoucí uživatelé nového systému chtějí. Ale již záhy při podrobnějším rozmýšlení, jak takové „zjišťování“ má probíhat, narazíme na problémy. Kdo všechno se má účastnit takových jednání? Kdo vlastně přesně ví, jakou funkcionalitu má systém poskytovat? Jak vyřešit konflikty v požadavcích? Je vždy to, co je uvedeno jako požadavek, skutečná potřeba zákazníka nebo jen jeho představa, jak by systém měl jeho potřebu naplňovat? Nicméně ani po zaznamenání požadavků na ně nelze s úlevou zapomenout. Až poté přichází samotné vytváření systému, který musí dané požadavky splňovat. Je tedy třeba zajistit vysledovatelnost požadavků v implementovaném produktu a zmiňovaná specifikace podle toho musí být vytvářena. Ale ani v situaci, kdy máme funkční systém doložitelně splňující všechny požadavky, to neznamená, že produkt stoprocentně splňuje zákazníkovy potřeby. Požadavků mohlo být specifikováno málo, mohlo dojít k nedorozumění při jejich formulování, případně k jejich nepochopení při vývoji či testování.

Chyba ve specifikaci požadavků je tou nejzávažnější v celém procesu vývoje softwaru. Její náprava stojí nejvíce peněz, protože z jedné chyby se odvinulo mnoho kroků, které je třeba revidovat. Změna v požadavcích v průběhu prací je pravidlem, ne výjimkou. Tyto změny opět ovlivňují všechny fáze následující po specifikaci požadavků a je tedy nutné vytvořit jasně definovaný a opakovatelný přechod této činnosti do pozdějších fází. Jak shrnuje Pressman, správa požadavků je složitá [1, 142]. V tomto světle se specifikace požadavků ukazuje jako jedna z nejdůležitějších fází procesu vývoje softwarového produktu.

Tato práce si klade za cíl zhodnotit změny, které v procesech správy požadavků v současnosti nastávají. Jako hlavní vliv těchto změn je označena především servisně orientovaná architektura jakožto způsob, jak se vypořádat s obtížnou integrací heterogenních informačních systémů a jejich celkové komplexity. Neméně podstatnou příčinou je stále významnější role uživatelů v procesu vývoje softwarových systémů a tím i výrazné uplatnění agilních přístupů k životnímu cyklu softwaru.

Práce nejdříve stručně zhodnotí dosavadní vývoj na poli sběru požadavků a jejich správy. Poté představuje rešerši prostředků specifikace požadavků a to jak obecným přehledem jejich možností, tak recenzemi třech konkrétních produktů. Text se poté zaměřuje na servisně orientovanou architekturu a její dopady na procesy specifikace a správy požadavků, především na to, jak umožňuje v širší míře využít agilních metod vývoje. Jsou zhodnoceny i možnosti stávajících CASE nástrojů při specifikaci této architektury. Na základě představené případové studie z internetové firmy, kde autor práce působil, ankety provedené mezi významnými pracovníky (nejen) softwarových firem a dalších průzkumů je pak vypracováno doporučení, jak by se v současné době a blízké budoucnosti měly požadavky sbírat a uvádět v praxi. Na důkaz, že uvedené rady jsou v praxi realizovatelné, byla kolem reálného produktu vytvořena jednoduchá aplikační architektura radám vyhovující.

2 Definice

Definovat požadavek se zdá být snadným úkolem, přesto Davisem citovaná definice z Webster's Ninth New Collegiate Dictionary se zdá být definicí kruhem, když požadavek popisuje jako „něco, co je požadováno nebo chtěno“. Davis ale připojuje i definici z IEEE Standardu 729, který požadavek definuje jako „(1) podmínku nebo vlastnost vyžadovanou uživatelem k vyřešení problému nebo dosažení nějakého cíle; (2) podmínku nebo vlastnost, kterou musí systém splňovat nebo mít, aby se splnila smlouva, respektoval standard, naplnila specifikace nebo jiný formálně zavedený dokument“ [3, 15]. Harwell pak ještě jednodušeji uvádí, že „pokud to nařizuje, že něco musí být dosaženo, transformováno, vyprodukováno

nebo poskytnuto, je to požadavek – tečka.“ [4, 2].

Specifikaci požadavků pak podle Davise definujeme jako „úplný popis toho, *co* má systém dělat bez toho, aniž bychom popisovali, *jak* to udělá“. Zároveň ale uvádí, že „jak“ jedné osoby může být pouze „co“ osoby jiné a upozorňuje, že záleží, na jaké úrovni abstrakce se pohybujeme. Například specifikace všech možných chování systému může být již „jak“ pro zákazníka, pro designéra je to však jednoznačné „co“. Davis uvádí pět úrovní abstrakce, na které se dá dívat zároveň jako na „co“ je třeba udělat (při pohledu „zespodu“, z pohledu programátora), zároveň však i „jak“ je to třeba udělat (při pohledu „seshora“, z pohledu zákazníka), [3, 16-19].

Wieringa do specifikace požadavků zahrnuje kromě určení požadovaného chování také specifikaci cílů produktu. Stejně jako u Davise je zjištěno, že cíle a jejich naplňování se objevují na více úrovních abstrakce; je tedy upřesněno, že cíle objevující se v definici specifikace požadavků se míní cíle na nejvyšší úrovni abstrakce. Popis chování systému pak představuje cíle pro další fáze vývoje produktu [5, 71].

Pressman správu požadavků definuje jako proces, který pomáhá softwarovým inženýrům lépe porozumět problému, který mají za úkol vyřešit. Vytváří množinu úkolů, z nichž je zřejmé, jaký dopad bude mít daný software na obchod; co zákazník požaduje a jak budou koncoví uživatelé systém používat [1, 42]. Jak ale poznamenává Bosch, toto nezahrnuje pouze funkční požadavky, ale v nemenší míře i omezení na kvalitu výsledného systému. Ty by pak měly být specifikovány velmi jasně, vágní formulace objevující se v některých specifikacích jako např. „Výkon systému by měl být dostatečný pro průměrného uživatele“ nepřinášejí potřebnou ověřitelnou podmínku [2, 27-28].

Specifikace požadavků je první fází v procesu vývoje softwaru. Během ní se zahajuje komunikace se zákazníkem a je zjišťováno, jaké má mít požadovaný softwarový produkt vlastnosti. Výsledek takových rozhovorů pak ovlivňuje, jak je software navržen a vyvíjen. Pressman označuje správu požadavků jako most k návrhu a programování, je ale na pochybách, kde takový most začíná – jestli přímo u zákazníkových potřeb přeložených do uživatelských scénářů nebo u obecnější definice systému jako části větší domény. Bez ohledu na tyto rozpory, metafora mostu naznačuje, že na produkt se v této fázi díváme s nadhledem, který nám umožňuje zasadit ho do širšího kontextu potřeb uživatelů [1, 144].

3 Význam

Bylo poznamenáno, že specifikování požadavků a jejich správa je náročný proces. Nebyly ale k tomu složeny žádné důkazy, a proto někdo může mít pocit, že tato fáze je oproti technicky náročným následovníkům pouze snadné dotazování a zaznamenání odpovědí na papír.

Již na počátku sedmdesátých let bylo zjištěno, že čím později je v softwarovém systému objevena chyba, tím dražší je její odstranění. Davis uvádí data od tří firem (GTE, TRW a IBM, firmy došly nezávisle na sobě k velmi podobným výsledkům), podle kterých stojí oprava chyby pětkrát až desetkrát více, je-li místo v průběhu specifikace požadavků objevena až při programování, až padesátkrát více, je-li objevena během akceptačního testu a dokonce až dvěstěkrát více, zjistíme-li závadu po nasazení [3, obr. 1-17]. Znamená to, že opravit chyby vytvořené programováním je nákladnější než chyby například v návrhu, za předpokladu, že jsou objeveny ihned? Pravděpodobně ne (uvážíme-li navíc skutečnost, že analytici a konzultanti jsou placeni lépe než programátoři). Spíše to znamená, že chyby se neobjevují brzy po jejich zanesení do systému, ale až v některých z dalších fází. I přes některé slibné pokusy nejsou požadavky nebo návrhy systematicky (např. strojově) testovány natolik, aby výsledky testů váhou odpovídaly akceptačním testům při přebírání softwaru. Aneb je velmi těžké říct, jestli požadavky byly správně zaznamenány do doby, než se podle nich vytvoří software, který uživatel zhodnotí jako splňující jeho přání (nejlépe po dlouhodobém

používání). Je tedy velmi snadné zanést chybu v úvodních fázích vytváření softwarového produktu a objevit ji až v jedné z těch posledních nebo dokonce při jeho používání. Davis k tomuto cituje další výzkum, podle kterého 54% chyb u projektů TRW bylo objeveno až po programování a testování. Není překvapením, že 45% těchto chyb bylo připsáno dřívějším stádiím projektu, tedy specifikaci požadavků a návrhu [3, 26].

Programování vychází z návrhu a ten vychází ze specifikace a testovat lze jenom podle požadavků, které byly na systém kladeny. Vše je zdánlivě správně, pokud ale systémové požadavky nesplňují skutečné potřeby uživatelů, vytvořili jsme sice funkční systém, ale ne pro našeho zákazníka. Nezanedbání specifikace požadavků a jejich správy tedy nejenom není zbytečné, ale výrazně pomáhá při snaze vytvořit systém, který přesně naplní zákazníkovy potřeby. Navíc může značně snížit náklady, pokud se nám podaří v této fázi udělat minimum chyb a zamezit nedorozuměním.

4 Tradiční postupy

Proces specifikace a následné správy požadavků byl předmětem zkoumání softwarových inženýrů již mnoho let. Dá se říci, že tyto procesy jsou již velmi vyspělé v teorii, problematické je pouze jejich ne úplně časté dodržování v praxi, především kvůli jejich složitosti. V této kapitole budou představeny tradiční postupy specifikace požadavků a jejich správy tak, jak se za mnoho let vyvinuly pro dodržování především při vytváření rozsáhlých systémů.

Pressman rozděluje proces správy požadavků do sedmi fází: zahájení, získávání, rozpracování, vyjednávání, specifikace, stvrzení a řízení [1, 144]. Takto podrobné rozdělení předpokládá delší cykly softwarového vývoje, často pak pouze jeden v průběhu vývoje produktu a použití některého tradičního softwarově inženýrského modelu jako například tzv. vodopádu [10].

4.1 Zahájení

Dříve než vznikne požadavek na vývoj softwarového systému, je v organizaci identifikována obchodní potřeba nebo objeven nový trh a hledají se možnosti, jak potřebu naplnit nebo trh nasytit. Vznikne první představa řešení (business plán) vypracovaná typicky obchodními manažery firmy. V této fázi není dokonce ještě jisté, bude-li potřeba naplněna softwarem. Dokonce i v případě, že do hry vstupují pracovníci softwarové firmy nebo konzultanti z oblasti ICT, nemělo by to automaticky znamenat, že diskuze vyústí v použití softwarového systému. Ve fázi zahájení by měla vzniknout základní představa o potřebě, o lidech, kteří ji mají, a o povaze možných řešení [1, 145]. Důležité je uvědomit si, co vlastně firma potřebuje a tuto vizi jasně formulovat (stačí poloformálně), aby existovaly nějaké podklady pro vyhledání potenciálních partnerů (výběrová řízení), rozhodlo se o outsourcingu nebo nákupu customizovatelného řešení [11].

V ideálním případě zákazníci pracují se softwarovými inženýry v jednom týmu a je proto velmi jednoduché zjistit požadavky na systém. Toto je ale velmi řídký jev, mnohem častěji zákazníci ani nejsou na stejném místě, znalost potřeb je roztříštěna mezi více osob a mnohdy ani uživatelé nejsou schopni specifikovat, co od systému požadují. První činností ve fázi zahájení je tedy identifikovat tzv. stakeholders, tedy všechny „zainteresované“, všechny osoby mající zájem na úspěchu produktu. Toto zahrnuje nejen různé manažery zákazníka, ale také koncové uživatele systému a vývojáře. Seznam osob přirozeným způsobem narůstá, jak je každá osoba na seznamu dotázána, kdo další by podle ní měl být osloven.

Od všech takových osob je pak získána jejich představa o vytvářeném systému. Protože dotazovaných lidí může být velké množství, je téměř jisté, že požadavky budou zaměřeny výhradně nebo alespoň z větší části na oblast, ve které se daný pracovník pohybuje,

a velmi pravděpodobně se vyskytnou i konfliktní nároky. Je ale důležité, aby v této fázi bylo zaznamenáno vše bez předčasné interpretace. V této fázi jsou doporučovány nezaujaté otázky napomáhající porozumění problému jako například „Kdo bude používat toto řešení?“ nebo „Existuje již nějaký způsob, jak problém vyřešit?“. Následují otázky snažící se zjistit, jak zpovídaná osoba vidí nové řešení, například „Na jaké problémy bude podle vás toto řešení zaměřeno?“. Důležité jsou také tzv. meta-otázky jako například „Jsou mé otázky relevantní k problémům, které máte?“ [1, 150-151].

4.2 Získávání

Princip otázek a odpovědí je ale dostačující pouze pro úvodní fázi, později je třeba zajistit mnohem podrobnější získávání informací. McDermid tuto fázi definuje jako proces podrobného opatřování informací o požadované funkcionalitě a dalších vlastností systému z mnoha zdrojů, potenciální uživatele nevyjímaje [6, 4].V této fázi čelíme třem okruhům problémů: problémům šíře působnosti (hranice systému je nejasná nebo uživatelé/zákazníci uvádějí příliš mnoho technických detailů, které znejasňují hlavní účel systému), problémům v porozumění (uživatelé/zákazníci přesně neví, jakou má mít systém funkcionalitu, nevidí ho z nadhledu, opomíjejí zmínit funkcionalitu, která se jim zdá samozřejmou apod.) a problému nestálosti (požadavky se s časem často mění) [1, 145]. Doporučovaným postupem pro předejití pozdějším potížím je po iniciální fázi, ze které vznikne základní jedno až dvoustránkový požadavek na produkt, zorganizovat pravidelná setkání a při nich dodržovat následující zásady:

- Setkání se účastní jak vývojáři tak zákazníci spolu s dalšími „stakeholders“.
- Jsou stanovena pravidla pro přípravu a aktivitu.
- Program je stanoven dostatečně přesně, aby se probralo vše důležité; zároveň je však rozumně flexibilní, aby se dostalo na nové myšlenky.
- Existuje koordinátor, který kontroluje schůzi.
- Nápady jsou efektivně zaznamenávány (nástěnky, nalepovací štítky, elektronické fórum apod.). Užitečné by v této fázi mohlo být užití tzv. wiki, tj. webové stránky, která může být editována více uživateli.
- Cílem je identifikovat problém, najít prvky řešení, prodiskutovat různé přístupy a navrhnout úvodní množinu požadavku na řešení, to vše v atmosféře orientované na splnění cíle.

Během schůzí je každý zúčastněný požádán o seznam objektů, které jsou podle něj součástí prostředí nového systému, objektů, které má systém vytvářet, a objektů, které jsou systémem používány pro zajištění jeho funkcí. Ke každému objektu by pak měla být připojena informace, jaké části systému (služby, funkce) s ním operují. Každý účastník je také požádán o seznam omezení, která jsou na systém kladeny, stejně jako nároky na výkonnost. Není předpokládáno, že tyto seznamy budou od každého participanta úplné. Právě naopak chceme, aby shromážděné informace obsahovaly specifické pohledy osob z různých stran [1, 154].

Tato fáze se zdá býti nejzásadnější také proto, že její důležitost může být často podceňována. Někteří pracovníci mohou mít pocit, že požadavky jsou pouze zaznamenávány, že existuje jasná představa zákazníků o novém systému a úkolem je pouze tuto představu získat a zaznamenat. Ale jak poznamenává McDermid, požadavky jsou většinou nestabilní a objevují se pouze jako vedlejší efekt fáze získávání [6, 5]. Tedy získávání nelze chápat jako pouhé zaznamenání jasných odpovědí na jednoduché otázky, ale jako na komplexní rozhovor o zákazníkových potřebách, z něhož požadavky vyplynou po delší úvaze. Jinak se stane, že nejsou zaznamenány skutečné potřeby a ty jsou pak naplňovány pozdějšími fázemi vývoje, což může vést k neúspěchu projektu [6, 5].

Ve fázi sběru informací by se nemělo kritizovat ani diskutovat, aby se předčasně nezavrhla nadějná myšlenka. Až nad úplným seznamem podnětů se vede diskuse, vynechává se a cizeluje. To vše pod vedením koordinátora, který dbá na zachování celkové struktury specifikace požadavků. Koordinátor by také měl dbát na to, aby se dostalo na každého, aby si účastníci vzájemně neskákali do řeči, aby se zkrátka hrálo fair-play [11]. Délka jednání by neměla být přílišná, navíc by se měly dělat časté přestávky.

4.3 Rozpracování

Poté je velký tým rozdělen do několika menších, které provedou mini-specifikaci částí sesbíraných požadavků tím, že je rozvedou a uvedou ke každému více podrobností. Tyto mini-specifikace jsou poté diskutovány ve velkém týmu a opravovány a zpřesňovány. Ke každé je připojen seznam kritérií a jejich hodnot, které by se měly objevit ve finálním projektu a které budou sloužit jako podklad pro ověřování funkčnosti. Nakonec je jeden z účastníků schůze nebo někdo externí požádán o sepsání kompletní specifikace.

V této fázi je velké nebezpečí, že se odchýlíme od původního záměru, je tedy důležité, aby se této činnosti zúčastňoval jak uživatel (který má konkrétní představu o budoucím užívání díla), tak někdo, kdo má představu o celkovém účelu systému [11]. Protože tedy spolupracuje mnoho různých profesí, je třeba najít společný jazyk, aby navrhovaná funkcionalita byla chápána všemi s co nejmenšími rozdíly. Specifikace by měla obsahovat úvodní text (v jazyce přístupném všem, je tedy dobré uvést i slovníček složitějších pojmů) vysvětlující účel, rozsah a základní funkce systému. Následuje pak rozepsaná funkcionalita jednotlivých částí budoucího produktu. Ta může být v některém z formálních jazyků, ten je ale nepoužitelný kdekoliv, kdy by specifikaci měli rozumět její budoucí uživatelé, což je ovšem případ většiny softwaru (výjimkou mohou být například komunikační protokoly, kde je formální popis systému výhodný pro ověření správnosti specifikace) [11]. Ale i pro systémy s tradičními uživateli je vhodné zvolit nějakou jednotnou strukturu, aby nedošlo k nedorozuměním z dezinterpretací.

4.3.1 Uživatelské scénáře

Kdyby totiž byly požadavky na systém pouze sepsány jako body, navazující fáze by mohly být příliš pomalé, protože tým by nemusel mít jasnou představu, jak různí uživatelé budou produkt používat. Také zákazník by si nemusel udělat dostatečnou představu o tom, naplňují-li požadavky jeho potřeby, případně by mohlo dojít k nedorozumění, jak se který požadavek po implementaci projeví na funkcionalitě. Vhodnou technikou, jak všem stranám zajistit lepší porozumění systému, jsou uživatelské scénáře (často nazývané use-casy, případy užití). Ty popisují systém z pohledu různých tříd uživatelů nebo jiných interagujících entit, kdy je přesně specifikováno, jaké kroky daná entita musí podniknout, aby využila implementovanou funkcionalitu.

Use-casy mohou mít různou formu, od souvislého textu přes specifickou šablonu po diagramy. Prvním krokem při vytváření uživatelských scénářů je identifikace aktorů. Těmi mohou být nejen uživatelé, ale i jiné systémy a zařízení, které se systémem komunikují. Aktoři reprezentují roli v kontextu daného use-casu a mohou zastupovat více uživatelů, stejně jako více uživatelů může vystupovat ve více rolích (jako více aktorů) v různých scénářích. Aktory můžeme formálně rozdělit do dvou typů, kde primární aktoři přímo a často pracují se systémem a benefitují na jeho funkcionalitě, zatímco sekundární aktoři interagují se systémem, pouze aby podpořili ty primární v jejich činnosti. Pressman dále cituje a rozšiřuje Jacobsonem navrženou sadu otázek, které by měl být každý use-case schopen zodpovědět [1, 160]:

- Kdo jsou primární aktoři, kdo sekundární?

- Jaké jsou cíle těchto aktorů?
- Jaké podmínky předpokládáme před zahájením scénáře?
- Jaké hlavní úkoly nebo funkce jsou aktorem prováděny?
- Jaké výjimky mohou nastat během scénáře?
- Jaké varianty jsou ve scénáři možné?
- Jaké systémové informace aktor získá, vytvoří nebo změní?
- Bude muset aktor informovat systém o změnách v prostředí mimo náš systém?
- Jaké informace se aktor snaží získat?
- Chce být aktor informován o neočekávaných změnách?

Na základě use-casů se pak vytváří analytický model. Ten v každém okamžiku odráží nasbírané požadavky, a proto se předpokládá, že se bude často měnit. V analytickém modelu můžeme použít mnoho typů vyjadření, často ve formě diagramů: elementy založené na scénářích, třídách, na popisu chování nebo na popisu toku informací. Často se zde také používají návrhové vzory, jakési již vyzkoušené šablony, které pouze zasadíme do našeho kontextu.

4.3.2 UML

I v případě aplikování zásad pro vytváření uživatelských scénářů může pro nás být specifikace požadavků nedostatečně standardizována. I při respektování pravidel uvedených výše totiž můžeme použít různé způsoby zápisu a může tedy docházet k nedorozuměním, co která část uživatelského scénáře znamená. Systém také můžeme chtít popsat podrobněji z různých úhlů pohledu a použít přitom standardizovanou notaci. Ačkoliv Unified Modeling Language (UML) pokrývá mnohem více úvodních fází softwarového cyklu, některé jeho diagramy lze použít i pro zachycení či zpřesnění specifikace požadavků.

UML je neproprietární specifikační a modelovací jazyk, jehož specifikace byla vytvořena skupinou Object Management Group (OMG); jazyk je momentálně ve verzi 2 [7]. UML používá především grafickou notaci a snaží se umožnit zevrubný popis především objektově orientovaného softwaru. UML zahrnuje velké množství typů diagramů, pro specifikaci požadavků bychom nejčastěji použily ty, které popisují systém obecně bez vazby na konkrétní implementaci a zároveň takové, které jsou snadno pochopitelné i pro pracovníky neznalé standardů UML, protože spolupráce s nimi (uživateli) je v průběhu specifikace požadavků zásadní. Mezi tyto diagramy můžeme kromě případů užití zařadit i sekvenční diagram, diagram aktivit, nasazení a komponent.

Diagram případů užití obsahuje tzv. případy užití samotné (horizontální elipsa s textem) reprezentující nějakou akci, aktory reprezentující uživatele systému a asociace znázorňující vazbu aktora k případu užití. Je-li aktor asociován s případem užití, znamená to, že má možnost vykonat akci popsanou textem v oválu (případně samozřejmě dalším přidruženým textem). Nepovinné prvky jsou pak obdélníky vyjadřující hranice systému a balíčky rozdělující větší diagram na více logických částí pro zvýšení přehlednosti. Případy užití mohou záviset i jeden na druhém, kde nejčastější vazbou je, když jeden scénář rozšiřuje funkcionalitu jiného (vazba typu extend, například když se z nějaké obrazovky dá přejít na jinou obsahující doplňkovou funkcionalitu), kdy je jeden scénář rozdělen na více (vazba typu include, například když se složitější podčást scénáře oddělí do samostatného elementu) nebo kdy jeden scénář je potomkem jiného (vazba typu generalize/specialize, kdy z obecného uživatelského scénáře vytváříme specializované potomky). Ačkoliv je diagram případů použití velmi přehledný, daní je že samotný nevyjadřuje mnoho informací. Velmi často se ke každé elipse připojuje dlouhý text vysvětlující její účel, tento text však nemá nijak standardizovanou notaci.

Sekvenční diagram zobrazuje posloupnost procesů v nějaké podmnožině

implementovaného systému. Diagram obsahuje části systému a posloupnost akcí vykonávanou těmito částmi. Jelikož části systému nemusí být nutně nízkoúrovňové třídy, je možné tento diagram použít i pro dostatečně abstraktní popis chování systému, například pro objasnění funkcionality nějakého případu užití. *Diagram aktivit* lze použít pro stejný účel jako diagram sekvenční, nabízí však více možností, jak reprezentovat více možných variant fungování systému, protože je možné do diagramu vkládat podmínky, za jakých se aplikace vydá danou cestou, cesty lze rozdělovat a zase spojovat apod.

Diagram komponent zobrazuje oddělitelné části systému a jejich vzájemné závislosti. Není-li příliš technický, tento diagram může dát dobrou představu o celkové architektuře i pro netechnicky orientované zákazníky. *Diagram nasazení* pak představuje celkovou architekturu tak, jak bude nasazena na fyzických strojích. Zobrazuje tedy hardware, software běžící na tomto hardware a middleware jednotlivé uzly architektury propojující [8].

Unified Modeling Language je velmi komplexní nástroj s rozsáhlými možnostmi popisu systému z více úhlů pohledu. Je výhodné použít jeho standardní notaci, když si chceme být jisti, že všichni zainteresovaní rozumí jednotlivým prvkům diagramů stejně (případně si mohou jejich význam snadno zjistit). Kritici UML vyčítají přílišnou komplexitu a tvrdí, že mnoho jeho částí je nadbytečných a používaných jen velmi zřídka. Jiní kritici naopak namítají, že jazyk je příliš vágní na to, aby byl schopen realizovat skutečně formální popis systémů [9].

4.4 Vyjednávání

Málokdy nastane ideální případ, kdy po sepsání specifikačních dokumentů máme všechny informace k postupu do dalších fází vývoje softwaru a kdy jsou všechny strany spokojeny. Častěji je ještě třeba projít fází vyjednávání, kdy je zákazník požádán, aby vyvážil náklady s funkcionalitou a výkonností nového systému. Dokonce jsou-li všechny požadavky kompletní, cena potenciálního řešení se může ukázat jako příliš vysoká a zákazník bude nucen z nároků slevit, nechce-li navýšit rozpočet. Poté McDermid doporučuje klást otázky jako „Jak moc by vám daná funkcionalita chyběla?“ pro zjištění, jaké požadavky je možné vynechat [6, 7]. Pokud tedy nastane situace, že se navrhovaný projekt zdá příliš rozsáhlý, tj. buď příliš drahý nebo nadměrně náročný na realizaci, je třeba vybrat požadavky, které jsou pro zákazníka nejdůležitější. Kromě dotazování zákazníka je možné využít i fundované analýzy, která odhalí úzká (problematická) místa ve fungování podniku; požadavky na řešení těchto problému pak ve specifikaci být musí [11]. Cílem této fáze specifikace požadavků je vytvoření realistického plánu projektu.

Vyjednávání je věčně diskutované téma, velmi častou radou ale je snažit se dosáhnout tzv. výsledku vítěz-vítěz. Po konci takového jednání jsou všechny strany spokojeny a nikdo nemá pocit, že musel ustoupit příliš (tj. že prohrál). Pressman cituje Boehmovu sadu vyjednávacích aktivit odehrávajících se na začátku každé fáze vývoje softwaru (speciálně tedy i ve fázi specifikace požadavků) [1, 170]:

- Identifikace klíčových stakeholders.
- Zjištění cílů těchto stakeholders, zjištění podmínek, za kterých odejdou z dané fáze podle svých představ úspěšně, tj. jako vítězi.
- Vyjednání takového stavu, který se bude shodovat s „vítěznými“ podmínkami zákazníka a zároveň bude splňovat i vítězné podmínky ostatních.

Velmi důležitá je důvěra mezi partnery, protože zamlčování podstatných informací a neochota sdělit skutečné potřeby může vést k situaci, kdy smlouva bude výrazně nevýhodná pro jednu ze stran. Vítězství protistrany je ale pouze zdánlivé, protože taková situace vede dlouhodobě i ke ztrátě zdánlivého vítěze. Například se může zdát výhodné získat nefér lukrativní zakázku, získat takovou druhou od stejného zákazníka je ale nemožné, od někoho

jiného pak obtížné (protože daná společnost nemá dobré reference). Pokud nebyl poškozen zákazník, ale konkurenti, firma se dostává do izolace apod [11].

4.5 Stvrzení

Po specifikování všech požadavků jsou jejich jednotlivé prvky zákazníkem ohodnoceny prioritami; dále jsou prověřeny, jestli jsou úplné, jednoznačné a konzistentní. Fáze stvrzení nám pomáhá nabýt jistoty, že specifikace požadavků reprezentuje přesně to, co je od systému požadováno a je většinou prováděna s lidmi, se kterými se komunikovalo ve fázi získávání [6, 4]. McDermid dále cituje definici kvalitní specifikace požadavků: „říká vše, co je potřeba při návrhu systému, který bude uspokojovat potřeby zákazníka/uživatele a nic navíc“. Závěr definice je zdůrazňovaný, je třeba se vyvarovat přílišným detailům, které v tuto chvíli nejsou relevantní, mohlo by to příliš omezit a tím prodražit další fáze vývoje [6, 6]. Pro ověření specifikace se provádí takzvaná revize. Ta by měla zahrnovat následující otázky. Tyto otázky de facto určují vlastnosti kvalitně specifikovaných požadavků (z [1, 171] a [11]).

- Je seznam požadavků úplný? Tj. uvádí ve všech oblastech (funkcionalita, rozhraní na okolí, výkon) dostatečné množství podrobností a vazeb včetně reakcí na případné chyby?
- Je každý požadavek v souladu s celkovým účelem produktu?
- Jsou zřejmé důvody, proč byl daný požadavek implementován?
- Mají všechny požadavky odpovídající úroveň abstrakce? Především, neuvádí některý z požadavků příliš technických detailů nevhodných v této rané fázi?
- Je každý požadavek dobře ohraničený a jednoznačný?
- Má každý požadavek přiřazenou kompetentní osobu?
- Koliduje některý požadavek s nějakým jiným? Např. požaduje nějaký požadavek funkcionalitu, která je vyloučena jiným požadavkem?
- Je možné implementovat každý požadavek s technologickým zázemím, které bude projekt mít?
- Je každý požadavek testovatelný?
- Reflektují požadavky správně informace, funkci a chování budoucího systému?
- Jsou požadavky konzistentní? Tj. jsou při řešení podobných problémů použity stejné nebo podobné vzory?
- Jsou požadavky snadno modifikovatelné? Zde především požadujeme, aby se změna dělala pouze na jednom místě, dále pak aby místo změny bylo snadno dohledatelné.
- Bude možné každý požadavek identifikovat ve výsledném produktu? (Více v popisu fáze řízení)

4.6 Řízení

Řízení je sada aktivit zahrnující identifikaci, kontrolu a sledování požadavků a jejich změn. Ve fázi identifikace je každému požadavku přiřazen jednoznačný identifikátor, poté je možné vytvořit sledovací tabulky (traceability tables). Tyto tabulky dávají do vztahu požadavky s jedním nebo více aspekty vyvíjeného systému nebo jeho okolí. Může existovat mnoho druhů sledovacích tabulek, Pressman uvádí následující [1, 148-149]:

- Sledovací tabulka funkcionality. Ukazuje, jak souvisejí požadavky s důležitými požadavky zákazníka na funkcionalitu.
- Sledovací tabulka zdrojového kódu. Identifikuje zdroj každého požadavku.
- Sledovací tabulka závislostí. Ukazuje, jak požadavky vzájemně souvisí.
- Sledovací tabulka subsystémů. Kategorizuje požadavky podle subsystému, na který kladou nároky.
- Sledovací tabulka rozhraní. Ukazuje, jak požadavky souvisí s interními i externími

rozhraními.

Sledovací tabulky jsou často (například specifikačním software) nazývané matice závislostí, čímž se myslí sledovací tabulky obecně, nejen ty pro závislostí požadavků mezi sebou. Sledovací tabulky mohou mít následující tvar, kde aspekt může představovat i osobu ze zákazníkovy strany, která danou požadavek vznesla:

Požadavek / Aspekt systému	***A1***	***A2***	***A3***	***...***	***Ai***
P1		√			√
P2					√
P3	√	√			
...					
Pi			√		

Sledovací tabulky dávají lepší přehled o požadované funkcionalitě, kdo ji vyžaduje a jak tyto požadavky vzájemně souvisejí. Jsou výhodné především ve velkých projektech majících stovky identifikovatelných požadavků [1, 148].

Ve fázi řízení je také důležité mít jasně definovaná pravidla, jak se do požadavků zanáší změny a jak se pak změny projevují na dalších fázích vývoje. Stejně jako je důležité vysledovat požadavky v implementovaných částech, podstatné je i zaručit, že v požadavcích provedené změny budou opravdu realizovány. Je také důležité udržovat historii změn i s identifikací jejich autorů, aby se takto provedené úpravy mohly případně revidovat. Ve všech činnostech vývoje musí být jasné, na jakých verzích požadavků jsou postaveny.

4.7 Standardy

Kromě standardních vyzrálých softwarově inženýrských metod byly vytvořeny i standardy, které se snaží cyklus vývoje softwaru ještě více sjednotit. Pionýrem v této oblasti je americké ministerstvo obrany, které přišlo se standardem MIL 1679 standardizujícím procesy při vývoji softwarového produktu. Standard se vyvinul v DoD 2167, který se pak sloučil s MIL 7935 používaným pro kritické systémy, čímž vznikl MIL 498. Z těchto norem a standardu IEEE 1074 pak vznikl nejnovější „evropský“ ISO 12207, jež byl poté s úpravami převzat zpět Spojenými státy [13].

ISO 12207 je standard pro softwarově orientované procesy, aktivity a úkoly a je určen pro nákup softwaru na zakázku (tedy ne pro nákup již hotových produktů) a to pro obě strany, dodavatele i zadavatele. Představuje množinu procesů, ze kterých se podle jeho autorů skládá softwarový projekt, se důrazem na snahu v organizaci implementované procesy neustále zlepšovat. Standard definuje procesy (každý patří do jedné z množin primární, organizační nebo podpůrné) jak pro dodavatele, tak objednavatele, každý se pak skládá ze souvisejících aktivit a jeho efekt je transformace vstupů na nějaké výstupy [14]. Standard nepředepisuje žádný konkrétní model softwarového cyklu, ani formát či obsah dokumentací, lze tedy použít v širokém spektru organizací a projektů. Pokud bychom chtěli dodržovat detailnější standardy, musíme doplnit obecný ISO 12207 normami jinými. Pro detailní pravidla pro specifikaci požadavků můžeme použít například IEEE 830 nebo IEEE 1233, které jsou právě doporučenými postupy a návody pro vytváření specifikací požadavků na systémy či specificky software.

Standardů a doporučení je velké množství pro všechny části životního cyklu softwaru, nejčastěji pak pocházejí kromě standardizačních organizací ze státních institucí (armády, ministerstev). Jeden z mnoha, pocházející od kanceláře pro obchod britské vlády, se jmenuje Structured Systems Analysis and Design Method (SSADM) a reprezentuje právě tradiční dokumentově-orientovaný přístup k softwarovému cyklu. Dalšími jsou například Scenario-based Requirements Engineering (SCRAM) nebo STARTS.

5 Agilní metody

Vedle tradičních softwarově inženýrských metod pro vývoj softwaru vznikla také skupina tzv. agilních metod. Většina vzniklých agilních technik se snaží minimalizovat riziko tím, že software vyvíjí v kratších časových intervalech (typicky délky jeden až čtyři týdny) chápaných jako iterace. Každá iterace pak představuje jakoby samostatný projekt miniaturní velikosti, ve kterém jsou přítomny všechny fáze z tradičních metod vývoje nutné k implementování nové funkcionality: plánování, specifikace požadavků, návrh, implementace, testování a dokumentování. Ačkoliv na konci takových fází nemusí být produkt připravený k uvolnění (např. software nepovýší na novou verzi po každé iteraci), cílem agilních metod je mít funkční a teoreticky publikovatelný produkt po skončení každé iterace. Funkční software je tedy hlavním měřítkem postupu v projektu. Vezmeme-li v úvahu skutečnost, že je preferována osobní komunikace, ukazuje se, že agilní metody navíc produkují daleko méně písemné dokumentace než metody tradičnější [12].

V roce 2001 zástupci mnoha agilních technik vytvořili Manifest agilního vývoje softwaru prezentující myšlenky, které všechny agilní metody spojují. Mezi nimi je významná brzká a poté častá dodávka fungujícího prototypu, schopnost reagovat na rychle se měnící požadavky, blízká osobní spolupráce všech zainteresovaných osob (nejen technických pracovníků) a zdůraznění snahy redukovat komplexitu vyvíjených řešení na minimum. Samotný manifest zdůrazňuje čtyři preference [15]:

- Individualitu a spolupráci nad procesy a nástroji
- Funkční software nad podrobnou dokumentací
- Spolupráci se zákazníkem nad vyjednáváním o podobě smlouvy
- Schopnost reakce na změnu nad držení se plánu

Jednou z nejznámějších agilních metod je extrémní programovaní, založené na podobně znělých principech: komunikaci, jednoduchosti, zpětné vazbě a odvaze [16]. Extrémní programování vzniklo ze snahy snížit náklady v prostředí neustálé změny a klade důraz na spokojenost zákazníka [17]. Programátoři by měli vítat změnu a chtít požadavky (onu zpětnou vazbu) od zákazníka tak často, jak to jen jde. Velmi důležitou součástí této metodiky je časté a podrobné testování, a to především automatické.

Agilní metody vývoje jsou doporučovány především pro menší zkušenější týmy a menší nekritické projekty, u nichž jsou pravděpodobné velmi dynamické změny požadavků.

5.1 Specifikace požadavků v agilních metodách

Použití nějaké agilní metody namísto tradiční má zásadní dopad na správu požadavků. U tradičních metod se (jak uvedeno v předchozí kapitole), velmi podrobně požadavky zjišťují, sepisují a revidují dříve, než se začne cokoliv vyvíjet, natož jen navrhovat. A ačkoliv změny v požadavcích jsou očekávané vždy, do značné míry to znamená, že před vlastní implementací by měla existovat velmi dobrá představa o výsledné podobě budoucího produktu. Jak navrhováno Pressmanem, nejdříve je třeba vyjednat a vyjasnit zákazníkovy potřeby, vytvořit specifikační dokument a výsledný produkt pak navrhovat a testovat podle této specifikace. Tyto metody lze nazvat prediktivními, protože se soustředí na podrobné dlouhodobé

plánování budoucích prací a významnější změny nejsou snadno aplikovatelné. Naproti tomu agilní metody lze označit jako adaptivní metody, protože jejich plánování je daleko krátkodobější a reakce na změnu tudíž pružnější [12]. V každé iteraci se tedy specifikuje nevelké množství požadavků, ty se implementují a na základě hotového prototypu se pak vytvářejí požadavky další. V podstatě se postupy příliš neliší od těch popsaných v předchozí kapitole, jen všechny probíhají nad daleko menší množinou požadavků, jsou tedy kratší, přímočařejší a jednodušší. Také fáze vyjednávání a stvrzení nemusí probíhat příliš formálně; několik požadavků je snadné implementovat rychle, faktické stvrzení pak probíhá nad hotovým prototypem produktu.

Kritici tradičního postupu namítají, že věštit budoucnost nelze a že především v dnešní době, kdy rychlá reakce na změny na trhu přináší významnou konkurenční výhodu. Není proto podle nich možné dopředu vyspecifikovat všechny detaily systému a to z mnoha důvodů, např. zákazník si není jistý, co vlastně požaduje; může dojít k nedorozuměním při interpretaci specifikace; zákazníkovy potřeby se mění každý měsíc apod. Jason Fried ze softwarové společnosti 37signals dokonce s oblibou píše, že specifikace přinášejí pouhou iluzi o dohodě a navrhuje sepsat pouze jednostránkový „příběh“ o budoucím produktu, začít pak budovat uživatelské rozhraní a diskutovat se zákazníkem nad ním [18]. Obecně velká část současné vlny Web 2.0 je založená na významné roli uživatelů neustále přispívající svými podněty (a také způsobem používání) do vývoje nových verzí. Protože software je přístupný přes Internet a je vyvíjen jako služby, hovoří se o konci cyklu publikování softwaru a nástupu kontinuálního zlepšování produktů podle požadavků uživatelů – zákazníků [19, 4]. Rychle se ale vyvíjejí nejen požadavky, to stejné platí pro technologie, ve kterých se software vyvíjí. V tomto prostředí velmi často pracujeme s něčím, co ještě dokonale neovládáme, co se stále ještě učíme. Navrhnout celý systém s nedokonalou znalostí technických možností našich prostředků by nemuselo dopadnout nejlépe. Inkrementální zlepšování, na nichž jsou agilní metody založeny, počítá kromě rozšiřování kódu i s jeho přepisováním a úpravami (tzv. refactoringem), do rychle se vyvíjejícího se prostředí se tedy hodí více.

Naopak kritici agilních metod argumentují, že nedostatečné plánování a příliš horečné implementování vyvstalých požadavků nakonec vede k ošklivým architekturám a návrhům a že iterativně přidávat funkcionalitu je dražší, než si vše nejprve rozmyslet a až poté – se znalostí všech vazeb a následků – ji implementovat [16].

6 Prostředky

Proces specifikace a správy požadavků představuje vícekrokový proces a velkou komunikační zátěž pro mnoho pracovníků. U velkých projektů vyvíjených podle tradičních zásad pak v jednom okamžiku vysoký počet požadavků různého rozsahu s množstvím závislostí mezi nimi. Je přitom zásadní, aby seznam požadavků byl přehledně organizovaný, aby se v ní například dalo pohodlně vyhledávat a především aby bylo vysledovatelné, které požadavky jsou již implementované, případně konkrétně jakými komponentami vyvíjeného systému. Jak podotýká Pressman, mnoho úloh správy požadavků je možno vykonávat použitím jednoduchého tabulkového procesoru nebo malé databáze [1, 149]. Často je ale výhodné – opět se to týká především rozsáhlých projektů – použít specializovaný a komplexnější systém.

Na trhu jsou dostupné desítky produktů podporujících správu požadavků, seznam těch nejpoužívanějších udržovaný na stránkách projektu Volere čítá téměř padesát položek [20]. Funkcionalita těchto systémů sahá od nepříliš složité databáze přes rozsáhlé kolaborační možnosti k schopnosti vázání požadavků na mnoho jiných CASE nástrojů (UML modelovací nástroje, projektové plánovací nástroje jako například Microsoft Project, či dokonce vývojová prostředí). Podrobně rozepsané vlastnosti mnoha prostředků specifikace požadavků lze nalézt na stránkách International Council on Systems Engineering (INCOSE) [21].

6.1 Nejčastější funkcionalita

Hlavní funkcionalitou prostředků specifikace je podpora zadávaní požadavků v přirozeném jazyce a jejich uchovávání v databázi. Je možné vytvářet textové dokumenty přímo (častá je integrace s Microsoft Word) nebo je importovat, porovnávat a identifikovat v nich jednotlivé požadavky. Některé nástroje pak podporují i identifikování automatické, založené především na vyhledávání klíčových slov (z předpřipraveného slovníku editovatelného uživatelem). Text je použit i k detekci neúplných nebo nejasných požadavků, například spojení „odpovídající standardy“ nebo „všude, kde je to nezbytné“ jsou označena a je navrženo jejich upřesnění (specificky jaké standardy, kde konkrétně je něco nezbytné).

Požadavky lze také velmi často zařazovat do kategorií a přidávat množství doplňujících informací. Těmi mohou být například subjekt, který tento požadavek vznesl, nebo osoba, která za naplnění požadavku nese odpovědnost. Podstatná je také možnost přiřazovat požadavkům prioritu, časovou náročnost a další požadované zdroje, dále pak vazbu na testovací scénař apod. Požadavky také mají svůj stav indikující například, jestli již byly zveřejněny, zdali byly schváleny zákazníkem či již plně implementovány. Většina nástrojů nabízí předpřipravené šablony pro více různých typů požadavků, funkčních i ostatních.

Velmi důležitým aspektem procesu správy požadavků je jejich vystopovatelnost ve výsledném produktu. Bez ní je těžké říci, jestli vlastně vytvořený software odpovídá vzneseným požadavkům. Prostředky specifikace požadavků tedy obsahují funkce tento proces podporující: Každý požadavek je možné navázat na entitu tento požadavek naplňující, jedná se tedy o realizaci sledovacích tabulek popsaných výše v sekci Řízení. Entitou může být část WBS, architektury nebo modelu systému (například třídy z objektově orientovaného návrhu, UML diagramy aktivit atp.), požadavek na změnu v příslušné externí databázi, proces požadavek implementující (například proces instalace) nebo dokonce i zdrojové soubory. Některé nástroje umožňují vytvářet jednoduché modely systému přímo v aplikaci, častá je ale vazba na externí programy, především UML (obecněji MDA [60]) modelovací prostředí, ale také vývojová prostředí. Vytvořit vazbu lze jak od požadavku k aspektu systému, tak od entity k požadavkům nebo jejich skupině. Takto lze i přidat nové požadavky vázající se na prvky systému, jako například nároky na výkon, cenu, přenosovou rychlost apod. Vazbu požadavků na prvky systému lze často kromě textové identifikace zobrazit i graficky. Spojení lze pak samozřejmě následovat, aby bylo možné vysledovat, jaké požadavky jsou kladeny na jakou část systému a naopak jaké entity implementují nějaký požadavek.

Obvykle jsou požadavky vzájemně závislé, například když je ve fázi Rozpracování obecnější požadavek rozložen na několik podřízených. Nástroje podporují tuto funkcionalitu možností vytvoření vazby mezi požadavky; tato vazba navíc může být pojmenovaná. Příjemnou funkcí pro uživatele je pak vyhledávání nekonzistencí v požadavcích, především požadavků bez jakékoliv vazby na entity, které jej implementují, a zároveň bez jakýchkoliv podřízených požadavků. Takový požadavek by měl být buď na něco navázán nebo smazán jako redundantní nebo nepotřebný. Toto se tedy týká především fáze Stvrzení, kdy jsou požadavky sepsány, ale ještě se nezačalo s dalšími fázemi životního cyklu softwaru a je třeba ověřit, že požadavky jsou smysluplné a konzistentní.

Možnost ověření, zda a jak byl nějaký požadavek implementován je další důležitá funkcionalita popisovaných nástrojů. Často existuje možnost u požadavku označit, že již byl implementován spolu s podrobnými informacemi, jak, kdy a kým tak bylo učiněno. Pro vybrané části systému lze pak zobrazit seznam již implementovaných požadavků stejně jako těch, které na naplnění ještě čekají. Pokud jsme zadali časový odhad naplnění požadavku, lze pak porovnat skutečný čas s tímto odhadem. Možnosti statistik a sestav mohou být rozsáhlejší, lze například zobrazit odhadovaný čas dokončení celého systému (na základě odhadů ve všech požadavcích), již dokončené komponenty (komponenty, jenž mají již implementované všechny požadavky) apod. Mnoho nástrojů také umožňuje vytvářet vlastní

dotazy nad databází požadavků (například formou pokročilého vyhledávání).

Mnoho požadavků je během celého vývoje několikrát modifikováno, je proto třeba udržovat historii těchto změn spolu s informacemi, kdo a kdy změnu provedl a především důvod, proč byla změna do požadavku zanesena. Příjemné je také zvýraznění změn v jednotlivých verzích, například změny od poslední verze od vybraného uživatele. Překvapivě mnoho nástrojů podporuje správu verzí do jisté míry tak, jak ji známe ze správy verzí zdrojového kódu. Máme možnost vytvořit separátní větve (branches, ta hlavní je pak často nazývána baseline) požadavků a oddělit tak různé varianty produktů nebo například začít specifikaci požadavků nového systému založeného na jádře projektu existujícího. Jsou-li například všechny naše produkty založené na stejné architektuře, požadavky na ni jsou neměnné (nebo je změn velmi málo); stejně jako mohou být některé obecné požadavky (nároky na uživatelské rozhraní, výkon, cenu za hodinu práce apod.). Můžeme tedy z této množiny vytvořit baseline a požadavky pro konkrétní produkty pak sbírat do jen pro tento účel vytvořené větve. Aktuální verze požadavků identifikovaných nějakou větví se pak dají označit „nálepkou“, například podle verze projektu, ve kterém jsou tyto požadavky naplněny.

S možnostmi správy verzí souvisí i požadavek na více úrovní uživatelských práv, uživatelských skupin a možností zamknout čtení nebo modifikaci požadavků nebo celé jejich skupiny pro vybrané uživatele. Ačkoliv se požadavek na zamčení čtení nějakého požadavku může zdát nepotřebný, může být výhodný v případě, kdy si nějaký uživatel přeje vidět jen velmi málo „rodičovských“ požadavků a nezajímají ho podrobná rozpracování. Výhodné to také může být nechceme-li detaily implementace zveřejňovat třetí straně, protože ho považujeme za své intelektuální vlastnictví a konkurenční výhodu, zároveň však tento subjekt chceme částečně o specifikaci informovat, protože s námi v nějakém smyslu spolupracuje.

Specifikaci téměř nikdy nevytváří pouze jedna osoba, především je nezbytné zachycovat reakce a názory zákazníků. Je tedy výhodné, pokud náš software podporuje spolupráci více uživatelů, například možností připojit komentář k částem požadavku bez jeho změny nebo vyjádřením názoru nevázaného ke konkrétnímu požadavku. Důležité také je, aby o změnách v požadavcích byly vhodným způsobem informováni všechny osoby, pro které je změna relevantní. Jelikož je obtížné tyto osoby vůbec zjistit, natož uchovávat u každého požadavku, informováni jsou jen někteří pracovníci (například všechny osoby, které daný požadavek někdy modifikovali, a osoba za požadavek zodpovědná), ti pak dají vědět všem, kteří by měli podle nich o změně vědět. Nejčastější formou upozornění je e-mail, může se ale využít i firemní informační systém. Některé systémy a to ne nutně ty nejznámější nebo největší, jako například Cradle, dokonce samy vytvářejí vlastní „informační systém“ - webový portál obsahující všechny potřebné informace o celém projektu. Zpracovávání upozornění informačním systémem může částečně pomoci vyřešit klasický problém e-mailu: zprávy dostává buď příliš mnoho nebo nedostatečné množství pracovníků.

Obecnější funkcionalita pak zahrnuje vícejazyčnou kontrolu pravopisu a možnost exportovat nasbírané požadavky do více typů dokumentů. Těmi mohou být textové soubory volitelně vyhovující některému ze standardů specifikace požadavků, různá grafická vyjadření modelu systému nebo vazeb požadavků, statistiky a grafy. Taková reprezentace požadavků je důležitá, především když zákazník nemá přístup do našeho softwaru, protože je nutné, aby i tak specifikaci pomáhal spoluvytvářet. Též se může hodit pro třetí strany (standardizační komise, spolupracující firmy apod.). Export do standardních strojově čitelných formátů (například do dohodnutého XML) se pak hodí pro přenos databáze požadavků mezi programy (například při přechodu na nový nástroj), ať už od různých výrobců nebo více instancemi produktu stejného.

Z vlastností, které nás zajímají u každého softwaru, jsou podstatné především finanční a systémové nároky. Vyžadovaná platforma klientů produktů je velice důležitá, přispívá-li na specifikaci mnoho různých pracovníků z více firem. V dnešní době je téměř nezbytné, aby

software bylo možné obsluhovat i z webového prohlížeče, i když takové použití nemusí umožnit využít všechnu jeho funkcionalitu. Samozřejmým požadavkem je pak příjemné a jednoduché uživatelské rozhraní, které umožní i netechnickým pracovníkům s produktem efektivně pracovat a zároveň využívat pro nás důležité funkce. Pokud je například příliš komplikované vytvořit závislosti mezi požadavky, nebude tato funkcionalita uživateli příliš využívána; závislosti pak budou muset být doplněny fundovaným pracovníkem, což přinese nadbytečné náklady. Této situaci se dá částečně předejít nakoupením softwaru s velmi dobrou uživatelskou podporou. Úplně eliminovat by ji pak teoreticky mělo požádání uživatelů o absolvování školení. Toto je však velmi obtížně realizovatelné u klientů, kdy je problematické žádat je o návštěvu byť sebekratšího školení.

Uživatelská přívětivost tedy zůstává podstatným požadavkem na specifikační software. Pokud je vytvořen tak, že zákazníci jsou schopni velmi snadno a rychle zanést do systému požadavek, který pak obsahuje co nejvíce pro nás důležitých informací (jak je možné vidět výše, takových údajů může být velké množství), proces správy požadavků se stane hladší a tím i levnější a více oceňovaný spokojeným zákazníkem.

6.2 Konkrétní produkty

V této sekci zhodnotíme možnosti tří konkrétních prostředků specifikace požadavků: Borland CaliberRM, IBM Rational RequisitePro a Telelogic Doors. Ačkoliv se nemůžeme opřít o data žádného rozsáhlého průzkumu, tyto produkty jsou v oblasti správy požadavků zmiňovány nejčastěji.

6.2.1 Borland CaliberRM

Přístup Borlandu k správě požadavků je založen na existenci centrálního úložiště všech potřebných údajů. Dokumenty nejsou ve středu pozornosti tohoto nástroje, ačkoliv je možné je generovat kdykoliv je třeba. To podle autorů systému CaliberRM osvobozuje uživatele od nutnosti udržovat mnoho verzí dokumentů i větší možnosti ve sledování zavislostí mezi požadavky a dopady při jejich změnách [23].

I v CaliberRM jsou požadavky organizovány ve více projektech, v nich jsou pak uspořádány do stromové struktury, kde stromů může být více s tím, že každý reprezentuje požadavky určitého typu. Kromě předpřipravených typů (Business, Marketing, GUI apod.) program také umožňuje administrátorovi vytvářet typy nové. Ke každému požadavku se při jeho vytváření mohou zadat následující údaje: jméno požadavku, vlastník, stav, priorita a popis. Vlastníkem se stává automaticky uživatel, který požadavek vytvořil, lze však specifikovat i uživatele jiného. Popis lze editovat ve stylu pokročilejšího textového editoru, lze tedy měnit velikost písma, barevně jej zvýrazňovat apod. Požadavky také obsahují sekci uvádějící, jak se bude daný požadavek testovat. Tato sekce umožňuje zadat pouze prostý text bez podrobnější struktury. Každému požadavku lze pak také přiřadit libovolný počet zainteresovaných uživatelů, kteří budou uvědomeni vždy, když se v požadavku provede nějaká změna.

Ačkoliv dokumenty nejsou v centru pozornosti CaliberRM, je možné libovolný požadavek svázat s neomezeným množstvím dokumentů libovolného typu (včetně internetových odkazů). Pokud je takový referenční dokument typu Microsoft Word, lze se dokonce odkazovat přímo na části textu uvnitř souboru. Tento text se pak v požadavku stane buď popisem této reference nebo – je-li soubor označen jako klíčová reference – přímo popisem požadavku. Lze využít i možnosti importovat velké množství požadavků z textu typu Microsoft Word. Při takovém importu se specifikují jmenné a formátovací konvence dokumentu, aby bylo možné rozpoznat jednotlivé požadavky a jejich popisy.

Požadavky mohou mít neomezený počet atributů, opět pak předdefinovaného typu

nebo typu definovaného uživatelsky. Předdefinované atributy byly již zmíněny a jsou pouze dva: priorita a stav. Ty mají definované hodnoty (priorita čísla od jedné do pěti, stav pak může být „zadán“, „odsouhlasený“ nebo „odmítnutý“), tyto hodnoty lze však libovolně měnit. Vlastní atributy mohou být mnoha typů, od tradičních „programátorských“ jako jsou logická hodnota, datum a různé druhy čísel, přes více typů seznamů (s možností označit pouze jednu nebo více položek, kdy položka může být i skupina) po volné textové pole o definovaném počtu řádků. U všech atributů lze nastavit jejich výchozí hodnotu, jestli má jejich změna změnit hlavní číslo verze požadavku a jestli mají být po jejich změně následovány všechny vazby a jejich cíle navrženy k revizi. Uživatelem definované atributy lze také přiřadit k nějakému typu požadavku, takový atribut pak bude automaticky přítomen u všech vytvořených požadavků tohoto typu.

Vztahy mezi požadavky slouží k vyjádření jejich vzájemné závislosti, aby bylo zřejmé, co všechno změna v nějakém požadavku ovlivňuje. V CaliberRM je možné vytvořit závislost mezi libovolnými dvěma požadavky stejného nebo různých projektů, objekty rodiny produktů StarTeam (rodina systémů pro správu změn a konfigurací, také od Borlandu), množinu testů, testy nebo kroky testů z produktu Quality Center/TestDirector, objekty správy konfigurací nebo další objekty zpřístupněné pomocí doplňků jiných výrobců. Po vytvoření vazby jsou doplněny i další závislosti předpokládané díky tranzitivitě (má-li požadavek A vazbu na B a B na C, je vytvořen implicitní vztah A k C). Je-li pak změněn nějaký požadavek, všechny navázané objekty jsou označeny jako kandidáti k revizi (v případě uživatelských atributů jen pokud to bylo při jeho vytváření vyžádáno), protože učiněné změny mohou znamenat nutnost modifikace i těchto objektů. Všechny nebo jen vybrané (po aplikování filtru) odkazy lze zobrazit v matici – obdobě sledovací tabulky z kapitoly o řízení požadavků uvedené výše.

Všechny požadavky jsou verzovány a obsahují také podrobnou historii svých změn, najdeme zde všechny změněné položky včetně jejich autorů a jejich komentářů ke každé verzi. Pro vybranou množinu požadavků lze vytvořit základní větev (baseline) a prvky množiny k ní přiřadit. Baseline lze také zamykat a porovnávat – například po schválení první verze specifikace lze všechny v ní obsažené požadavky přiřadit k baseline vybraného jména, takovou základní větev pak zamknout pro další modifikace a později ji porovnat s jinou baseline, například s tou obsahující další verzi specifikace.

Jak bylo poznamenáno, specifikaci požadavků lze kdykoliv exportovat do dokumentu vybraného formátu. Implicitně jsou k dispozici tři sestavy: detailní se všemi požadavky a informacemi o nich (jméno požadavku, popis, priorita, stav vlastník a všechny uživatelem definované atributy), stavová (všechny požadavky projektu seřazené podle jejich stavu) a odpovědnostní (všechny požadavky projektu seřazené podle zodpovědných osob). Vytvořit lze však i vlastní šablonu sestavy (ve formátu Microsoft Word), systém nabízí rozsáhlé možnosti, jak takovou šablonu vytvořit včetně integrovaných příkazů, které umožní definovat položky dokumentu (například pro jeho nadpisy můžeme vložit jméno projektu a baseline příkazy $PROJECT a $BASELINE). Příkazy mohou mít ale i téměř programátorský charakter a umožní pak vytvořit seznamy mnoha druhů (záznamy o historiích požadavků, závislostech, diskuzích apod.).

CaliberRM podporuje spolupráci více osob nad stejnými požadavky tím, že k nim dovoluje uživatelům připojovat diskusní příspěvky. Každému pak přehledně zobrazuje, jsou-li v diskuzi zprávy, které zatím nebyly přečteny. Systém umožňuje vytvářet skupiny uživatelů a pro jednotlivé typy požadavků pak definovat, mají-li právo uživatelé náležející do těchto skupin požadavky daného typu vytvářet, číst, mazat

Aplikace pracuje v režimu klient-server, a tak všichni uživatelé mají možnost sdílet data a pracovat s nimi ve stejný okamžik. Klient existuje i ve webové verzi, ten však neumožňuje využít některé funkce, například zobrazit matici závislostí (sledovací tabulky),

import a export dokumentů včetně vytváření sestav a také funkce drag-and-drop. Server i tlustý klient běží pouze na operačním systému Microsoft Windows.

Borland CaliberRM působí jako snadno ovladatelný nástroj umožňující mnoho pokročilých funkcí i nepříliš technickým uživatelům. To by mohlo usnadnit komunikaci se zákazníky z jiného oboru než ICT. Velmi slibně působí možnost vytvářet poměrně libovolně strukturované sestavy. Nepříjemná jsou omezení webového klienta, který se dá použít jen jako nouzové řešení v případě nedostupnosti toho nativního.

6.2.2 IBM Rational RequisitePro

IBM charakterizuje svůj systém na správu požadavků jako efektivní a snadno použitelný nástroj, který umožňuje požadavky spravovat vyčerpávajícím způsobem, podporuje komunikaci a spolupráci v týmu a snižuje riziko neúspěchu projektů. Rational RequisitePro kombinuje sílu textového editoru Microsoft Word s rozsáhlými výhodami databáze. Produkt spravuje „živé“ specifikace požadavků ve formě textových dokumentů, jejichž části jsou svázány s položkami databáze, díky čemuž je možné nad požadavky využít tradiční silné databázové dotazovací schopnosti [22]. Požadavky je možné prioritizovat, sledovat závislosti mezi nimi stejně jako zjišťovat dopady jejich změn. Je také zaznamenávána historie každého požadavku, díky čemuž je možné sledovat (případně auditovat), jak se požadavky měnily v čase. Produkt dokonce zaznamenává historii změn i u typů požadavků. Díky integraci s dalšími produkty rodiny Rational Suite může být přechod do dalších fází životního cyklu vytvářeného produktu velmi hladký.

RequisitePro je vysoce konfigurovatelný nástroj. Nepoužijeme-li žádnou z projektových šablon, musíme sami nadefinovat vše od začátku – především typy používaných dokumentů a požadavků. Na jedné straně máme možnost chovaní produktu uzpůsobit našim specifickým potřebám, v případě běžných projektů pak můžeme použít šablonu a například ji jen lehce modifikovat. Výhodné může být vytvořit šablonu specifickou pro naši firmu a tu pak znovupoužívat ve více projektech.

Obsah je v RequisitePro uspořádán do stromové struktury. Kořen stromu reprezentuje projekt, jeho potomci jsou pak tzv. balíčky (packages). Balíčky mohou obsahovat další balíčky, pomáhají tak vytvořit obdobu adresářové struktury souborového systému. Listy stromu (v analogii pevného disku soubory) pak mohou být kromě v programu vytvořených požadavků a pohledů na ně i textové dokumenty.

Dokumenty i požadavky mají své typy; lze uvést i jméno, popis a příslušnost k balíčku. Nepoužijeme-li šablonu, nemá náš projekt žádné typy požadavků, před vytvořením prvního tedy musíme definovat alespoň jeden. Při vytváření typu opět definujeme jeho jméno a popis, zde lze však i definovat jmenné konvence – omezením, jaký podřetězec musí název obsahovat. Každý požadavek má svou značku (tag), která se skládá z uživatelem definovaného prefixu a ze systémem generovaného čísla (například pro uživatelské scénáře může definovaný prefix být „UC“, celé označení pak bude například „UC3.1“). Typy požadavků lze odlišovat i barevně a typem písma.

K typu požadavku je pak třeba přiřadit jeho atributy. Požadavky přítomné u všech typů požadavků jsou (hodnoty atributů lze vždy změnit, stejně jako celé atributy odebrat): priorita (vysoká, střední nebo nízká), stav (navržený, schválený, uskutečněný nebo ověřený), cena (reálné číslo), obtížnost (vysoká, střední nebo nízká), stabilita (opět předchozí tři hodnoty) a přiřazeno (volné textové pole). Typy uživatelských atributů mohou být následující: seznam (s tím, že bude možné vybrat pouze jednu nebo více hodnot), volný text, celé číslo, reálné číslo, datum, čas, uživatelské jméno, URL odkaz nebo jiný objekt programů rodiny Rational Suite (včetně modelů a uživatelských scénářů v Rational Rose nebo změnový požadavek z Rational ClearQuest). U každého atributu lze definovat, jestli se po jeho změně mají označit vazby na jiné požadavky jako „podezřelé“, tedy jestli existuje nutnost navázané požadavky

překontrolovat, jestli je daná změna ovlivnila natolik, že je třeba změnit i je.

Produkt je silně integrován s textovým editorem Microsoft Word. Pokud se v programu otevře dokument tohoto typu, objeví se v editoru nová položka menu umožňující vytvářet nebo modifikovat požadavky nebo zobrazit pohled. Aby byl RequisitePro schopný dostatečně kontrolovat dokumenty, některé funkce Wordu nejsou umožněny. Všechny požadavky jsou uloženy v projektové databázi, vytvořeny mohou být však v textovém dokumentu, pohledu nebo přímo v databázi. V případě vytvoření v dokumentu se na něj použijí formátovací konvence odpovídající v programu definovaném typu dokumentu. Požadavky mohou být vzájemně závislé, což vyjadřuje, že změna zdroje odkazu může způsobit nutnost změnit i požadavek v cíli vazby. Požadavky lze dokonce provázat i mezi více projekty. RequisitePro také podporuje spolupráci více uživatelů nad stejnými požadavky – je možné ke každému z nich připojit diskusní příspěvek, který lze dokonce nechat zaslat uživatelům i e-mailem. K tomu je třeba vybrat uživatele programu a definovat je jako účastníky diskuse.

Stejně jako v CaliberRM i v RequisitePro existuje možnost vytvářet skupiny uživatelů a určovat, jaká oprávnění tyto skupiny mají. Práva lze definovat pro projekty (lze přidělit právo na změnu projektové struktury nebo právo na změnu bezpečnostních vlastností projektu), pro požadavky (pouze číst, aktualizovat, vytvářet nebo mazat) a pro vazby požadavků (opět pouze číst, možnost označit cíl vazby jako kandidáta k revizi, možnost toto označení smazat a možnost vazby vůbec vytvářet a mazat).

Typ dokumentu určuje jeho formátovací konvence a vůbec celkový vzhled. Odpovídá šabloně Microsoft Wordu. Šablony dostupné v předdefinovaných projektech jsou z velké části inspirované metodikou Rational Unified Process a zahrnují: vize (obsahuje hlavní vlastnosti produktu, klíčové potřeby zákazníků a poskytované služby), vysvětlivky (glossary, rejstřík pojmů se jejich definicemi), plán správy požadavků (stanovuje pravidla pro vytváření typů dokumentů a požadavků a jejich atributů), specifikace uživatelských scénářů (představuje funkční požadavky ve formě uživatelských scénářů), specifikace doplňujících požadavků (definuje požadavky, které nelze navázat na žádný uživatelský scénář) a testovací plán (identifikuje komponenty, které mají být testovány a stanovuje jednotlivé kroky testů).

Pohled reprezentuje náhled na množinu atributů s vybranými vlastnostmi, těmi mohou být atributy požadavků a jejich závislosti. V pohledech mohou být zobrazené buď atributy nebo závislosti, ne obojí najednou. Atributy lze zobrazit v matici, závislosti pak v matici nebo stromu. Matice atributů obsahuje požadavky jednoho typu a má na levé straně jména a krátké popisy požadavků a nahoře jména jednotlivých atributů. Díky tomuto pohledu je možné požadavky řadit podle různých kritérií, schovávat či zobrazovat různé atributy a také vyhledávat v jejich hodnotách a pohled tak filtrovat. Buňky „tabulky“ pak zobrazují hodnotu příslušného atributu. Matice závislostí má na obou stranách požadavky (buď stejného typu nebo dvou různých typů na každé straně), v buňkách je pak šipkou vyznačeno, jestli a jak na sobě dané dva požadavky závisí. Program také zobrazuje, jestli byl změněn nadřazený požadavek a je třeba ověřit, není-li nutné změnit také cíl závislosti. Tento pohled zobrazuje pouze přímé závislosti. Strom závislostí zobrazuje požadavky jednoho typu a má jako předky všechny požadavky, jenž mají nějaké jiné na nich závisející – ty jsou pak zobrazeny jako potomci.

Pro požadavky v dokumentech, maticích atributů, balíčcích nebo celých projektech lze vytvořit základní větev (baseline), čímž se tato množina požadavků označí nějakým identifikátorem a vytvoří se XML soubor danou baseline obsahující. Dvě základní větve lze pak porovnat díky Baseline manageru zabudovaného do aplikace; z výsledků porovnání lze také vygenerovat sestavu tyto výsledky zobrazující. Sestavy lze generovat i z libovolného množství požadavků s definovanými atributy, historií a vztahy. Požadavky lze před vytvořením sestavy filtrovat, lze však použít i již hotový pohled.

Rational RequisitePro pracuje v režimu klient-server, kde server může být separátní pro licence, databázi a webový server. Webový server lze nainstalovat pouze na operačním systému Microsoft Windows Server, jako databázi lze pak použít některou z Microsoft Access, IBM DB2, Microsoft SQL Server nebo Oracle s omezeními, které se na to které datové úložiště vztahují. Klient běží na operačních systémech Microsoft Windows 2000 SP4 a novějších. Je k dispozici i webová verze klienta, ta umožňuje téměř všechny funkce, je ale nutné mít nainstalovaný Microsft Word, jinak není možné editovat specifikační dokumenty.

Velká přednost IBM Rational RequisitePro je jeho velká modifikovatelnost a zároveň možnost změny udržet pod kontrolou (například díky udržování historie změn typů požadavků). Silná integrace s Microsft Word je výhodou pro ty, kteří momentálně používají jen psané dokumentace a zároveň není překážkou pro ty ostatní. Použitelný webový klient navíc snižuje závislost na platformě Windows. Oproti CaliberRM se ale zdá, že možnosti exportů do sestav a reportů jsou omezenější.

6.2.3 Telelogic DOORS

Telelogic nazývá svůj produkt na správu požadavků světovým lídrem v této oblasti [24], vychází to z několika průzkumů podílů na trhu, například společnosti Gartner (podle nich byl v roce 2003 tržní podíl Telelogic DOORS celých 41%) nebo Standish Group citovaných v interních materiálech Telelogic. Produkt také funkcionalitou mírně převyšuje výše popsané programy. Stejně jako prostředky specifikace požadavků od Borlandu a IBM je i Telelogic DOORS součástí celé rodiny CASE nástrojů poskytující podporu celému softwarovému životnímu cyklu.

Základním prvkům databáze (tedy především požadavkům) se v DOORS říká objekty. Ty jsou umístěny v modulech (oddělující například obecné a nějakým způsobem specifické požadavky; modul může být chápán jako samostatný specifikační dokument), které mohou být pro zvýšení přehlednosti umístěny ve složkách nebo projektech. Je příjemné, že složky mohou obsahovat projekty stejně jako projekty mohou obsahovat složky, můžeme tak mít přehledně uspořádané jak větší množství projektů, tak mnoho modulů v rámci projektu jednoho. Hlavní okno programu připomíná Průzkumníka operačního systému Windows, na levé straně se zobrazují projekty, složky a v nich obsažené moduly, vlevo pak obsah aktuálně vybraného modulu. Modul působí jako podrobně strukturovaný textový dokument nebo záznamy tabulkového procesoru. Kromě textového popisu objektu můžeme vidět i atributy objektu nastavené pro zobrazování. Lze však také přepnout do tzv. grafického zobrazení, ve kterém jsou moduly a objekty zobrazeny jako „krabičky“, v níž se zobrazuje hodnota vybraného atributu. Toto zobrazení umožňuje snadnější orientaci v hierarchii objektů a modulů, při větším množství těchto položek se ale obtížně vše vejde na jednu obrazovku.

Vytvoření nového objektu znamená vyplnění některých základních údajů: nadpis objektu, krátký popis a hlavní text objektu. Více požadavků najednou lze však také importovat z dokumentu v některém z formátů Microsoft Word, TXT, RTF nebo CSV. Systém DOORS pro nový objekt automaticky vygeneruje tzv. číslo nadpisu odvozené z pozice objektu v hierarchii modulů. Tato čísla vypadají jako nadpisy sekcí textového editoru, například 4.2 pro druhý objekt ve čtvrtém modulu. Automaticky jsou vyplněny i některé další systémové atributy objektu, například číselný identifikátor objektu, uživatel, který objekt vytvořil, kdy se tak stalo, nebo kdo a kdy objekt naposledy změnil. Takové údaje není dovoleno žádnému uživateli měnit, bez ohledu na to, jakými právy disponuje. Kromě těchto základních atributů může objekt obsahovat libovolné množství údajů dalších, definovatelných uživatelem. Produkt umožňuje pro každý modul nadefinovat nové typy atributů (odvozené z předdefinovaných typů – kterými jsou text, řetězec, celé a reálné číslo, datum, výčet a uživatelské jméno – nadefinováním povolených hodnot, výchozí hodnoty apod.) a atributy samotné těmto typům příslušejícím. Je třeba připomenout, že atributy lze kromě objektů

přiřazovat i modulům (stejně jako jsou modulům i generovány systémové atributy důležité pro zaznamenávání historie jejich změn).

DOORS umožňuje vytvořit závislosti mezi libovolnými objekty bez ohledu na to, v jakém modulu se nacházejí. Vazby jsou uloženy v k tomu určeném modulu, popř. v jakémkoliv jiném definovaném uživatelem. Libovolný objekt může být cílem i zdrojem více vazeb; závislosti pak obsahují již jmenované systémové atributy, stejně jako lze také vytvořit atributy vlastní. Vazby lze pak zobrazit v matici závislostí nebo v grafickém zobrazení, který je podobný tomu pro moduly a objekty, pouze zobrazuje také definované závislosti mezi objekty. Nechybí označování cílů vazeb jako kandidátů k revizi v případě, že dojde ke změně v jeho zdroji.

Neobvyklou funkcionalitou tohoto produktu je možnost vytváření a správy testovacích scénářů a běhů testů. DOORS umožňuje vytvořit speciální objekt - definici testu, který má dvě skupiny atributů, jednu příslušející testovacímu scénáři (například nutné počáteční podmínky, předpokládaný výsledek apod.), druhou vázající se k jednotlivým provedením testu (jako například osobu, která test prováděla, datum, kdy se tak stalo, nebo délku trvání testu). Protože test je typicky během vývoje produktu proveden víckrát, systém může uchovávat více kopií atributů druhé skupiny. Běhy testů lze pak zobrazit v seznamu podle nějakého z jejich atributů, například podle výsledku testu (Úspěšný/Neúspěšný/Neznámý). Pro vyspělejší práci s testovacími scénáři je program možné propojit s Telelogic TAU, případně Mercury TestDirector nebo LoadRunner.

Další ne úplně častou možností je uzpůsobovat program v integrovaném skriptovacím jazyce DOORS eXtension Language (DXL), který syntaxí připomíná C/C++. DXL skripty se dají použít například pro komplexnější atributy, jejichž hodnoty je třeba například netriviálním způsobem odvodit z hodnot atributů jiných, nebo dokonce pro přidávání vlastních položek do menu programu.

Produkt zaznamenává podrobnou historii pro mnoho položek databáze: nejen pro objekty a změny hodnot jejich atributů, ale i pro typy atributů a další položky modulů. Historie obsahuje pro všechny provedené změny zodpovědného uživatele, datum a čas změny a jaká změna byla provedena. Pro mnoho změn je k dispozici více údajů, například byla-li změněna hodnota atributu, v historii se lze dokonce dočíst, jaká byla jeho nová i stará hodnota. Pro zvýšení přehlednosti lze data historie filtrovat.

Zobrazování obsahu modulu je definováno tzv. pohledy. Výchozí pohled obsahuje všechny objekty a ke každému jeho vygenerovaný identifikátor a nadpis. Tento pohled lze však velmi flexibilně ovlivňovat: lze nastavit, které atributy se mají v pohledu zobrazit, do jaké hloubky hierarchie se mají objekty zobrazovat nebo jestli se mají zobrazovat pouze objekty s nějakými závislostmi. Objekty lze také velmi podrobně filtrovat podle klíčových slov, které obsahují nebo neobsahují jejich atributy nebo odkazy. Objekty v pohledu lze i řadit podle více atributů. Vytvořené pohledy je možné uložit do aktuálního modulu a to například jako výchozí pohled pro tento modul pro aktuálního uživatele.

Stejně jako konkurenční produkty i DOORS umožňuje zachytit množinu objektů modulu nebo více modulů v určitém časovém okamžiku – tedy vytvořit tzv. baseline. Baseline v DOORS představuje kopii modulu, kterou již nelze dále editovat. Taková kopie modulu obsahuje i jeho kompletní historii. Systém umožňuje základní větev i tzv. podepsat, tedy připojit k ní jméno uživatele, který baseline vytvořil, a také ji označit jedním z nabízených pojmenování (například Všechny změny odsouhlaseny, případně Odmítnuto apod.) a připojit komentář. Dve základní větve lze pak samozřejmě porovnat, DOORS nabízí hned dvě možná zobrazení výsledku takového porovnání: buď ve stylu unixového diffu zobrazující celé změněné řádky atributů nebo podobně jako u revizí editoru Microsoft Word s v textu vyznačenými změnami (nahrazené slovo je zobrazeno přeškrtnuté a červeně, nové pak modře a podtrženě).

DOORS samozřejmě podporuje spolupráci více uživatelů a hierarchii uživatelských práv. Navrhovat modifikace požadavků bez jejich vlastní editace lze prostřednictvím tzv. požadavku na změnu nebo poznámky (návrhu, suggestion). Požadavky na změny mohou být připojeny k jednomu nebo více modulům, lze jim i definovat atributy; po vytvoření pak mohou být slučovány nebo zařazovány do skupin. Návrh může být připojen k projektu, složce nebo modulu. Při jeho vytváření se udává navrhovaná změna a důvody, proč by se měla podle navrhovatele uskutečnit. Uživatelé s příslušnými právy pak mohou požadavky na změny nebo návrhy odsouhlasit či odmítnout a to i více najednou, jsou-li uspořádány do skupin. Navrhovatel je pak upozorněn e-mailem, že se změnil stav jeho žádosti (například na Odsouhlaseno, Revidováno apod.). Máme-li externí dodavatele, kteří přímo vytvářejí specifikaci nějaké části vytvářeného systému, můžeme část DOORS databáze vydělit z centra (tím se tam označí jako pouze pro čtení), exportovat k subdodavateli a pak zase importovat zpět.

Pokud se neurčí jinak, může být jeden modul editován pouze jediným uživatelem, ostatní ho mohou pouze číst. DOORS přehledně barevně zobrazuje, které objekty byly změněny od vytvoření poslední baseline (navíc s rozlišením, jestli již změny byly již v aktuální uživatelské relaci uloženy). Modul lze však i rozdělit na více editovatelných částí, to pak umožní souběžnou práci více uživatelů na objektech jednoho modulu. DOORS dále umožňuje nastavit uživatelská práva prakticky ke každému typu obsahu databáze. Lze nastavit, kteří uživatelé nebo které uživatelské skupiny mohou číst, editovat, vytvářet, mazat nebo měnit uživatelská práva u projektů, složek, modulů, objektů, ale i atributů a jejich typů. Při vytváření nového synovského objektu (například podřízeného modulu) jsou přístupová práva děděna z rodiče, chceme-li tedy změnit přístup k celému stromu položek, není třeba to dělat pro všechny jeho uzly. Dědičnost lze ale také vypnout a nastavit práva pro nějakou položku zvlášť.

Vytváření tištěných sestav je silně založeno na výše popsaných pohledech. Vytvoření takové sestavy v podstatě pouze znamená vytištění pohledu aktuálního, pohledu z toho aktuálního odvozeného nebo vytvořeného dříve. Existují tři základní vzhledy takového dokumentu – v prvním jsou data uspořádány ve stylu dokumentu, výsledné stránky pak vypadají obdobně jako se zobrazuje obsah modulu v programu. Druhý vzhled – nazývaný tabulka – zobrazuje vysoce strukturovaná data všech objektů, třetí se pak nazývá kniha a ten připomíná částečně tabulku, také však požadavky sepsané v textovém editoru, kde pro každý atribut požadavku vytvoří program nový řádek. Kromě očekávatelných možností úprav (velikost papíru, okraje apod.) lze do záhlaví a zápatí vkládat pomocí speciálních značek prefixovaných ampersandem objekty jako jméno modulu, projektu, uživatele nebo například databázovou cestu k modulu. Používáme-li některé sestavy často, můžeme si jejich definice uložit a vytisknout si je tak kdykoliv, bez opětovného procházení průvodce na jejich vytváření. DOORS umožňuje i export pohledů do Microsoft Wordu nebo Excelu, do souboru TXT nebo webové stránky v jazyce HTML. Jednotlivé objekty pak lze exportovat do e-mailové zprávy programu Microsoft Outlook nebo jako část prezentace Microsoft Powerpoint.

Telelogic DOORS je možné integrovat s velkou řadou jiných softwarů, ať už z rodiny produktů pro podporu životního cyklu od Telelogic nebo programů od jiných výrobců. DOORS/Analyst je nadmnožinou DOORS integrující základní systém na správu požadavků s možností vytvářet UML 2.0 modely vytvářeného řešení a prvky těchto modelů navazovat na objekty databáze z DOORS. Používáme-li i Telelogic TAU, modely lze také exportovat do tohoto programu a tam pokročile upravovat a zároveň si udržet stopovatelnost zpět do databáze DOORS. Vystopovatelnost požadavků až na úroveň vytvářeného kódu a hlavně jeho různých verzí je možná, využijeme-li s DOORS i produkt Telelogic Synergy. Podobným způsobem lze DOORS integrovat s produkty rodiny IBM Rational, konkrétně Rose,

ClearCase a ClearQuest.

Program pracuje v režimu klient-server, kde klient i databázový server může běžet na Unixu, Linuxu, Citrixu nebo Windows. Klient existuje i ve webové verzi (tento klient je nazvaný DOORSnet), abychom ho mohli používat, je samozřejmě nutné nainstalovat příslušný webový server. To umožňuje prakticky veškerou funkcionalitu, k dispozici není možnost tisku a ukládání strukturovaných sestav (pouze tisk aktuální stránky – díky funkčnosti pohledů ale můžeme dosáhnout velmi podobného výsledku) a podpora DXL skriptů. DOORSnet umožňuje také přístup ke správě návrhu změn požadavků.

DOORS je vyspělý produkt na správu požadavků schopný podporovat práci mnoha uživatelů s velkým množstvím často se měnících požadavků. Velmi propracovaná je především historie a možnost velmi podrobného nastavování přístupových práv ke všem typům položek databáze. Zajímavou funkcionalitou je integrace správ testovacích scénářů, hodí se ale pravděpodobně spíš pro menší projekty, pro ty větší by byl lepší specializovaný software. Uživatelé nepoužívající operační systém Windows ocení existenci klienta pro unixové systémy, stejně jako cestující uvítají velmi dobře použitelného webového klienta. Poněkud omezené v základní verzi programu (bez rozšíření) se zdají možností závislostí, především jejich grafické zobrazování. Zřejmě se automaticky počítá s použitím dalších programů spolu s DOORS. Samotný DOORS se totiž zdá poněkud izolovaný, spolu s dalšími produkty od Telelogic (případně IBM) se ale stává velmi mocným nástrojem z možnostmi absolutní vystopovatelností požadavků až k jejich implementaci.

7 Specifikace požadavků v SOA

Významnou částí této práce je zhodnocení vlivu, který mají na specifikaci požadavků a prostředků při ní používaných změny, které se v IT v posledních letech odehrávají. Nejvýraznější změnou při budování rozsáhlých softwarových systémů je nástup servisně orientované architektury (SOA), v následujících kapitolách se jí proto budeme podrobně věnovat.

7.1 SOA obecně

Abychom mohli zhodnotit vliv SOA na specifikaci požadavků a úvodní fáze životního cyklu softwaru obecně, nejdříve se musíme seznámit se servisně orientovanou architekturou jako takovou. Následující odstavce obsahují motivaci, na nichž vývoj této architektury stojí, definice SOA a její možné přínosy.

7.1.1 Situace

S rozvojem informačních technologií a jejich pronikáním do všech oblastí podnikání přichází kromě usnadnění práce také řada komplikací. Jak se rozvíjejí potřeby firmy, vzrůstá i touha řešit tyto potřeby počítačovým systémem. Tento požadavek ale nevznikne najednou, nepočítačové zpracování může stačit dlouhou dobu předtím, než se vedení firmy rozhodne, že nasazení softwaru přinese zjednodušení a snížení nákladů. Společnost také může již nějaký systém používat, management však například dospěje k rozhodnutí, že stávající funkcionalita nepokrývá současné potřeby dostatečně a že je třeba software obnovit. Nemusí ovšem najít nové řešení stejného rozsahu, nahrazena tedy může být v daném okamžiku pouze část, další mohou pak (později) jiným produktem. Ve firmě je tedy postupně nasazováno mnoho různorodých informačních systémů, které typicky nejsou navrhovány s tím, že budou vzájemně komunikovat. Propojení bývá velmi dobře možné mezi produkty stejného výrobce, vazby různorodější skupiny softwarů jsou ale často realizovány ad-hoc exporty nebo proprietárními rozhraními, které je třeba modifikovat při jen malé změně některého z propojovaných systémů. Integrovat všechny produkty bez přizvání zkušené konzultantské

firmy je prakticky nemožné [27, 65]; i s poradci ale často vede k novým velkým projektům, které sice realizují integraci, komplexitu architektury ale nesnižují.

I když má firma perfektně konsolidovanou informační politiku, stále ještě musí komunikovat se svým okolím, s jinými firmami (například dceřinými) a se zákazníky [35, 2]. Stále častěji se také některé firemní úkony outsourcují, časté jsou i akvizice jiných firem [35, 7]. To opět přináší další informační systémy, které je třeba ve firmě zkoordinovat s těmi aktuálními. Takové neustálé změny udržují potřebu integrace na konstantní úrovni. Kdyby firma chtěla udržet portfolio systémů přehledné a vzájemně komunikující, musela by platit integrátora každý rok. Naopak chce-li firma prodat jednu ze svých divizí, nemůže jí prodat s proprietárním softwarem, který používá celá společnost, protože si chce toto své intelektuální vlastnictví chránit. Obtížná možnost takový systém rozdělit může být příčinou neuskutečnění obchodu nebo značného zvýšení nákladů na jeho realizaci.

Dále, s rostoucí globalizací roste počet nadnárodních firem [35, 6]. Nároky na informační systémy poboček takových společností se mohou radikálně lišit. Příčinou může být rozdílná legislativa [35, 7], poněkud odlišná koncentrace obchodu podle kultury a stavu ekonomiky dané země apod. Ačkoliv tedy patříc pod jednu společnost, jednotlivé národní filiálky se velmi liší v požadavcích na své systémy. Software pro dvě pobočky v různých zemích mohou vyvíjet různé firmy (například právě z důvodu znalosti místního práva), což je další důvod, proč takové systémy budou rozdílné. Centrála ale chce mít přehled o datech z obou systémů zároveň. A vybuduje-li se takový „dohledový“ systém, chtěli bychom, aby se do něj nemuselo zasahovat pokaždé, když se změní nebo dokonce vymění dohledované systémy, aby dohled byl realizován automaticky, bez nutnosti zásahu člověka (např. exportujícího data z lokálního systému) a aby přidání nebo odebrání lokálního systému nemělo zásadní vliv na ten nadřazený.

Typickým příkladem heterogenního informačního prostředí je státní správa. Už z podstaty jejich fungování jsou softwarové systémy na každém úřadě poskytujícím různé služby rozdílné. Je ale žádoucí, aby tyto systémy komunikovaly, aby sdílely data. Například by bylo příjemné, kdyby Živnostenský úřad automaticky informoval Finanční úřad nebo Správu sociálního zabezpečení o změnách osobních údajů živnostníka. Je ale nemožné vyvinout jakýsi univerzální modulární systém, který by byl pak použitelný na všech úřadech bez rozdílů [36, 2]. Takový projekt by byl velmi drahý, časově náročný, obtížně koordinovatelný [35, 7] a daleko před jeho dokončením by se požadavky jednotlivých subjektů natolik proměnily, že by se mohlo začít znovu.

Ze stejného soudku je i informační architektura v rámci velké university. Bylo by příjemné přidat nového studenta do jednoho systému a poté o něm vědět všude, kde je tato informace potřeba bez nutnosti přepsaní všech takových systémů tak, že by používaly stejnou databázi.

Nakoupit „univerzální“ rodinu softwarových produktů od velké firmy se zdá být způsobem, jak snížit komplexitu portfolia informačních systémů, předpokládajíc, že jednotlivé subprodukty rodiny jsou prokazatelně a snadno propojitelné. V případě rozšiřujících se potřeb se jednoduše nakoupí nový prvek rodiny a napojení na stávající používané systémy je nepříliš náročná operace. Tento postup ale řeší jen část problémů zmíněných výše, pokud nějakou funkci outsourcujeme nebo nakoupíme jinou firmu (případně se s nějakou spojíme) a partnerská společnost nepoužívá software od stejné firmy, opět vyvstává palčivá otázka integrace. Problém je i s oddělováním divizí v případě jejich prodeje. Zcela nevyřešená pak zůstává otázka komunikace s ostatními subjekty, jejich informační systémy totiž nemáme typicky možnost příliš ovlivnit.

7.1.2 Definice SOA

V tomto kontextu se nadějnou myšlenkou zdá idea servisní orientace a na tomto principu postavené architektury (architektura orientovaná na služby, Service Oriented Architecture, SOA). Servisní orientace reprezentuje specifický přístup k dekompozici množiny velkých problémů na větší množství menších úkolů. Tento přístup tradičně přináší lépe promyšlená řešení, jejich snadnější implementaci a manažerské řízení. Přidaná hodnota myšlenky servisní orientace je ve způsobu, jakým jsou jednotlivé části řešení odděleny [27, 32]. Tyto komponenty fungují jako služby v reálném životě, což umožňuje to, že vazba mezi nimi a okolím je volná a nenáročná. Erl uvádí příklad města, které je plné firem orientovaných na služby, krátce řečeno v běžné řeči: plné služeb. Každá služba nabízí vykonání definované činnosti libovolnému zákazníkovi, jímž může být i jiná firma poskytující službu odlišného charakteru. Taková firma však není pevně svázaná s tímto dodavatelem, typicky má možnost počátečního výběru i pozdější změny. Společnosti jsou tedy soběstačné, zároveň však využívají jiné služby [27, 32-33].

Aplikováno do prostředí softwarové architektury, můžeme říct, že SOA reprezentuje prostředí více menších vzájemně komunikujících systémů. Protože tyto systémy fungují jako služby v reálném životě, jejich rozhraní je přesně definováno, není platformně závislé ani orientované, díky čemuž jsou tyto služby znovupoužitelné [26]. Jednotlivé části architektury vykazují vysokou míru soběstačnosti (jejich stav nezávisí na stavu ostatních služeb [43, 1]), mohou být vyvíjeny nezávisle na svém okolí, přesto však nejsou izolovány, ba co víc, s ostatními systémy tvoří middlewarem vzájemně propojený „ekosystém“. Říkáme, že tyto systémy jsou vázané volně (loosely coupled).

Z předchozích podmínek plyne, že komponenty SOA jsou *navrženy* proto, aby byly používány zvenčí bez znalosti vnitřního fungování, jako černé skřínky [35, 2]. Ačkoliv každá služba samozřejmě má svojí vnitřní logiku, pro ostatní systémy v rámci SOA jsou tato specifika nedůležitá, je-li dodržena podmínka jasně definovaného rozhraní, které mají okolní služby k dispozici [27, 32-33].

Důležitou podmínkou také je, že rozhraní jsou orientovaná na obchodní potřeby spíše než technicky. Erl uvádí, jak se může zefektivnit automatizace obchodních procesů firmy, když se celé procesy nebo vzájemně související kroky procesů (podprocesy) přetvoří ve služby – je poté možné takovou činnost znovupoužít v jiné (např. nově vytvořené) činnosti a také tyto služby zefektivňovat (obecněji: jakkoliv uvnitř měnit) bez ovlivňování celku, protože ostatní služby vyžadují jen předem domluvené (obchodně orientované) rozhraní [27, 34]. Procesy mohou být také snadno slučovány a rozšiřovány.

Podle Erla klíčovými aspekty architektury orientované na služby jsou [27, 37]:

- Volná vazba – Služby udržují vztah bez velkého množství závislostí a vyžadují pouze schopnost mít povědomí o vzájemné existenci.
- Dohoda o službě – Služby odpovídají dohodnutému komunikačnímu rozhraní, které bylo definováno v jednom nebo více popisech služby.
- Samostatnost – Služby mají kontrolu nad vnitřní logikou, kterou zapouzdřují.
- Abstrakce – Vše přesahující dohodnutému rozhraní je vnějšímu světu skryto. Je navíc důležité, aby rozhraní bylo orientováno uživatelsky – tj. aby netechnickým způsobem nezávislým na implementaci zachycovalo obchodní funkce dané služby [11].
- Znovupoužitelnost – Funkcionalita je rozdělena mezi služby za účelem lepšího opětovného použití.
- Kombinovatelnost - Více služeb může být zkombinováno za účelem vytvoření sdružené služby.
- Bezestavovost – Služby minimalizují množství uložených informací specifických pro jednu činnost.
- Objevitelnost – Služby a architektura jsou navrženy tak, že jsou navenek natolik

popisné, aby bylo možné je nalézt a přistoupit k nim za použití dostupných prostředků zpřístupnění.

Je možno si povšimnout, že díky kombinovatelnosti služeb je častou vlastností SOA znovupoužitelnost na více úrovních abstrakce [27, 47]. Budujeme-li již části obchodních procesů jako služby, z nichž pak skládáme samotné procesy a z nich pak aplikace a systémy, znovupoužitelnost se objevuje na všech úrovních této hierarchie.

Myšlenka servisně orientované architektury není nijak nová, její prvky lze vysledovat již v systémech psaných v Cobolu od konce šedesátých let a v operačních systémech mainframů a minipočítačů při obsluze periferií [11]. Král uvádí tyto příčiny, proč se tato idea prosazuje naplno až nyní ([37, 6] a [11]):

- Až dosud hlavním úkolem bylo vyvinout spolehlivé aplikace. Komunikace mezi nimi nebyla závažným problémem.
- Dříve nebyla k dispozici dostatečně efektivní technologie poskytující žádoucí funkce (middleware) umožňující efektivní implementaci SOA.
- V minulosti používané informační systémy nebyly natolik složité, aby nemohly být přepsány v případě potřeby od začátku.
- Servisní orientace vyžaduje změnu obchodních strategií a výrobních postupů softwarových firem.
- Dříve nebyli dostatečně připraveni vývojáři (ti jsou školeni spíše pro vývoj monolitických aplikací, typicky objektově orientovaných). Pro většinu vývojářů se jedná o zcela nové paradigma s novými přístupy a praktickými dovednostmi Servisní orientace se liší od objektové, dokonce existují případy, kdy anti-vzory v objektově orientované metodice jsou pozitivní vzory v metodice servisně orientované, např. tzv. Ostrůvková automatizace (Island Automation).
- Rostou nároky na rychlost reorganizací, např. na začlenění/vyčlenění jednotek do/z organizace.
- Prosazují se nové prvky v ekonomice - např. nákupní koalice nezávislých výrobců, CRM systémy (Customer Relationship Management, systémy pro správu údajů o zákaznících a komunikaci s nimi), SCM (Supply Chain Management, spolupráce se subdodavateli), *e*-komerce, *e*-government, globální podniky atd.
- Existují i důvody obchodně technické, jako je potřeba snížit závislost na dodavatelích a technologiích, protože často mizí z trhu i produkty velkých firem, na nichž mnohdy závisí chod organizací. Je-li systém složen z relativně nezávislých spolupracujících částí a přestane-li být některá z těchto částí podporována výrobcem, je snazší nahradit tuto část jiným systémem.
- Servisní orientace nestimuluje k sofistikovaným teoriím a není proto příliš zajímavá pro akademickou obec.
- Vývoj SOA systémů vyžaduje úzkou spolupráci s uživateli, což není příznivé pro mnoho absolventů studií informatiky.

Konfederace a aliance

Král a Žemlička ukazují, že požadavky na těsnost vazby servisně orientovaných systémů se v různých kontextech liší [36, 1-2]. Zatímco propojení informačních systémů divizí globální korporace může být poměrně těsné (ačkoliv stále dostatečně volné, aby umožnilo např. integraci nové divize nebo odprodej některé stávající; paralelní vývoj a podobně), síť dynamicky kooperujících subjektů obchodujících přes Internet musí být provázána velmi volně. Mezi těmito dvěma extrémy pak existují případy podobné více či

méně jednomu z nich. Král s Žemličkou nazývají ty pevněji provázané konfederace, ty s volnějšími vazbami pak aliance (např. [37, 1]). Typickou vlastností aliance je, že subjekty v ní se navzájem neznají a hledají se (typicky kdekoliv v Internetu) až v ve chvíli, kdy to situace vyžaduje. V aliancích je proto nutné používat široce uznávané standardy jako jsou například SOAP a WSDL.

Konfederace se naopak mění pomaleji, komponenty se znají velmi dobře, návrháři tedy mají více možností, jak taková seskupení realizovat. Příklady konfederací jsou státní informační systém, IS globální popř. decentralizované firmy nebo nově vytvořené systémy spojené do servisně orientované architektury [37, 1].

Příklady a dopady

Buduje-li se software podle těchto principů, dopady mnoha problémů s komplexitou a propojováním heterogenních systémů se zdají být zmírněny. Protože systémy mají známé (a často i standardní) rozhraní, je snadné je spojit dohromady v komplexnější systém. Tento vzniklý systém je ale daleko flexibilnější, protože je snadné vyměnit nějakou jeho část, neboť to minimálně ovlivní ostatní prvky prostředí; je třeba jen poskytnout rozhraní zpětně kompatibilní s tím předchozím, což je při jeho rozumném návrhu poměrně snadné [27, 43].

Pro lepší představu problému si představme fúzi obřích firem jako jsou Oracle a Peoplesoft nebo Logica a CMG. Takové firmy se dlouhá léta vyvíjely samostatně, poté se však sloučily a začaly společně pracovat a organizovat se. Pokud měly obě firmy velký proprietární monolitický informační systém, náklady na propojení zpracování informací musely být enormní. Pokud by ale obě firmy měly mnoho menších systémů v roli služeb, obtížnost by výrazně klesla [27, 60] a [36, 8]. Kdyby například k softwaru na správu požadavků v CMG bylo možné přistupovat zveřejněným standardním protokolem (např. SOAP), bylo by snadné extrahovat z něj data pro vykazování nákladů v aplikaci používané v celé LogiceCMG. Taková aplikace by pak mohla být lehký webový klient umožňující přehledovat data z různých systémů více firem.

Během autorovy praxe ve firmě Bosch se tato firma spojila s jednou z částí firmy Philips – s její divizí Philips Security Systems. Během krátkého časového období bylo vyděleno mnoho týmů z národních poboček Philipsu a připojeno k existujícím týmům v Boschi. Bosch v té době používal ElanorGlobal jako svůj HR systém, SAP pro ERP, Microsoft Exchange pro elektronickou komunikaci a spolupráci a více jiných proprietárních systémů. Připojivší se divize nepoužívala ani jeden stejný systém pro stejnou věc. Spojení znamenalo množství exportů a konverze dat, přeškolování uživatelů a jejich následnou nespokojenost. Proč by ale obě skupiny nemohly používat nadále své systémy (i když se menší tým spojoval s obřím)? Mnoho uživatelů potřebovalo ke své práci pouze data své divize, nepotřebovalo vstupovat do nového „centrálního“ systému a tam si filtrovat data pro ně relevantní (tj. data jejich divize). Ostatně i v rámci původní pražské pobočky Bosche bylo několik vzájemně příliš nesouvisejících podskupin: Elektrické nářadí, Domácí spotřebiče, Blaupunkt, Tenovis a další. Pro velkou většinu pracovníků nebyla data ostatních divizí zajímavá, naopak každá „subfirma“ měla odlišné potřeby, například již proto, že si sama určovala obchodní strategii. Dvě z divizí mající hodně obchodních cestujících používaly software Soft-4-Sale pro jejich koordinaci, ne však ostatní. Ovšem Bosch je tradiční globální firma a strategickým rozhodnutím z centra se přešlo na SAP. Software na koordinaci obchodních cestujících se ale používal nadále, protože SAP potřebnou funkcionalitu neposkytoval (popř. nebyla zakoupena). Servis ručního nářadí používal ještě tři roky po přechodu prvních částí pražské pobočky starý proprietární systém BD-Soft, transformace na nový systém byl bolestivý a zdlouhavý proces, protože velká část funkcionality BD-Softu byla specializována právě na servisy.

Co by přineslo, kdyby firma používala více drobnějších systémů navržených jako

vzájemně se využívající služby? Umožnilo by to lepší a detailnější zachycení odlišností divizí v informační architektuře, systémy by mohly být lépe vázány na jejich aktuální organizační strukturu [36, 8]. Mnoho procesů a požadavků by bylo společné pro více divizí, případně by se jen pracovalo nad jinými daty, velká část nároků by ale byla specifická buď pro daný typ divize (všechny divize Blaupunkt na světě) nebo pro danou jednu konkrétní. Systémy by byly jednodušší a pokrývaly by přesně potřeby dané skupiny v daném čase. Například místo komplexního tradičního SAPu, který má ohromné množství funkcí (protože je to velmi obecný systém snažící se vyjít vstříc mnoha potřebám), by stačily drobné jednodušší produkty vzájemně volně propojitelné podle aktuální potřeby. Při restrukturalizaci by se tak pouze změnily vazby mezi systémy, ne systémy samotné (alespoň ne zásadně, viz níže). Při větší granularitě by se pak dokonce části obchodních procesů využívaly všemi, kteří danou část provádějí stejně.

Architektura orientovaná na služby přináší mnoho výhod i z hlediska softwarového inženýrství. Činnosti jako modelování, inkrementální vývoj, otevřenost, testovatelnost a další jsou v SOA velmi snadno implementovatelné a použitelné [36, 9].

Pokud vytváříme softwarovou architekturu jako složenou z menších produktů poskytujících jasná rozhraní, umožňuje nám ji více přizpůsobit našim potřebám, méně se musí naše potřeby přizpůsobovat jí. Zatímco při tvorbě monolitického systému se snažíme vměstnat naše procesy a požadavky do rozumně udržovatelného programu a musíme neustále mít na zřeteli technická omezení; pokud uvažujeme o funkcích informačních systémů jako o prvcích stavebnice nebo jako o službách v reálném světě, můžeme se koncentrovat na naše obchodní potřeby a získat potřebnou abstrakci nad implementací a technickými detaily [27, 49]. Pokud měla divize Bosche specifický požadavek, bylo velmi obtížné a drahé ho do SAPu zavést, do vlastních malých systémů by to nebyl velký problém.

Pokud vystane potřeba integrovat stávající systémy, v zásadě jsou dvě krajní možnosti (přičemž vzniklé řešení může mít aspekty obou): vytvořit nový systém nebo propojit ty stávající. Myšlenka SOA odkrývá mnoho výhod propojování více systémů oproti implementaci jednoho monolitického (snažícího se řešit vše). Vývoj softwaru v rámci SOA může být daleko flexibilnější, protože jelikož mají produkty na sebe pouze volnou vazbu, vývoj může probíhat relativně odděleně a tak jak co do komunikace tak do času. Velký systém je často obtížné sestavit do funkčního celku, příliš mnoho zúčastněných přináší nedorozumění a rozpory do formulovaných požadavků. V SOA ale můžeme mít menší týmy produkující software na téměř libovolné platformě [27, 43]. Samozřejmě uživatel daného systému je nadále důležitý a je třeba s ním komunikovat neméně intenzivně jako dříve, systém je ovšem zaměřen na menší uživatelskou skupinu, získávání požadavků je tedy snazší a výsledný systém bude typicky jeho uživatelům vyhovovat více [27, 61]. Navíc veškeré starosti o zasazení produktu do širšího kontextu (architektury) řeší dohodnuté rozhraní; tyto otázky neovlivňují vnitřní fungování systému. Vlastně se tak vracíme do fáze, kdy firma měla mnoho malých systémů a vývojáři se nestarali o jejich propojování, jen díky odlišnému myšlení na úrovni architektury a implementaci jasných obchodně orientovaných rozhraní jejích prvků je propojování nejen snadno realizovatelné, ale je přímo charakteristickým rysem našeho řešení.

Klíčovou výhodou pro firmy všech velikostí je možnost využít stávající aplikace [27, 45 a 62]. Pokud se ke stávajícím systémům pouze doprogramují brány sloužící k jejich zapojení do „nové“ infrastruktury (takové brány překládají proprietární výstupy do námi specifikovaného formátu servisně orientovaného rozhraní [35]), není třeba přepisovat něco, co dobře funguje a čehož jediná nevýhoda je nemožnost komunikovat s okolím. Nebudujeme-li nový systém, není třeba přeškolovat uživatele ani správce, nemusíme shánět nové vývojáře, můžeme také zůstat u stávajícího hardware. A pokud se za pár let takový starší systém ukáže jako nevyhovující, nebude obtížné díky jasně definovanému rozhraní vyvinout náhradu, která

bude z pohledu ostatních aplikací a celkové architektury plnit kompatibilní funkci.

Stejně i použití externích řešení pro část informačního systému není zásadní problém. Zde je důležité, že můžeme outsourcovat pouze podmnožinu systémů, ne vše nebo nic [36, 8]. Snažší je také prodej částí systémů, neboť tyto jsou soběstačné a jejich vydělení z celkové informační architektury je snadné [36, 8].

Rychlost reakce na změny je v dnešní ekonomice velmi důležitá. Požadavky na informační systémy se tedy mění každým měsícem a čím déle jeho vývoj trvá, tím větší je pravděpodobnost, že bude zastaralý ještě před dokončením. Jak bylo naznačeno výše, založení informační politiky na portfoliu menších aplikací dává lepší možnost podchycovat změny snadněji a tudíž dříve [27, 51]. Během provozu je úprava jednodušší a bezpečnější, ovlivní-li jenom ty části systému, kde je změna potřeba, a nijak se to nedotkne celkové architektury [27, 64].

Další výhodou, kterou SOA přináší, je snazší a flexibilnější údržba a správa aplikací. Pokud má firma globální monolitický systém, je velmi složité ho spravovat, již jen z příčiny, že je těžké dohledat, kdo je za aktuální chybu zodpovědný. Protože zatímco systém je globální a (teoreticky) všude stejný, správa a údržba velmi často globální není. Jsou-li ale rozhraní v souladu s obchodní logikou („jsou orientována uživatelsky“), je snadné i pro netechnického pracovníka určit, jaký systém vykázal nefunkčnost. Díky distributivitě takové architektury není pak téměř žádná chyba fatální, vyřadí pouze jeden uzel, ostatní služby a eventuálně i systém jako celek zůstává dostupný.

Za nevýhodu SOA by mohly být považovány zvýšené nároky na hardwarový výkon. Tím, že systémy nebudujeme jako monolitické celky, ale jako víc vzájemně komunikujících entit vlastně přidáváme další vrstvy do zpracování informací a tím i zvyšujeme nároky na výpočetní výkon serverů a kapacitu spojů [27, 67]. Tento argument ale nemusí uspět vždy. Bude efektivnější velký nepřehledný systém nebo několik menších dobře vyladěných systémů propojených dohromady? Také je třeba podotknout, že monolitický systém může být obtížné umístit na více serverů a zajistit tak paralelizaci zpracování, tuto otázku ale pravděpodobně tvůrci velkých informačních systémů mají na paměti. Faktem ale zůstává, že hardware je levný jako nikdy v historii (a bude nejspíše jen levnější) a daleko významnější položkou v nákladech je lidská práce (jejíž cena spíše klesat nebude). Náklady na pracovníky používající, udržující a modifikující složité informační systémy jsou tedy primárními zdroji snížení nákladů.

Obtížná může být praktická realizace myšlenek SOA. Protože je to koncept ovlivňující fungování celé firmy (či dokonce více subjektů), náročná je implementace především z hlediska získávání informací, jejich transformace v požadavky na architekturu, jejich správu a uvedení do praxe. Níže v této práci bude ukázáno, jak lze tyto problémy zmírnit.

7.1.3 Obchodní procesy v SOA

Velmi výhodné pro zefektivnění fungování firem může být mít počítačovou podporu již samotných obchodních procesů servisně orientovanou. Jak bylo zmíněno, je-li tato podpora implementována jako prostředí vzájemně komunikujících služeb, je snadné procesy kombinovat a vytvořit tak sdružený proces. Také není náročné změnit fungování stávajících činností, protože díky přesně definovanému obchodně orientovanému rozhraní je bezproblémové zapojit nové sub-procesy do těch stávajících, aniž by to ovlivnilo jiné procesy na nich postavené.

Podíváme-li se na problém z druhé strany, ze strany našich požadavků na slučování a komunikaci obchodních procesů, dobrou představu nám může dát příklad fiktivní společnosti z Erlovy knihy o SOA. Tato společnost má proces Vytvoření objednávky skládající se ze subprocesů Získání požadovaného zboží, Kontrola dostupnosti ve skladu, Vytvoření objednávky, Zveřejnění objednávky klientovi. Tento proces je možné zkombinovat s

procesem Vytvoření faktury a vytvořit tak novou činnost Zpracování objednávky. Stejně tak je možné proces Zpracování objednávky rozšířit o získání zákazníkovy fakturační adresy. Tento proces může již existovat jako součást samostatného procesu Získání kontaktních udajů zákazníka.

Erl uvádí, že aby takový model mohl být implementován, je nutné splnit následující podmínky:

- Schopnost počítačové podpory obchodních procesů být rozdělena na skupinu jasně definovaných služeb.
- Schopnost těchto částí podpory být vzájemně poměrně nezávislé, aby se daly různě komponovat.
- Schopnost těchto částí vzájemně komunikovat takovým způsobem, že je zachována jejich vlastní nezávislost.

Král a Žemlička poznamenávají, že kompletní změna obchodních procesů je složitá a nebezpečná. Dále že je dobrou praxí, že každý proces má svého vlastníka zodpovědného za dané obchodní záležitosti. Každý proces by také měl být snadno modifikovatelný jeho vlastníkem a to jak jeho model („šablona“), tak také instance procesu v průběhu jeho provádění (nezbytné ve chvílích, kdy je třeba vyřešit neočekávané události – chyba dodávky apod.). Tento požadavek je významný převážně v menších a středních firmách [37, 2].

Krátce řečeno, Erl požaduje zapouzdření částí počítačové podpory procesů, jejich volnou vazbu a komunikaci založenou na servisní orientaci, což odpovídá definici SOA, jak byla uvedena výše [27, 39]. Podle Krále pak požadavek na modifikovatelnost procesu jeho vlastníkem implikuje, že rozhraní služeb procesy implementující musí být čitelné pro takové pracovníky – tedy běžné uživatele. V [37, 3] pak ukazuje, jak je možné podporu obchodních procesů v SOA implementovat a že je to obecně snadnější v prostředí konfederací než aliancí.

ERP/SOA Resource Center americké armády nedoporučuje procesy založené na transakcích implementovat v SOA, protože ty jsou podle něj příliš tíživé pro její technickou infrastrukturu. Naopak jako dobří kandidáti jsou uvedeny prověřovací a řídicí procesy [43].

7.1.4 ERP

S rostoucím počtem informačních systémů ve firmách vznikl v devadesátých letech koncept jejich integrace pod názvem ERP. Samotná zkratka nemá s významem pojmu mnoho společného, vychází spíše z historie a znamená Enterprise Resource Planning, plánování firemních prostředků. Naproti tomu ERP zahrnuje všechny klíčové systémy firmy jako jsou finanční, personální, výrobní, skladové, někdy též plánovací a CRM [42, 1].

ERP se vyvíjelo od monolitických systémů pokrývajících všechny zmíněné oblasti k systémům složeným z komponent, kdy si zákazník mohl vybrat jen ty části, které skutečně potřeboval. Nyní je i v této oblasti snaha přejít k pružnější architektuře, kdy jednotlivé firemní systémy jsou sice propojovány, ale zároveň si uchovávají dostatečnou autonomitu – tedy k myšlence servisně orientované architektury.

SOA nenahrazuje ERP, ale poskytuje myšlenkové paradigma, které pomůže tradičně pevně vázané komponenty ERP systémů učinit samostatnějšími (volně vázanými) se všemi výhodami, které to přináší. Nicméně Koch uvádí, že SOA ještě stále není dostatečně vyzrálá a existují důvody, proč by mohlo být pro firmu výhodnější (prozatím) monolitické řešení [42, 4]:

- Existují spíše běžné procesy zahrnující všechny divize.
- Používají se staré systémy, které potřebují být obnoveny.
- Je přítomno více ERP instancí od stejného výrobce.

Naopak ve prospěch SOA hovoří následující situace:

- Existují divize se specifickými procesy, které nemohou být změněny.
- Firma hodnotí investici do nejlepšího systému v určité kategorii jako konkurenční výhodu.
- Je vyžadováno prostředí, do kterého je snadné začlenit nové aplikace.

K druhému bodu je třeba podotknout, že koupením ERP řešení od jednoho výrobce málokdy získáme to nejlepší na trhu ve všech oblastech. Takový software lze zpravidla pořídit jen od specialistů v daném segmentu.

7.1.5 Implementace

Termín SOA je často zaměňován s webovými službami (tj. XML a specifikacemi SOAP, WSDL a UDDI), je však třeba zdůraznit, že tyto standardy představují jen jednu z možností, jak systémy vytvořené podle principu orientace na služby implementovat [36, 7]. Tyto standardy jsou široce přijímané a umožňují vytvořit dobře znovupoužitelná rozhraní, mají ale i své nedostatky, kvůli kterým se mohou architekti rozhodnout je nepoužít. Neznamená to ale, že takto vytvořená architektura by odporovala principům SOA.

Standardy webových služeb

Jak bylo již poznamenáno výše, SOA nemusí být vždy implementována pomocí webových služeb. Jejich standardy jsou ale velmi rozšířené a podporované většinou firem aktivních v oblasti servisní orientace. Webové služby v základní verzi publikované konsorciem W3 v letech 2000 a 2001 se skládají ze tří standardů: SOAP, WSDL a UDDI ([38], [39] a [40]). Všechny tři protokoly jsou založeny na XML a ačkoliv teoreticky nezávislé na přenosovém kanále, nejčastěji jsou používány s „internetovým“ protokolem HTTP.

SOAP (Simple Object Access Protocol, protokol pro snadný přístup k objektům) definuje formát zpráv, které služby mohou použít pro vzájemnou komunikaci. Jak plyne z názvu, protokol by navržen pro vzdálené volání metod objektů. Od verze 1.2 však byl kromě zpráv ve stylu RPC (zprávy obsahující identifikaci volaného objektu, jeho metody a parametrů) povoleny i dokumentově orientované zprávy, což znamená že tělo SOAP zprávy může být libovolné XML ([27, 74] a [41, 1]). Vedlejším efektem tohoto rozšíření je, že jméno protokolu již není považováno za zkratku, ale je chápáno jako samostatný termín

Aby služby vědely, jaké SOAP zprávy mají pro vzdálené systémy konstruovat, kde je nalézt a jaké objekty (a jejich metody s parametry a návratovými hodnotami) jsou na nich k dispozici, vznikl protokol WSDL (Web Service Description Language, jazyk popisující webovou službu). Pomocí WSDL služby mohou vytvářet své popisy ve formátu XML dokumentu tak, že tento dokument je vše, co vzdálená strana potřebuje k zahájení komunikace.

UDDI (Universal Description, Discovery, and Integration) pak představuje návod na tvorbu úložišť popisů služeb - obdob žlutých (resp. zlatých) stránek. Pomocí nich se dají vyhledat popisy služeb (nejčastěji ve formě WSDL dokumentů) podle různých kritérií. Oproti standardům WSDL a SOAP se UDDI zatím nedočkalo velkého rozšíření. Erl jej uvádí jako volitelné rozšíření SOA [27, 74].

SOAP v základní verzi nenabízí vše, co je třeba pro kritické korporátní aplikace, například nemá zabudovanou žádnou podporu bezpečnosti, transakčního zpracování, spolehlivosti apod. Proto vznikají další tzv. WS-* standardy, které mají tyto nedostatky do webových služeb doplnit. Těchto standardů (přesněji řečeno doporučení organizací jako W3C, OASIS, WS-I či Liberty Alliance) vzniklo velké množství. Z pohledu IBM každý z nich spadá do jedné z kategorií Transports (přenos), Messaging (zprávy, obsahuje kromě SOAP například WS-Addressing specifikující jednotný formát adres služeb), Description and

Discovery (popis a objevování, kromě UDDI a WSDL obsahuje např. WS-Policy rozšiřující možnosti WSDL pro popis účelů služeb a požadavků na komunikaci s nimi), Reliability (spolehlivost, WS-ReliableMessaging zajišťuje spolehlivé doručení všech zpráv), Transactions (transakce, například WS-BusinessActivity slouží pro zajištění integrity rozsáhlých distribuovaných obchodních komunikacích), Security (bezpečnost, obsahuje WS-Federation pro distribuci oprávnění), Business processes (obchodní procesy, WS-BPEL slouží pro modelování obchodních procesů založených na webových službách) a Management (správa, WS-Manageability mluví o identifikaci jednotlivých prostředků poskytujících služby, jejich konfiguraci, stavu apod.) [28].

Ačkoliv těchto doplňujících standardů je velmi mnoho, jsou velmi komplexní, často pro vývojáře obtížně pochopitelné a dokonce se i svým účelem překrývají [29], existence minimálně některých je ale nezbytná pro potřeby přetváření tradičních korporátních systémů v soustavu webových služeb.

Jiné přístupy

Ačkoliv se při vyslovení termínu webové služby vybaví především tři hlavní standardy popsané výše, systém s platformně nezávislým rozhraním přístupným z Internetu lze postavit i na jiných protokolech. V obecném smyslu slova jsou pak i tyto systémy webové služby. Tyto přístupy se využívají především v prostředích, kde se množství WS-* specifikací zdá jako příliš komplikované řešení [30]. Následující odstavce představují dvě nejčastější alternativní metody.

XML-RPC

Jak již název napovídá, XML-RPC je specifikace realizující volání vzdálených procedur, kde data přenášená mezi aplikacemi jsou formátovány jako XML. Volání se realizují přes protokol HTTP (přes POST žádost), což umožňuje komunikovat aplikacím takřka všech platforem. Cílem XML-RPC je být co nejjednodušší, zároveň však umožnit pracovat s komplexními datovými strukturami. Formát zpráv XML-RPC je velmi přímočarý, XML strom požadavku obsahuje jméno metody a její parametry, které mohou být jednoho ze šesti základních typů nebo struktury či pole složené z těchto elementárních typů. Odpověď pak obsahuje pouze návratové hodnoty metody, případně popis nastalé chyby [31].

Jednoduchost jako výhoda XML-RPC je zároveň jeho nevýhodou, pro komplexnější problémy nedostačuje. Nelze například pojmenovávat složené typy nebo v odpovědi rozlišit, návratovou hodnotu jaké metody vlastně obsahuje (požadavky je tedy třeba zpracovávat sekvenčně). Nevýhodou je také nepřítomnost strojově čitelného standardu pro popis schopností služeb – tedy obdobu WSDL pro webové služby. (Teoreticky lze použít WSDL i pro XML-RPC, generátory a wrappery by se ale musely napsat nové speciálně pro XML-RPC. Také by se tím snížila jednoduchost celého konceptu XML-RPC). XML-RPC je tedy dobře použitelné v prostředí málo se měnících služeb a v situaci, kdy se zřídka používají služby nové – dříve nepoužívané, tj. především uvnitř softwarových konfederací.

REST

Representational State Transfer (přenos reprezentace stavu) neboli REST, prvně pulblikovaný v dizertační práci Joye Fieldinga, představuje styl webové archtektury umožňující specifický způsob, jak přenášet různou reprezentaci zdrojů (resources) představujících libovolný dokument v prostředí webu. Obdržená reprezentace pak změní stav příjemce, který na základě této změny může vykonat další akci – vyžádat si nějakou reprezentaci jiného zdroje [32, 5.2]. REST používá HTTP, reprezentace je tedy známý typ obsahu (media type, content type), například text/html, image/jpeg apod. Podstatné je, že každý zdroj, který chceme klientům zpřístupnit, má svoji jedinečnou adresu (klasické URL).

Budování webových služeb v souladu s konceptem RESTu tedy představuje vytvoření URL pro každý zdroj, který zpřístupňujeme. Toto URL může být pouze logické, tj. chápeme-li požadavek jako volání procedury, v adrese zdroje vlastně dekódujeme její jméno (např. URL /orders/25 je ekvivalentní volání procedury getOrder(25), REST používá v adresách podstatná jména namísto sloves, protože URL reprezentují jednotlivé zdroje, ne operace). Po odeslání GET požadavku na toto URL server odešle ve vybraném formátu zpět data (v případě webových služeb tedy typicky v XML). Pokud chce klient upravit stav nějakého zdroje, pošle na určené URL domluvený (opět typicky XML) dokument v HTTP požadavku typu POST [33].

REST lze použít jako snadno pochopitelný a implementovatelný způsob, jak budovat webové služby. Nevýhodou je opět neexistence standardního popisu možností webových služeb – kromě formátu nabízených reprezentací a především adres. Opět lze s výhodou použít pouze pro určité konfederace systémů. Tato omezení například nijak nevadí pro přístup různých stran na webové služby (původně pouze) knihkupectví Amazon.com, které ačkoliv nabízí přístup na své služby jak prostřednictvím SOAP tak přes REST, druhý jmenovaný je používán z více jak 80% [34, 2].

7.2 Rozhodnutí o SOA

Než začneme SOA uvádět v praxi, nejdříve musíme učinit rozhodnutí, že naše architektura bude právě servisně orientovaná. Jak ukážeme, rozhodnutí o architektuře zásadně ovlivňuje strukturu a obsah specifikace systémových požadavků, protože volbou určité architektury můžeme umožnit nebo naopak znemožnit nějakou funkcionalitu. Navrhujeme-li tedy rozsáhlý systém nebo celou firemní infrastrukturu, je rozhodnutí o architektuře zásadní pro všechny následující kroky včetně fáze získávání požadavků pro její prvky.

SOA řeší problémy specifické skupiny rozsáhlých softwarových systémů, Král a Žemlička zdůrazňují především informační systémy států a globálních korporací. Kvůli špatnému naplňování požadavků těchto systémů stávajícími implementacemi vlastně koncept SOA vznikl. Král a Žemlička mezi tyto požadavky řadí [35, 8]:

- Nekompletní, odporující si, často nejasné a neustále narůstající a měnící se požadavky způsobené např. změnou obchodních podmínek nebo legislativy.
- Nutnost integrovat produkty třetí stran, stejně jako existujících starších systémů.
- Nutnost komunikovat se softwarovými systémy partnerů na různých úrovních součinnosti od komunikace mezi softwary firemních divizí po komunikaci s příležitostným zákazníkem.
- Software podporující automatizaci práce divizí by měl fungovat i v případě, kdy systém jako celek funkční není (např. v důsledku výpadku jeho middleware).
- Proměnlivá, komplexní a stále rostoucí množina uživatelů a uživatelských rolí a měnící se vnitřní struktura a velikost takových systémů.
- Nutnost zajistit samostatnost některých subsystémů, např. policejní databáze v rámci státního informačního systému. Samostatnost je výhodná i ze softwarově inženýrského hlediska.
- Nutnost spolupráce s informačními systémy měnícího se charakteru a nepředvídatelné skupiny partnerských organizací (např. obchodních partnerů v případě globální korporace).
- Měnící se pravidla a standardy pro přístup uživatelů. Uživatelské rozhraní by mělo být snadno modifikovatelné a mělo by skrývat interní strukturu systému.
- Protože počet komponent takových systémů může být velmi vysoký a někteří jeho členové nemusí být v době vývoje systému známy, není možné striktně a centrálně koordinovat a/nebo dohodnout komunikační rozhraní mezi komponentami.
- Neustálé a snadné modifikace systému jsou zásadním požadavkem. Proces změny be

neměl být příliš centralizovaný, protože to by mohlo způsobit nepřípustná zpoždění, způsobená čekáním na centrální autoritu.

Dokumenty americké armády doporučují zvolit SOA, jsou-li ve firmě přítomny následující skutečnosti [43]:

- Firma používá více (starších) systémů, které spolu nejsou schopny komunikovat. (Bod platí i přesto, že i pro zařazení do SOA bude třeba takové aplikace vybavit bránami, aby byly schopné komunikovat s okolí přes nějaký middleware (často jako webové služby přes HTTP))
- Pokud nemá žádnou výhodu (obchodní případ nebo ekonomický význam) budovat nebo kupovat nové řešení, alternativní k těm stávajícím.
- Pokud chce firma snížit svoji závislost na produktech specifických pouze pro jednoho výrobce a použít více softwarových komponent jako služeb.
- Pokud chce firma maximalizovat svojí schopnost vytvořit flexibilní obchodní procesy.

Na stejném místě se dočítáme o faktech, které jsou podle ERP/SOA Resource Center zásadní pro implementaci SOA [43]:

- Zkušený tým potřebný pro vyhodnocení servisně orientovaných projektů.
- Specialisty na tvorbu obchodních procesů.
- Technické úložiště služeb a správce udržující jejich katalog.

7.2.1 Chyby a problémy

Výhody SOA uvedené výše by mohly leckoho motivovat začít tuto architekturu ihned implementovat. Koncept servisní orientace je přehledný a jasný a zdá se, že bude snadné přetvořit stávající architekturu nebo vytvořit novou podle principů SOA a poté již vše bude vzájemně snadno komunikující a tudíž propojitelné a snadno udržovatelné. Nicméně se ukazuje, že cesta k servisní orientaci není až tak snadná, vyžaduje investice, analýzu a především vysoký stupeň standardizace [27, 59].

Díky nekompletnímu porozumění principům SOA není těžké vybudovat něco, co se pouze zdá být architekturou orientovanou na služby, co ale nepřináší zásadní zlepšení celkového fungování automatizace, protože nesplňuje některé ze zásadních požadavků na správně sestavenou SOA. Z těch, které uvádí Erl, se zdají být důležitějšími následující [27, 65]:

- Nesprávné rozdělení funkcionality mezi jednotlivé služby.
- Vytvoření nekomponovatelných nebo pouze částečně sdružitelných služeb.
- Větší důraz na synchronní komunikace.
- Vytvoření hybridní nebo nestandardní architektury.

Dokumenty ERP/SOA Resource Center uvádějí omezení servisně orientované architektury, která by mohly vést ke zklamání po její implementaci [43]:

- Objemné transakce vyžadují významné množství prostředků, to jak technických, tak lidských.
- Bezpečnost je obtížné implementovat, vyměňují-li si systémy dynamicky a globálně informace.
- Začátky vývoje SOA mohou být zdlouhavé.
- Výkonnost nemusí být tak dobrá jako při těsnějších vazbách mezi komponentami architektury.

Král se Žemličkou připomínají, že výhody softwarových konfederací lze plně využít

jen ve firmách, jejichž struktura je decentralizovaná a flexibilní a umožňuje samostatnost organizačních divizí a vertikální i horizontální spolupráci [35, 13].

Stejní autoři na jiném místě zmiňují další problémy, které mohou nastat při přechodu ze stávajících paradigmat (především objektové orientace) k servisně orientovaným systémům, tentokrát problémy psychologické. Mnoho IT pracovníků je neochotných změnit své zaběhlé stereotypy a učit se nové myšlenky (ačkoliv idea SOA není vůbec novátorská) [37, 7]. Softwarové společnosti – integrátoři pak nemusí tuto architekturu příliš prosazovat, protože umožňuje využít stávající aplikace pouze s malými modifikacemi, práce (a tedy peněz) na implementaci tudíž ubyde. Také výrobci software mohou být neochotní vybavovat svoje systémy standardními rozhraními pro použití v SOA (typicky na bázi webových služeb), protože pak může zákazník celkem snadno přejít na software jiného poskytovatele. Někteří poskytovatelé pak vybírají poplatky za používaní middleware propojující jejich software s produkty jiných výrobců [42, 4].

7.3 Klasické metody specifikace požadavků v prostředí SOA

Je postup získávání a obsah specifikace požadavků odlišný pro servisně orientované systémy nebo lze použít beze změn tradičních postupů popsaných v předchozí kapitole? V této sekci se pokusíme na tuto otázku zodpovědět a uvést případné odlišnosti.

7.3.1 Přítomnost vyššího managementu

Ačkoliv si ve fázi zahájení dokážeme představit, že i ti nejvyšší představitelé firmy odpovídají na otázky obchodníků a analytiků, již obtížnější to je ve fázích podrobnějšího rozpracovávání požadavků; na těch se podílejí většinou lidé z nižších stupňů vedení. Ale aby servisně orientovaná architektura přinášela zásadní výhody popisované výše, je třeba, aby (alespoň na nejvyšší úrovni) služby tvořily flexibilní prostředí pro vykonávání obchodních procesů a jejich snadnou modifikovatelnost. Není to však tak snadné, že by se vytvořilo „univerzální“ portfolio služeb, ze kterého se pak procesy jednoduše složí podle aktuální potřeby. Taková flexibilita by znamenala implementovat daleko více, než je skutečně potřeba, tak trochu informační architekturu „univerzální“ firmy. To by bylo velmi drahé a zbytečné. Navíc jako prvky obchodních procesů (podprocesy) vystupují systémy třetích stran, nejen obchodních partnerů, ale také systémy konkurence, případně zákazníků, jenž chceme dostatečně oddělit od interních informací firmy. Zde se nejedná pouze o data, ale i samotný obchodní proces firmy je její konkurenční výhoda, kterou nechce všem zpřístupnit. Velmi často také dochází k reorganizaci fungování firmy a jak bylo naznačeno výše, ačkoliv velkou výhodou SOA je její snadná modifikace, tato flexibilita samozřejmě není nekonečná a při jejím budování je třeba nabrat správný směr.

Tyto důvody vedou k zásadnímu rozdílu specifikace požadavků pro servisně orientované systémy od té pro systémy klasické. Protože vlastně do architektury obtiskáváme fungování firmy jako celku, je nutná aktivní účast vysokého vedení na specifikaci požadavků. Pouze tito lidé mohou učinit zodpovědná rozhodnutí o nejvyšší vrstvě obchodních procesů, spolupráce s třetími stranami i o výhledech do budoucna. Důležité také je, aby tito lidé byli seznámeni s aspekty servisní orientace, aby chápali, kdy a proč přináší výhody. Bez jejich lidově řečeno nadšení servisně orientovanými koncepty a jejich dopady by bylo velmi obtížné je přesvědčovat, že tato cesta jim v dnešním světě přináší větší konkurenční výhodu než řešení tradiční, tedy především architektury s těsnějšími vazbami. V poslední době můžeme sledovat rostoucí zájem i velkých a v korporátním prostředí zavedených firem o SOA (například Oraclu, [62]), nemuselo by proto být těžké o jejích výhodách přesvědčit i skeptičtější manažery.

7.3.2 Charakteristiky SOA ve specifikaci

Pojďme se nyní podívat na hlavní odlišnosti servisně orientované systémy od tradičních (jak je uvádí Erl a jak byly také naznačeny výše) a formulovat, jaké změny to znamená pro specifikaci požadavků [27, 291].

Prvky servisně orientovaných systémů mají být znovupoužitelné. To znamená odlišný přístup ke specifikaci rozhraní, protože takový systém nebudujeme ani tak pro aktuální obchodní potřeby, jako spíš užitečný pro naše dlouhodobé obchodní *cíle*. Rozhraní musí být specifikována jako znovupoužitelná, aniž aktuálně existuje potřeba použít ho na více místech. Do specifikace požadavků se tak dostává implicitní požadavek široké znovupoužitelnosti. Abychom tohoto požadavku dosáhli, je dobré se ptát, jaké systémy mohou *potenciálně* službu využít a jaké by při takovém přístupu měly požadavky.

Rozhraní servisně orientovaných systémů je široce přístupné přes formální kontrakt. Znamená to, že pro každou službu je definované, kde se k ní dá přistoupit, jaké podporuje operace, jaké je třeba zaslat a přijmout zprávy pro každou operaci a také pravidla a charakteristiky služeb. To vše je tedy obsahem tzv. formálního kontraktu, který jakoby představuje smlouvu, jakou bychom uzavřeli s poskytovatelem služby v reálném životě. Tato charakteristika SOA říká, co bychom měli ve specifikaci servisně orientovaného systému nalézt především: dekompozici na služby, ke každé její účel a rozhraní a také informaci, jak její formální kontrakt získat.

Formální kontrakt pak přirozeným způsobem podporuje znovupoužitelnost, protože rozhraní je přesně definováno a je jasné, jak ke službě přistoupit. Technická realizace formálního kontraktu může působit složitě, je ale z velké části dostupný „zdarma“ díky použití technologie webových služeb. Díky WSDL víme, kde službu nalézt, jaké operace s jakými parametry podporuje; díky SOAP pak umíme zprávy realizující komunikaci náležitě formulovat. Pro získání kontraktu je pak třeba vybudovat adresář služeb (postavený na UDDI nebo jeho obdobě), který pak může obsahovat kromě WSDL popisu rozhraní i dodatečné informace.

Spíše než technicky detailně specifikovaný kontrakt by se ve specifikaci požadavků měl objevit způsob, jak kontrakt získat nebo zjistit, tzn. jakým způsobem přistoupit k adresáři služeb a jak s ním komunikovat (to jak pro strojový přístup, tak pro přístup přes uživatelské rozhraní). Samotná definice rozhraní by se pak neobjevila ve specifikaci požadavků jako XML strom, ale jako přirozeným jazykem specifikované požadavky na toto rozhraní. Rozhraní je pak definováno analytiky (architekty) ve spolupráci s manažery firmy, což zaručuje, že bude jak znovupoužitelné, tak také v souladu s potřebami firmy – aktuálními i budoucími.

Servisně orientované systémy jsou vázané volně. Znamená to, že ve specifikaci požadavků by se neměly objevit jiné vazby mezi službami než ty definované jejími formálními kontrakty, tedy přenosy zpráv definovaného formátu. Je též výhodné mít v celé architektuře jednotný middleware, protože jinak je třeba mít brány pro přechody mezi technologiemi, což zeslabuje vlastnost volné vazby.

Služby zapouzdřují svou vnitřní logiku. Pro specifikaci požadavků na nejvyšší úrovni to znamená příjemnou věc: nemusí být příliš podrobná. Není třeba specifikovat technologii ani způsob, jak bude požadovaná funkcionalita docílena. To vše je záležitostí konkrétní implementace dané služby, kterou může vytvářet úplně jiný tým bez znalosti celkové architektury. Ve specifikaci servisně orientovaného prostředí se objeví pouze onen popis rozhraní.

Služby je možné kombinovat. Tento požadavek je, jak uvádí Erl, pouze jiný způsob znovupoužitelnosti, a proto nepřináší zásadní další požadavek na specifikaci systému [27, 302]. Ještě ale více zdůrazňuje nutnost jednotného middleware, jinak se kombinovatelnost stává složitější.

Služby jsou samostatné. Tato vlastnost se zdá mít nejzásadnější vliv na celou specifikaci. Samostatnost znamená, že funkcionalita služby přístupná přes její rozhraní leží v přesně definované oblasti [27, 303]. Tato oblast by pak měla být dostatečně malá, aby došlo k žádané fragmentaci, také však dostatečně velká, aby zahrnovala úzce související funkcionalitu. Otázka dekompozice na služby je předmětem podrobné analýzy a tvoří hlavní činnost fáze Rozpracování. Je více způsobů, jak tuto analýzu provést, na koncích možných řešení jsou analýza shora-dolů a analýza zdola-nahoru[27, 364], [37, 2].

V analýze *shora-dolů* jsou ve středu našeho zájmu na prvním místě obchodní zájmy firmy a její procesy, na základě nich je pak provedena servisně-orientovaná analýza, během níž jsou identifikované potřebné služby na nejvyšší úrovni (konceptuálně, bez vazby na technickou infrastrukturu). Tento přístup často přináší kvalitní architekturu orientovanou dostatečně „netechnicky“, aby byla schopna flexibilně reagovat na měnící se požadavky společnosti a zajišťovala potřebnou znovupoužitelnost jejich prvků. Problémem je pak vysoká počáteční investice (jak finanční tak časová) bez toho, aniž by zlepšení byla brzy viditelná [27, 365-366]. Tento postup se zdá být výhodnější především pro budování architektury „na zelené louce“, tedy úplně od začátku, ještě lépe pak v případě, kdy jsou navíc i všechny služby programovány od začátku.

Analýza *zdola-nahoru* buduje služby nad existujícími aplikacemi a to dle aktuální potřeby. Častou motivací je potřeba integrovat několik stávajících aplikací. Tyto aplikace jsou přetvořeny ve služby tak, že je jim navrženo volně přístupné rozhraní (odpovídající servisně orientovaným principům). V případě proprietárních systémů jsou pak vytvořeny předřazené brány, které na jedné straně komunikují specifickým způsobem (např. proprietárním protokolem) a tuto komunikaci transformují tak, aby ji mohli na druhé straně zpřístupnit platformně nezávislým způsobem ostatním systémům. Výhodou tohoto přístupu je okamžité zvýšení flexibility architektury: Jakmile se byť jen jediný systém přetvoří ve službu a ostatní aplikace s ním začnou komunikovat přes jeho obecné rozhraní, závislost na konkrétní instanci aplikace za rozhraním (branou) se z hlediska integrace sníží na minimum. Tato aplikace může být nahrazena jinou bez vážných globálních dopadů, stejně jako přestane být podstatné z hlediska této služby, jaká konkrétní aplikace s ní komunikuje (protože komunikace již není proprietární, ale standardizovaná). Vytvoříme tak ostrůvek částečné servisní architektury. Jak ale podotýká Thomas Erl, „strategie zdola-nahoru“ je možná příliš vznešené pojmenování, protože tento postup vlastně příliš strategický není [27, 368]. Protože budujeme služby kolem našich stávajících aplikací, přicházíme o mnoho výhod, především tolik žádané flexibility obchodních procesů. Často se pak stane, že nad vrstvou aplikačních služeb pak je vytvářena další vrstva servisní orientace, což může působit realizační potíže. Navíc také nakonec můžeme zažít nevýhody obou popsaných strategií.

Protože oba tyto způsoby mají své nevýhody, je snaha najít „třetí cestu“, která by zdůrazňovala výhody a zároveň zmírňovala nevýhody obou řešení. Snahou tzv. *agilní* strategie je vnést servisně orientované principy do prostředí obchodních procesů bez ztráty možnosti velmi rychlého implementování standardů webových služeb do technické infrastruktury. Tato strategie začíná stejně jako analýza shora-dolů, tedy zeširoka identifikováním obchodních potřeb, jednotlivých procesů a možnosti jejich přetvoření na servisně orientované. Při agilní strategii ale nečekáme, až bude vše vyspecifikované do posledního detailu, ale již ve chvíli, kdy můžeme části architektury prohlásit za „stabilizované“, tyto části začneme implementovat, tedy budovat infrastrukturu webových služeb kolem existujících nebo nových aplikací [27, 370-373]. Rozhodnutí, kdy a jaké části je možné začít implementovat je nejsložitější na celém procesu a je dalším důvodem, proč je důležitá aktivní přítomnost a zasvěcenost vysokého managementu. Toto rozhodnutí je totiž daleko více obchodního než technického charakteru.

Zatímco tedy analýza shora-dolů stále probíhá, již je možné těžit z výhod

implementovaných řešení. Jak se analýza stává podrobnější, nastávají požadavky na změny stávajících služeb. Tyto změny mohou (a typicky budou) vyžadovat rozšíření či změnu funkcionality (formálních kontraktů) již implementovaných služeb. Zde je důležité zajistit kompatibilitu s předchozími verzemi rozhraní, dokud je nějaká část architektury vyžaduje. Agilní strategie přináší konzistentní řešení a zároveň se velmi rychle skutečně implementuje. Jako u ostatních agilních metod, kvůli velmi rychlému implementování existuje riziko, že mnoho bude uděláno zbytečně, protože při podrobnější analýze může vyjít najevo, že již implementovaná funkcionalita nevyhovuje nově objeveným skutečnostem. Uvědomíme-li si ale, jak rychle se mění současné trhy a tím i požadavky na obchodní procesy, zjišťujeme, že agilita je jakási nutnost, kterou podstupujeme, pokud chceme flexibilně reagovat na změny na trhu. To, že implementovanou funkcionalitu nepoužíváme příliš dlouho nemusí být příčinou špatné analýzy či jejího předčasného uskutečňování, ale přirozenou vlastností dnešního světa.

V tomto světle se přístup zdola-nahoru ukazuje jako nejméně vhodný a je žádoucí, aby specifikace požadavků byla vytvářena přístupem shora-dolů. Otázku, zda uplatňovat agilitu či ne již tak snadné zodpovědět není. V sekci o agilních metodách softwarového vývoje bylo poznamenáno, že velká část kritiky těchto postupů je zdůvodňována výslednou ošklivou architekturou. Při specifikování SOA však máme k dispozici jasné principy, jimiž se můžeme řídit. Především vlastnosti jako volná vazba a zapouzdřování vnitřní logiky nám nedávají příliš prostoru pro vytvoření těžkopádné architektury. Ošklivým je často nazývá něco, co je příliš složité, málo flexibilní a tudíž špatně udržovatelné. Servisně orientovaná architektura je ale založena na principech s přesně opačnými cíli.

A jak již bylo poznamenáno, tímto způsobem ovlivňujeme fungování firmy jako celku, tedy prostředí velmi dynamické, kde bychom se nemuseli při úplné analýze shora-dolů dočkat implementace nikdy, protože bychom museli specifikaci neustále obnovovat tak, jak se nám požadavky mění pod rukama. Je též dobré považovat tuto analýzu za kontinuální proces zlepšování, ne za jednorázovou činnost.

Vytváříme-li služby na základě obchodních potřeb namísto pouze jako obálky stávajících aplikací, stává se proces jejich identifikování složitějším. Při specifikaci požadavků nové servisně orientované architektury můžeme vyjít ze stávajících obchodních procesů a identifikovat nové služby na základě jejich analýzy. Thomas Erl doporučuje rozložit proces na jednotlivé kroky – operace – a tyto operace pak sdružit ve služby [27, 400]. V tomto směru je zajímavá úvaha, že jsou to právě jednotlivé operace, které jsou vyžadovány ostatními aplikacemi a uživateli, ne služby jako celky [27, 387]. Služby pak jsou jen skupina souvisejících operací, sdružené za účelem autonomie takto vytvořeného prvku architektury a možnosti opakovaného využití logiky uvnitř takové služby. Bylo by například nepraktické mít implementované operace vytvoření objednávky a její změnu v různých službách, protože takové operace budou ve většině případů sdílet mnoho implementační logiky. Je ale nutné poznamenat, že jejich oddělení by nemělo mít vliv na celkové fungování architektury.

Po identifikování, které operace mají být přeměněny na součást služeb, je možné vytvořit jeden ze dvou typů služby: úkolově orientované nebo entitně orientované [27, 392-394]. Úkolově orientované služby jsou víceméně osamostatněné prvky procesů, příkladem může být služba Ověření faktury. Takové služby mají ale omezenou možnost znovupoužitelnosti, ačkoliv je možné analýzou více procesů nalézt společné podprocesy a vytvořit operaci použitelnou na více místech. Entitně orientované služby sdružují více operací pracujících s jednou entitou v systému, například služba Účetní kniha. Tyto služby vytvářejí flexibilnější prostředí, ve kterém modifikace obchodních procesů nevyžadují tolik změn jako při použití služeb orientovaných na úkoly.

Služby pracují bezestavově. Toto se daleko více váže k definici komunikačního rozhraní než k vnitřní logice. Zprávy mají být navrženy tak, že si služby nemusí pamatovat historii. V případě použití technologie webových služeb pak požadujeme, aby SOAP zprávy

byly spíše dokumentově orientované.

Služby jsou objevitelné. Tento požadavek jednoznačně implikuje potřebu adresáře (registru) služeb, jak bylo již poznamenáno výše. Adresář uchovává všechny informace potřebné ke komunikaci se službami, tedy jejich formální kontrakt. Ten by pak měl být vyhledatelný podle jména služby i podle operací, které obsahuje. Zatímco počítačové systémy (jiné služby) budou typicky vědět identifikaci operace, kterou potřebují využít pro vyřízení požadavku na ně kladený (dotaz typu „všechny služby s danou operací"), měl by adresář být schopen zodpovědět i přehledové dotazy uživatelů (všechny operace vybrané služby, seznam všech služeb apod.)

Jelikož předpokládáme časté změny, ještě důležitější než samotné implementování servisně orientovaných aspektů je vytvořit přehledné prostředí, ve kterém je jasné, podle jakých pravidel se mají stávající služby modifikovat nebo vytvářet nové. Musí být jasné, jak se zjišťuje, že nová služba neobsahuje funkcionalitu již existující jinde a jak se postupuje, pokud se taková duplicita zjistí ex post. Nová funkcionalita by měla být podrobena analýze, při které se zjistí, jestli je výhodnější rozšířit nebo pozměnit stávající službu nebo vytvořit službu novou. Musí být též specifikováno, jak se nová služba zařazuje do adresáře.

Velmi důležité je také specifikovat používané technologické standardy a implementační zásady. Architektura může být těžko nazývána servisně orientovanou, pokud je vytvoření vazby mezi jejími prvky složité. To se ale může stát, pokud se s vývojem požadavků a integraci nových služeb vyvíjí i komunikační standardy bez kompatibility se stávajícím prostředím a staršími řešeními. Je pochopitelné, že pokud vyvineme novou funkcionalitu (ať už ve formě nové služby nebo ve formě rozšíření nějaké stávající), bude třeba změnit služby, které takovou funkcionalitu budou využívat. Žádoucí ale je, aby tato změna byla co nejméně bolestivá. Pokud přineseme s novou funkcionalitou i úplně nový nebo nestandardní způsob komunikace, může být změna zdlouhavá nebo budeme nuceni vytvořit těsnější vazbu, které se chceme v SOA vyvarovat. Pokud například naše infrastruktura používá standardů webových služeb, veškerá komunikační práce s přidanou operací se odbude voláním nové metody, stávající wrappery beze změny donesou potřebné informace až k příjemci. Pokud bychom byli nuceni vyvíjet wrappery nové, bylo by to zbytečně drahé. Samozřejmě s rozvíjejícími se technologiemi jednou bude nutnost udělat radikálnější změnu, žádoucí ale je aby tato změna byla řízená (například i v tomto případě použít verzovaná rozhraní).

Aby bylo vytvořeno stabilní prostředí je žádoucí, aby se rozhraní služeb měnila co nejméně [37, 2]. K tomu nám napomáhá právě jejich na implementaci nezávislá definice vycházející z analýzy obchodních procesů. Rozhraní tak reprezentuje část velmi globálního pohledu na fungování firmy a není vytvářeno programátory „zdola", jenž mají omezené představy o kontextu, do kterého bude jejich řešení zasazeno.

Specifikace požadavků by měla obsahovat několik dalších zásadních bodů, její úplnost a správnost lze (například ve fázi stvrzení) ověřit například položením následujících otázek, částečně z Erla [27, 360-361]:

- Bylo pamatováno i na krajní případy, kdy je některý ze subsystémů nedostupný?
- Respektuje nová funkcionalita dříve dohodnuté politiky a standardy?
- Jak snadné je služby sdružovat?
- Jak snadné je získat popisy služeb (formální kontrakty)?
- Přinášejí nové služby požadavky nebo omezení, které nejsou současnou architekturou požadovány?
- Bylo specifikováno, jak budou navrhované operace testovány?
- Kde budou služby umístěny?
- Jak nové služby ovlivní ty již existující?

- Přináší specifikace nové verze rozhraní, které budou nasazeny společně s těmi současnými?
- Jaké jsou požadavky na bezpečnost celé infrastruktury?
- Jsou služby a jejich adresáře dostatečně škálovatelné, aby zvládly rostoucí množství požadavků; jak se budou detekovat výkonnostní úzká místa?
- Jak budou služby monitorovány, především služby využívající starší a problémové proprietární systémy?
- Jakou formou se bude vykonávat správa verzí rozhraní?

7.3.3 Shrnutí

Chceme-li srovnat specifikaci požadavků servisně orientovaných systémů s tou pro systémy tradiční, měli bychom porovnávat podobné situace. Můžeme srovnat vývoj prvku servisně orientované architektury s tradičním „více izolovaným“ systémem. Pokud ale chceme porovnávat specikaci požadavků celé firemní SOA, na straně tradičních systémů nenalézáme vhodnou obdobu. Můžeme však zvolit velký ERP systém, pokrývající potřeby většiny obchodních činností firmy.

Prvky SOA vs. tradiční produkt

Při porovnávání specifikace služby a tradičního produktu nacházíme zásadní odlišnosti především v otázkách komunikace s okolím. Služba má jasně specifikované rozhraní a díky politice SOA (pravidlech o vytváření nové služby) nejen „uživatelsky“ - významově, ale je poměrně přesně definováno také technicky. Klasický systém naproti tomu taková omezení obvykle dopředu nemá a pokud jsou objeveny v některé z fází specifikace požadavků, jsou uspokojeny ad-hoc řešeními navrženými více z pohledu vytvářeného systému (případně dvou nově komunikujících systémů), bez zasazení do kontextu celé architektury. Ačkoliv samozřejmě taková řešení mohou být uspokojivá, chybí zde v SOA tolik důležité jednotné komunikační prostředí, kde jsou rozhraní definována jednotně (nebo alespoň řízeně).

Pokud se jedná o specifikaci vnitřního fungování, nemusí být u servisně orientovaného a tradičního systému nijak odlišná. U služby ale máme širší možnosti výběru, protože dohodnuté rozhraní je jediné, co je od takového systému požadováno. Protože komunikace v SOA je platformně nezávislá, nejsme již vázáni na komunikační technologie jako Corba nebo RMI, náš produkt může být vytvořen prakticky v čemkoliv. Máme (v nejlepším případě) volnou ruku ve výběru platformy, programovacího jazyka i vnitřní architektury. Pokud je součástí služby kromě určeného formálního kontraktu i uživatelské rozhraní, ani to nepřináší přílišná omezení co do platformy. Samozřejmě, pokud je požadavkem tlustý klient běžící na Windows i Linuxu, obtížně se to bude implementovat například ve Visual Basicu. Dnes je ale velmi často uživatelské rozhraní webové, otázka výběru platformy na straně serveru tím tedy není omezena.

Ačkoliv objektově-orientované principy mají s těmi servisně orientovanými společné prvky, je třeba zdůraznit, že použití objektově orientovaného programování nám automaticky nezaručuje, že systém bude servisně orientovaný. Díky přítomnosti rozhraní a abstrakce jsme schopni dosáhnout jisté znovupoužitelnosti a zcela jistě autonomie. Toto je však pouze na úrovni tříd. Navíc jsou zde nesnadno obdržitelné servisně orientované vlastnosti jako volná vazba (dědičnost nevyhovuje), bezestavovost nebo objevitelnost. Tyto dva přístupy ale nejsou soupeři na stejném hřišti, mohou být velmi dobře doplňovány. Servisní orientace by měla být uplatňována „vně“ systémů, v rapidně se měnícím rozsáhlém prostředí, kde požadujeme maximální flexibilitu. Jednotlivé prvky takového prostředí pak mohou být implementovány s použitím objektově orientovaného programování. Také tak často implementovány budou, vzhledem k tradici a vyspělosti tohoto přístupu. Je-li granularita služeb vysoká, jsou jednotlivé služby nevelkými projekty, po kterých tolik nepožadujeme, aby byly uvnitř stejně

flexibilní jako celková architektura. Zde si vystačíme s tradičními technikami jako jsou programy založené na mikrojádru a modulech apod.

Skutečnosti popsané výše ukazují, že specifikace požadavků servisně orientovaného systému může vypadat téměř totožně jako specifikace produktu tradičního. I zde bychom měli mít podrobně rozepsané fungování, design, testování a způsob instalace. Stejně je i třeba sledovat (stopovat), jsou-li požadavky v implementovaném systému naplněny. Prvky SOA by ale měly být významně méně rozsáhlé, a proto i jejich specifikace mohou být kratší a produkty podle nich vytvořené snadněji organizovatelné. Lze tedy pracovat s menšími týmy a také využít agilní metody vývoje.

SOA vs. velký tradiční ERP

Srovnáme-li specifikaci požadavků servisně orientované architektury a velkého ERP systému tradiční architektury, velký rozdíl je patrný již v samotném účelu takové specifikace. Jakkoliv obecný a modulární ERP systém může být, vždy bude něco vně, co jím pokryto není a ani není zamýšleno, že bude pokryto. Naproti tomu SOA představuje komunikační platformu pro aplikace všech možných účelů, existujících či budoucích. Tradiční ERP systém bude nejčastěji vyvíjen jednou firmou a v jejím zájmu jistě není, aby bylo snadné jejich produkt nebo jeho část nahradit jiným. Naproti tomu jedním z primárních cílů servisně orientované architektury je snadná substituovatelnost jejich prvků. Ta je podmíněna také volnou vazbou mezi nimi. Tradiční architektura, jakkoliv modulární, jen těžko zajistí tak volnou vazbu, jaká je požadovaná v SOA. Opět to vůbec není v zájmu firmy implementující takový systém.

ERP systém od jedné firmy ne vždy nebude závislý na jedné aplikační platformě. Jakékoliv jeho rozšíření a změny (i kdyby je mohl dělat někdo jiný než původní implementátor) jsou touto platformou omezeny. Stejně tak i vytváření propojení na jiné systémy je ovlivněné platformou produktu, často navíc nemusí být vůbec snadné, zvlášť pokud daný výrobce nabízí vlastní řešení nahrazující ono propojení.

V rozdílech specifikací požadavků těchto dvou systémů se odrážejí všechna specifika servisní orientace, jak byla popsána v předcházejících odstavcích a kapitolách. Ačkoliv úvodní specifikace SOA obsahuje také mnoho konkrétní funkcionality (dekompozici funkcionality na služby a jejich definice), slouží především k vytvoření otevřeného, platformně nezávislého komunikačního prostředí (middleware) volně vázaných softwarových děl. Při dalších revizích jsou pak myšlenky specifikace nadále udržovány v souladu se základními principy SOA. Tradiční ERP systém je daleko více orientován na aktuální potřeby a jeho specifikace řeší konkrétní obchodní problémy a nezdůrazňuje tolik otevřenost, volnou vazbu a platformní nezávislost.

Tradiční a ověřené metody správy požadavků však platí pro oba takové systémy. V obou případech je třeba zajistit stopovatelnost požadavků ve vytvářeném produktu, specifikovat způsob testování i instalace, stejně tak je třeba pamatovat na škálovatelnost a bezpečnostní otázky. Důležitá je i častá zpětná vazba od uživatelů a na jejím základě inkrementální zlepšování.

7.4 Použitelnost CASE nástrojů

Jedním z cílů této práce je zhodnotit použitelnost nástrojů pro počítačovou podporu softwarově inženýrských procesů (Computer-Aided Software Engineering, CASE) při specifikaci servisně orientovaných systémů. Stejně jako v předchozí kapitole je pohled odlišný, mluvíme-li o servisně orientované architektuře jako celku nebo jejích prvcích.

Pro prvky architektury nenacházíme tolik rozdílů od tradičních aplikací, nástroje lze tedy využít všechny téměř beze změny. Jak bylo zmíněno v předchozí kapitole, formální

kontrakt služby přichází „shora“ a je základním požadavkem na systém. Ostatní požadavky a návrh probíhá tradičně, máme však díky autonomitě obecně více možností při výběru platformy a dalších technologií. Produkt může být řízen systémem správy požadavků a změn a navrhován v objektových diagramech UML. Testování produktu je pak usnadněno díky existenci formálního kontraktu, je tedy snadné vytvořit automatické testy. Má-li ovšem systém uživatelské rozhraní, je jeho testování stejně složité jako u jiných aplikací; výhodou však může být, že služba je často nevelkého rozsahu.

V předchozí kapitole bylo uvedeno, že specifikace servisně orientované architektury by měla obsahovat kromě obecných nároků na systém technologie, které sjednocují komunikační prostředí firmy, pak také rozdělení požadované funkcionality mezi jednotlivé služby – tj. vytvoření jejich rozhraní (formálních kontraktů). Výhodné je navrhnout i adresář služeb, ten by měl fungovat jako každá jiná služba, komunikuje se s ním tedy stejnými protokoly využívajíc jeho formální kontrakt. Potřebujeme tedy nějakou formu psané specifikace (ať už je to textový dokument nebo databáze požadavků) a diagramy zobrazující množinu služeb a rozhraní a možnosti komunikace jejích prvků. Vzhledem k nutnosti významné spolupráce s vyšším managementem je velmi důležité, aby všechny prvky specifikace byly přesné a zároveň jednoduše pochopitelné. Pracujeme-li na tradičním systému pro velkého zákazníka, často požadavky kontroluje poměrně fundovaná osoba, která rozumí formálním popisům systému, můžeme tak trochu alibisticky snahu o jednoduchost opomenout; ne však v případě servisní architektury odrážející fungování celé firmy. Pro psanou specifikaci lze využít libovolný z nástrojů prostředků specifikace požadavků. Jelikož se jedná o požadavky na celou firemní architekturu, je očekávatelné, že jich bude velké množství a postupem času tento seznam bude výrazně měnit a narůstat. Je proto žádoucí použít sofistikovaný systém na správu takových požadavků. Managementu se pak mohou předkládat výstupy z tohoto softwaru, které by měly být napsány tak, aby nebylo těžké jim porozumět. Výhodné by mohlo být vytvořit více vrstev požadavků, kdy techničtější podrobnosti by nemusely být zobrazovány vždy. Pro menší řešení pak dostačuje verzovaný textový dokument (popř. listy tabulkového procesoru), specifikující vše potřebné.

Samotná definice služeb a jejích rozhraní je náchylná ke změně ještě výrazněji. Především u rozhraní je třeba hlídat velké množství závislostí verzí požadovaných jednotlivými službami. Implementujeme-li novou verzi formálního kontraktu, je třeba vědět, které jeho části (operace) jsou stále používané některými prvky systému a zachovat pro ně zpětnou kompatibilitu. Toto je spíš úloha pro konfigurační správu, můžeme tedy použít některý z nástrojů pro něj určený (IBM Rational ClearCase, Telelogic Synergy apod.). Lze ji však vyřešit i systémem na správu požadavků, uvážíme-li službu jako jeden požadavek a operace jejího formálního kontraktu jako další jednotlivé požadavky. Pak může být takový požadavek reprezentovaný službu závislý na mnoha prvcích formálních kontraktů služeb ostatních. Prvky formálního kontraktu každé služby jsou pak závislé na službě, ke které přísluší. Díky tomuto grafu závislostí se při změně operace snadno zjistí, které služby stále využívají starou verzi a jestli je ji třeba zachovat (případně novou verzi vytvořit tak, aby byla zpětně kompatibilní). Naopak pokusíme-li se odstranit celou službu z architektury, rychle zjistíme, jaké má operace a jestli nejsou stále někým využívány. Není to ani příliš velké zneužití softwaru pro správu požadavků pro jiné účely, protože služby i jejich operace nejsou nic jiného než požadavky na funkcionalitu systému. S výhodou lze také využít možnost propojit požadavky s prvky diagramu graficky zobrazující množinu služeb a jejich rozhraní. Pokud bychom takový diagram vytvořili v UML, snadno bychom se od obrázku mohli v přesunout k textu popisujícím přesně účel služby a její operace.

Vytvořit UML diagram servisně orientované architektury je snadnější, než vytvořit diagram architektury tradiční. Díky homogenitě architektury na nejvyšší úrovni může být množina vyjadřovacích prostředků (elementů v diagramu) velmi malá. Diagram by měl

zobrazovat služby, jejich operace, jaké služby volají jaké operace, případně pak možnost služby sdružovat. To vše je snadno vytvořitelné v UML diagramu komponent (a to dokonce již v první verzi UML). Komponenty reprezentují služby, rozhraní pak jednotlivé operace formálního kontraktu (při větším množství takových operací bude ale nutné je seskupit a rozepsat podrobně až v navázaném požadavku nebo jinde). Volání lze pak zobrazit závislostmi mezi komponentami. V případě potřeby lze využít i zobrazení vlastnosti dědičnosti.

Pokud by diagram komponent nedostačoval našim modelovacím potřebám, můžeme využít UML profil pro softwarové služby od IBM [45]. UML profil je standardní způsob, kterým lze rozšiřovat stávající UML diagramy a elementy; tento konkrétní pak byl vyvinut pro produkt IBM Rational Software Architect. Můžeme ho použít například v případě, kdy chceme zobrazit smíšenou architekturu obsahující jak služby, tak komponenty v tradičnějším slova smyslu (více ve smyslu UML 1.0, tedy například spustitelné soubory apod.). Díky jeho rozšířením pak můžeme zobrazit navrhovanou architekturu podrobněji.

Profil uvádí důležité koncepty nutné k modelování servisně orientovaných řešení: *službu*, která může být znázorněna pouze jako *poskytovatel* služby a být propojena s *odběratelem* pomocí *kanálu* a která obsahuje *specifikaci* ve více *protokolech* nebo *kolaboraci*, to v případě, že je chování služby definované funkcionalitou více služeb dohromady. Specifikace se pak skládají z *operací*, které jsou tvořeny libovolným počtem *zpráv*, volitelně s několika *přílohami*. Služby pak mohou být seskupeny do *sekce*, která má *bránu*, jenž reprezentuje rozhraní sekce a má stejné vlastnosti jako služba, jen nemůže být navázána na kolaboraci.

Jelikož některé elementy (stereotypy) z tohoto výčtu jsou již v UML 2 přítomné (např. protokol), profil definuje jen ty chybějící: zprávu, přílohu, službu, kanál služby, služební kolaboraci, odběratele služby, bránu, sekci, poskytovatele služby a její specifikaci. Stereotypy mají přirozená omezení, na jaké jiné se mohou vázat a také některé vlastnosti, například zpráva a příloha mají kódování, poskytovatel služby má své umístění apod.

Jak bylo ukázáno, je možné specifikovat servisně orientovanou architekturu s použitím stávajících CASE nástrojů. Je možné i udržovat pořádek v často se měnících rozhraních a to velmi pohodlně, použijeme-li systém na správu konfigurací nebo požadavků. V druhém případě pak můžeme navázat podrobné specifikace rozhraní služeb z těchto programů na prvky diagramu zobrazujícího celou architekturu. Takový diagram pak můžeme vytvořit pomocí tradičního UML diagramu komponent nebo za použití IBM rozšíření UML pro softwarové služby.

7.5 Případová studie – integrace nové služby do existující SOA

Pro lepší představu o praktické realizaci SOA a jejích dopadech na informační strategii byla do práce zařazena případová studie popisující zařazení nově vyvinuté služby do existující rychle se rozšiřující architektury směřující k implementaci servisně orientovaných vlastností. Ačkoliv příklad nepopisuje prostředí rozsáhlé korporace s komplikovanými obchodními procesy, může dát čtenáři základní praktickou představu, popř ukázat mu, kde všude může SOA přinést zásadní zkvalitnění informační infrastruktury.

7.5.1 Úvod

V době psaní této diplomové práce nacházíme jejího autora v amsterdamské firmě Noterik B.V. zabývající se media asset managementem a prezentací videa na webu. Mnoho projektů je postaveno kolem produktu StreamEdit MAM do více či méně servisně orientované architektury [46], 2].

StreamEdit se využívá k ukládání, konverzi, organizaci a streamovanému přehrávání

video souborů. Produkt má vlastní webové rozhraní, které umožňuje nahrát nové video, které je pak zkonvertováno do více formátů (například pro přehrávání na mobilních telefonech) samostatnou komponentou – tzv. transcoderem – a dekódované soubory jsou uloženy do souborového systému. Rozhraní také umožňuje - kromě tradiční funkcionality jako je správa uživatelských účtů - měnit popisné informace o videu, organizovat více filmů do kolekcí a filmy přehrávat. StreamEdit udržuje databázi o jednotlivých videosouborech, jejich příslušnost k jednomu prvku obsahu (tzv. assetu, jeden asset obsahuje videosoubory ve více formátech, všechny však reprezentují stejné video, například stejnou reportáž). Samotné soubory jsou pak ze souborového systému vyzvedávány streamovacími servery (těch je více pro různé video formáty) na požadavky aplikací.

Velmi často ale zákazníci rozhraní StreamEditu nepoužívají a funkcionalita tohoto produktu je využívána pouze jako inteligentní úložiště pro data jiných projektů. Jako například v případě EdTV, webového modulu obohacujícího stávající zpravodajskou stránku vydavatelského domu Eindhovens Dagblad o multimediální obsah – video reportáže [47]. Tento produkt má vlastní frontend, který mají možnost vidět uživatelé na stránce zpravodajství, také však vlastní backend umožňující správcům aplikace ovlivňovat obsah zobrazující se přes frontend (backend je navíc považován za obecnější produkt než jen pro EdTV a nazýván W-IPTV). Důvodem, proč jako backend není používáno samotné administrační rozhraní jsou odlišné požadavky na jeho funkcionalitu. Také jedna instance StreamEditu může být využívána více aplikacemi, ke všem datům StreamEditu by tedy měli mít přístup pouze správci této aplikace, ne jednotlivá vydavatelství systém využívající.

Na stejném principu je postaven video portál pro nizozemského poskytovatele Internetu XS4ALL, v době psaní práce stále ve vývoji bez běžící verze on-line. Aplikace má prezentační část (frontend), zobrazující vybraná videa vytvořená uživateli a správcovskou část, která umožňuje uživatelům nahrávat a upravovat jejich filmy a superuživatelům pak vybraný obsah zakázat, spravovat databázi uživatelů apod.

Při pohledu na celkovou architekturu obou aplikací vidíme tři volněji nebo vůbec vázané části, v nich pak těsněji napojené komponenty (viz Obr.1). Na jedné straně máme aplikačně specifickou část s její databází uchovávající informace, které v databázi StreamEditu nemohou být uloženy – StreamEdit například zatím neumí pracovat s obrázky z filmů pro jejich prezentaci na frontendech. Aplikační frontend i backend komunikují s databází přes MySQL konektory jazykem SQL, vazba je tedy velmi těsná. S ostatními částmi architektury aplikace potřebuje komunikovat ve třech rozdílných případech:

- je třeba vyzvednout nebo aktualizovat údaje o videosouborech, „assets“, jejich kolekcích, uživatelích apod. v databázi StreamEditu
- je třeba přehrát video pro uživatele frontendu nebo backendu
- uživatel chce nahrát nový film za použití backendu aplikace

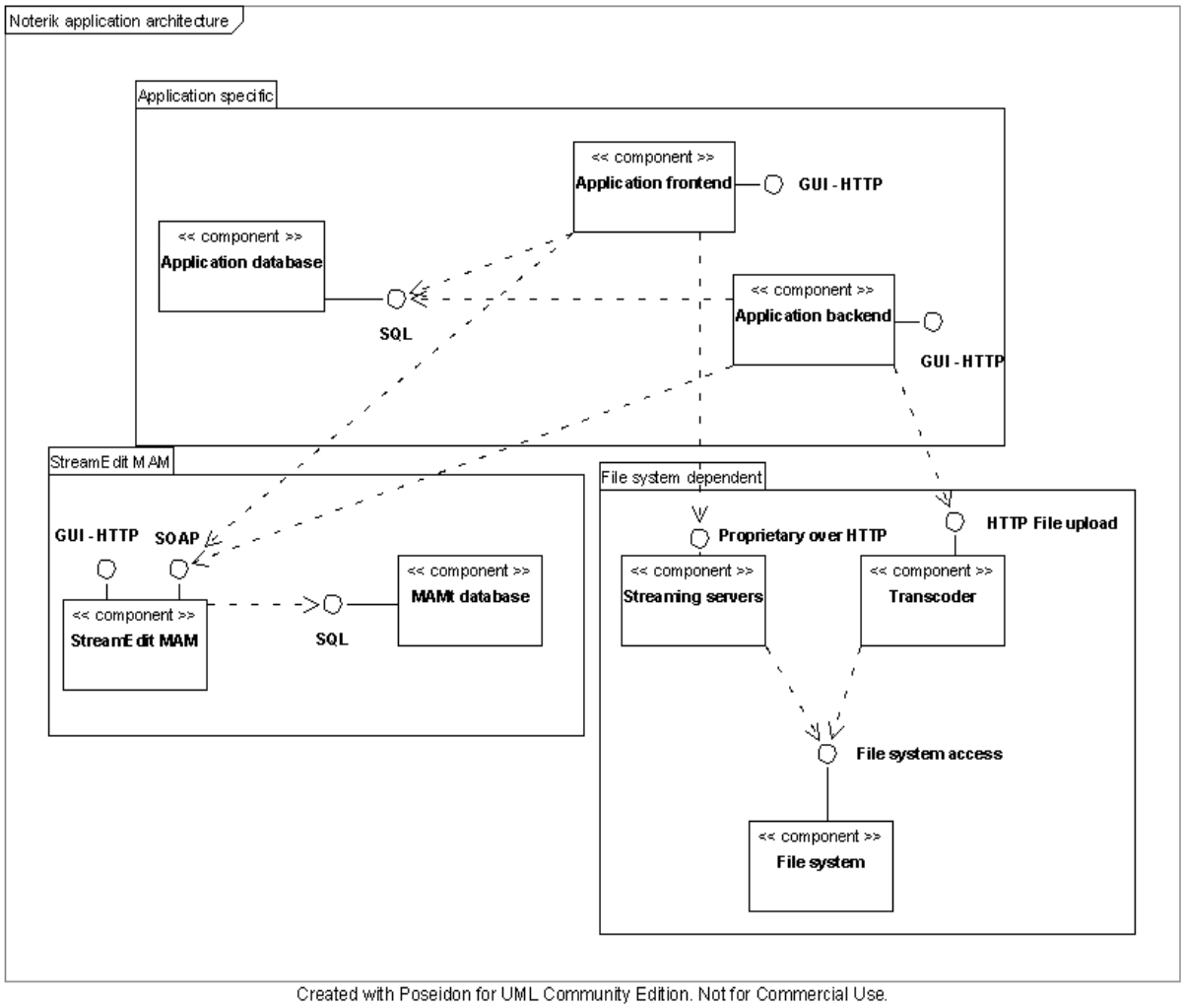
Poslední případ je realizovaný přes standardní HTTP upload souborů. Po nahrání souboru na serveru je (opět přes HTTP) aktivován transcoder, který pak ze souboru vytvoří více verzí v různých formátech. Webová část transcoderu není velká, pouze aktivuje v C++ vytvořený program. Ten musí mít přímý přístup k uploadovanému souboru, ať už lokálně nebo přes síťový souborový systém.

Druhý případ je realizovaný požadavkem na streaming server. Po získání informací o videu z různých zdrojů je příslušný streamovací server požádán o jeho přehrání, to se realizuje protokolem specifickým pro daný obsah, přenášený jak přes HTTP, tak proprietárně jako v případě Microsoft Media Server protokolu.

První případ pak vyžaduje kontakt StreamEditu. Ten je realizován požadavky na jeho rozhraní protokolem SOAP přes HTTP. Operace jsou orientovány hodně datově, jako například Seznam assetů dané uživatelské skupiny. StreamEdit pak komunikuje se svojí

databází (opět přes těsné SQL konektory) a se souborovým systémem. Na ten přistupuje také transcoder a streamovací servery.

Jelikož StreaEdit nijak nekomunikuje ani se streamovacími servery, ani s transcoderem, ty tři jasně oddělené části jsou tedy aplikační databáze, její frontend a backend, StreamEdit a jeho databáze a poté zbytek architektury: StreamEdit a jeho databáze, transcoder a streamovací servery. Zatímco je snadné umístit nějakou část na jiný počítač než ostatní dvě, velmi daleko od sebe a propojit je pouze Internetem (a často se tak také děje), rozdělit komponenty uvnitř těchto částí je nemožné nebo velmi obtížné.



Obr.1: Aplikační architektura Noteriku: tři oddělené soustavy těsně propojených komponent.

7.5.2 Úkol

Streamovací servery produkují podrobné statistiky o požadovaných přehraných „proudech“: jméno přehrávaného souboru a jeho umístění na souborovém systému, délka videa v souboru a skutečný čas přehrávaní, také podrobné informace o klientovi (IP adresa, prohlížeč, operační systém apod.). Takové statistiky jsou velmi užitečné pro zákazníky a jsou jejich častým požadavkem. V případě EdTv bylo třeba použít statistiky o přehrávaných souborech k zjištění tří nejpopulárnějších reportáží a jejich zobrazení na frontendu i k podrobnějšímu informování superuživatelů backendu. Pro tyto účely se evidovaly údaje o přehrávaných souborech v aplikační databázi.

Později ale vznikl požadavek na podrobnější statistiky jako například informaci o délce videa skutečně přehrané uživatelem. Takovou informaci nešlo (snadno) získat na straně

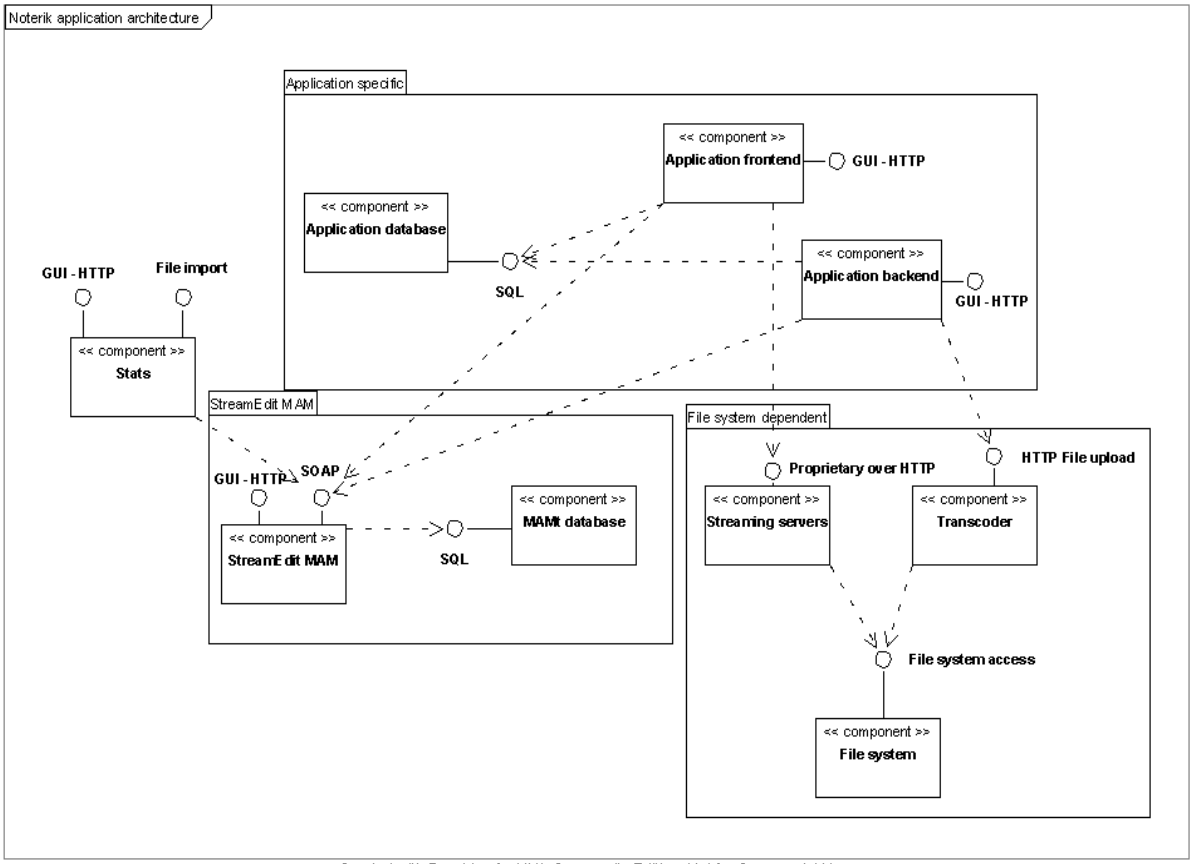
aplikace, protože přehrávání se realizovalo v do stránky vloženém proprietárním (black-box) přehrávači, který získaval data přímo ze streamovacího serveru. Protože tyto servery jsou dodávány též jako černé skříňky bez možnosti úprav, bylo třeba použít jeho logovací soubory. V té době se připravovalo rozšíření StreamEditu o parser těchto souborů, to však nemělo být hotovo dlouho před plánovaným dodáním EdTv. Bylo tedy rozhodnuto vytvořit dočasnou aplikaci na zpracovaní těchto informací.

Tato aplikace měla být schopna načíst logovací soubory streamovacích serverů, kontaktovat vybraný StreamEdit a zpřístupnit získané informace dalším aplikacím. Kontaktovat StreamEdit je nutné z důvodu převodu jména souboru na identifikátor „assetu“ (kolekce videosouborů reprezentující jeden film), který bylo nutné ve statistikách uvést. Povšimněte si, že streamovací servery neví nic o metadatech a kontextu přehrávaných souborů a jsou nezávislé na instancích StreamEditu a mohou tedy být (a jsou) použité pro více z nich. V jejich logovacích souborech se tedy objevují statistiky identifikované pouze jménem souboru.

Vzhledem k požadavku na rychlé dokončení a nevysokým nárokům na zapojení aplikace do stávající architektury (jak uvedeno níže, vazba měla být volná – výběr platformy tedy tímto nebyl příliš omezen) byl zvolen jazyk Ruby a framework Ruby on Rails [49]. Aplikace, která dostala jméno Stats, si udržuje vlastní databázi s údaji načtenými z logovacích souborů. Samotný import těchto dat probíhá ve dvou krocích: v prvním jsou načteny pouze informace z logovacích souborů streamovacích serverů, ve druhém je pak specifikován konkrétní StreamEdit a ten pak použit pro zjištění identifikátorů „assets“, které náleží k jednotlivým logovým záznamům. Tím je databáze aplikace naplněna a je možné data poskytovat ostatním aplikacím. Bylo vytvořeno jednoduché webové rozhraní, které umožnilo uživateli vybrat časové rozhraní a typ streamovacího serveru (v první verzi byly k dispozici servery pro soubory typu Windows Media a Real Media). Na základě těchto parametrů byl z údajů v databázi vygenerován CSV soubor podle požadavků uživatelů EdTv [50]. Na stránku Stats pak vedl odkaz z backendu EdTv (kdyby byl vzhled obou aplikací stejný, z pohledu uživatele by Stats byl částí EdTv).

Stats tedy dávkově načítala logovací soubory (jejich názvy byly předávané příslušným skriptům), poté komunikovala se StreamEditem využívajíc jeho SOAP rozhraní přes HTTP, uživatelům pak nabízela webové stránky a exportovaný soubor opět přes http (viz Obr. 2). Pokud uvážíme možnost zkopírování logovacích souborů od streamovacích serverů k aplikaci, dostává se Stats mimo obě části dosud popsané architektury, protože vazba na obě je volná.

Toto řešení postačovalo pro EdTv, nebylo však příliš použitelné pro jiné aplikace. Pro video portál XS4ALL byly opět požadovány obdobně podrobné statistiky, ne však exportované do CSV a v poněkud odlišné struktuře. Jelikož příslušný modul StreamEditu stále nebyl dokončen, bylo rozhodnuto použít existující aplikaci Stats. Důležitým požadavkem ale bylo, aby statistiky byly přirozeně integrované do rozhraní video portálu.



Obr. 2: Přidání aplikace Stats do stávající architektury.

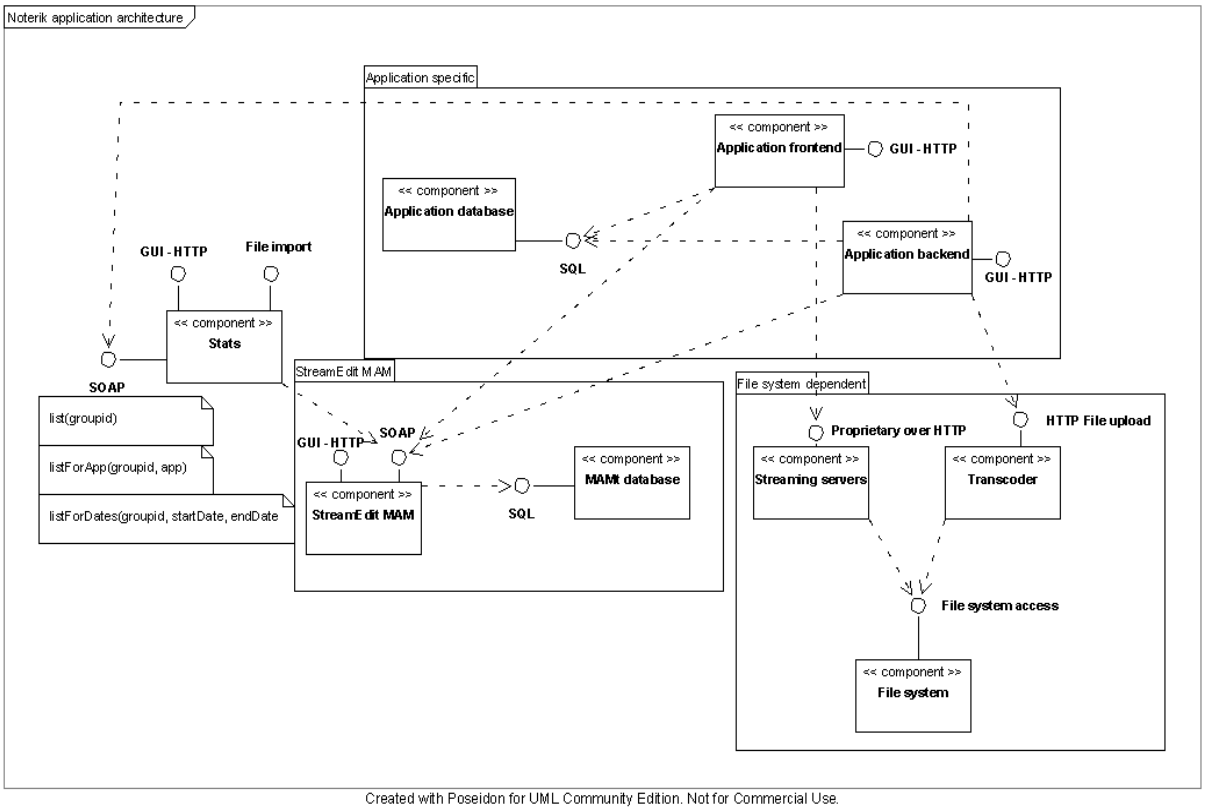
7.5.3 Řešení

Shrneme-li situaci z předchozích odstavců, dostáváme následující požadavky a omezení:

- Chceme upravit aplikaci Stats tak, aby se její funkcionalita dala použít v nové koncové aplikaci a nejlépe tak, aby se dala použít ve více takových produktech.
- Díky Stats máme existující parser souborů s vlastní databází.
- Databáze Stats obsahuje vše, co potřebujeme.
- Stávající rozhraní je nevhodné pro účely další aplikace ze dvou důvodů:
  - Nevhodné údaje (jiný než požadovaný formát) a nemožnost je ovlivňovat na straně specifické aplikace.
  - Nehodící se vzhled uživatelského rozhraní.
- Kód aplikace Stats není možné integrovat do stávajících aplikací, ty jsou vyvíjeny na jiné platformě a v jiném jazyce. Je obtížné vytvořit komponentu v jazyce Ruby a integrovat ji do aplikací používajících PHP nebo Javu. Takové řešení je též nevhodné z mnoha softwarově inženýrských důvodů (vytvořil by se špatně udržovatelný těsně vázaný heterogenní systém, de facto by se replikoval totožný kód apod.).

Na základě těchto požadavků bylo rozhodnuto vytvořit rozhraní Stats pohodlně přístupné ostatním aplikacím. Další (odmítnuté) varianty byly použít uvnitř XS4ALL video portálu (upravený) CSV soubor generovaný Stats nebo přímo logovací soubory streamovacích serverů, obě varianty ale nevyužívaly již hotovou funkcionalitu a byly proto zbytečně zdlouhavé.

Dále bylo rozhodnuto, že rozhraní bude implementované jako webová služba, především kvůli možnosti přistupovat přes běžný internetový HTT protokol a také kvůli velkému výběru volně dostupných nástrojů použitelných na straně aplikací pro volání takového rozhraní. Tato přístupová metoda je též nezávislá na technologiích použitých v aplikacích a je snadno implementovatelná. V aplikacích vyvíjených v Ruby on Rails je díky objektům kolem ActionWebService stejně obtížné vytvořit rozhraní přístupné jak protokolem SOAP, tak XML-RPC, tyto dvě možnosti tedy nebyly uvažovány zvlášť. Protože se ale již v XS4ALL video portálu, EdTv a obecně veškerých aplikacích této struktury využíval SOAP pro komunikaci se StreamEditem, bylo přirozené preferovat tento protokol na straně klientů. Protokol byl do aplikací implementován pomocí knihovny NuSOAP, která snížila technickou práci na zpracování XML zpráv na minimum – WSDL automaticky vygenerované ze tříd na straně Stats je na straně aplikací použito pro generování PHP tříd. Odeslat požadavek na vzdálené rozhraní Stats a zjistit jeho výsledek pak pouze znamená zavolaní příslušné metody třídy a přečtení její návratové hodnoty.



Obr. 3: Službě Stats přibylo SOAP rozhraní. Vzhledem k nemožnosti přístupu k originálnímu souboru a přílišné rozsáhlosti tištěného reportu byly do obrázku zaneseny i názvy implementovaných metod.

Také pak nebyla zvolena žádná varianta RESTu – posílání informací ve formě XML na URL definované pro jednotlivé „prostředky“ (resources, typy obsahu, často odpovídající jednotlivým tabulkám). Jak bylo naznačeno výše, tato forma tvorby rozhraní nevytváří dostatečně flexibilní architekturu, především díky neexistenci ekvivalentu WSDL pro SOAP, díky němuž je snadné zjistit definici rozhraní a například možné automaticky generovat kód pro wrappery SOAP volání.

Jediný aktuální požadavek na rozhraní Stats byly nároky video portálu XS4ALL. Jak bylo ale poznamenáno výše, služby by měly být budovány jako znovupoužitelné bez ohledu na to, jestli existuje aktuální potřeba využít jejich funkcionality na více místech. Jako první operace proto byly definované obecné operace zpřístupňující téměř všechna data získaná z

logových souborů. Jak víme, jeden StreamEdit může sdílet více aplikací, jejich data jsou identifikovány uživatelskou skupinou. Všechny operace tedy implicitně uvádějí skupinu uživatelů, jejichž data je zajímají. Prvotně definované operace byly Seznam všech logových záznamů, Seznam všech logových záznamů pro daný streamovací server a Seznam všech logových záznamů pro rozsah dat (viz Obr.3). Tyto operace shodou okolností jsou téměř shodné s tím, co bylo vyžádáno pro aplikaci EdTv a co je tedy možné získat přes webové rozhraní Stats.

Takto obecné operace jsou de facto vše, co aplikace může potřebovat, stačí si vyžádat potřebné záznamy a ty pak na svojí straně zpracovat – uplatnit další filtrační parametry a data zobrazit. Další možností je přidat do Stats operace, které tuto práci udělají na její straně. Například do XS4ALL bylo požadováno mimo jiné informace o spotřebovaném přenosovém pásmu pro jednotlivé video soubory. Taková informace se dá „vypočítat“ za použití informací z obecných operací nebo získat přímo specializovanou operací; aplikace by pak pouze data zobrazila. V následujících odstavcích najdeme návrhy i na toto téma.

7.5.4 Diskuse

Uvedený příklad reprezentuje prostředí velmi často se měnící architektury, kdy je dřívější těsně vázaná skupina systémů přetvářena ve flexibilnější architekturu postavenou na webových službách. Toto se děje bez zásadní analýzy, přístup je to tedy spíše zdola-nahoru (jak popsaný výše).

Můžeme si povšimnout, jak byla vyřešena potřeba znovupoužitelnosti StreamEditu ve více aplikacích: bylo mu vytvořeno SOAP rozhraní. Stejná strategie pak byla zvolena v případě nutnosti znovupoužít aplikaci Stats. Znovupoužitelnost je tedy v architektuře realizována pomocí „obálek“ stávajících aplikací zpřístupňujících jejich funkcionalitu. (Slovo obálka je použito záměrně místo termínu „brána“, které se uplatňuje v případě nemožnosti aplikace modifikovat). Architektura má v částech s již implementovaným rozhraním webových služeb následující vztahy k servisně orientovaným vlastnostem:

- Znovupoužitelnost. Možnost opakovaně použít danou funkcionalitu byl hlavní důvod, proč začala být tato rozhraní implementována. K službě Stats lze přistoupit z jakékoliv aplikace voláním jejího rozhraní.
- Volná vazba. Díky použití protokolu SOAP přes HTTP je vazba velmi volná. Prakticky z jakéhokoliv systému je možné poslat SOAP zprávu přes běžný internetový protokol, pro většinu aplikačních platforem pak navíc existují nástroje, které takovou činnost dělají velmi snadnou.
- Dohoda o službě. Formální kontrakt je k dispozici pouze částečně. Díky použití webových služeb máme WSDL popis rozhraní služby a tedy seznam všech operací, které služba podporuje včetně jejich parametrů a datových typů. Přístupový bod je ale definován staticky (neexistuje zde adresář služeb), při jeho změně je třeba ručně změnit kód aplikace.
- Samostatnost. Jelikož StreamEdit i Stats mají vlastní webová rozhraní, lze je používat bez ohledu na funkcionalitu dalších prvků architektury. To je ale spíše důsledkem toho, že služby byly (do)budovány jako obálky existujících aplikací, ty pak samozřejmě již měly svá uživatelská rozhraní. Nepřítomnost celkové strategie může způsobit potíže. Služba Stats byla míněna jako dočasné řešení, zamýšleným krokem je integrovat pársování statistik do StreamEditu. Když se ale na situaci podíváme z nadhledu, zjistíme, že separátní služba více vyhovuje servisně orientovaným požadavkům a tím také přináší mnoho výhod. Jak víme, streamovací servery mohou být používané pro aplikace používající více StreamEditů. Bude-li stávající funkcionalita aplikace Stats implementována v nich, výrazně to poruší žádanou vlastnost znovupoužitelnosti, protože stejná funkcionalita bude v architektuře

přítomna dvakrát. To je sice přirozený důsledek existence více instancí stejné aplikace (StreamEditu, například z výkonnostních důvodů), ale při větší granularitě služeb by se množství duplicit mohlo výrazně snížit.

- Abstrakce. Požadavek abstrakce je u nových služeb poměrně splněn. Uživatelské rozhraní není v tomto případě na překážku a jiné než specifikované (v tomto případě implementované) operace volat nelze. Je důležité podotknout, že pro aplikačního programátora není téměř žádný rozdíl mezi voláním lokální a vzdálené metody – operace formálního kontraktu. Ukazuje se tak, že je-li infrastruktura postavena na zavedených standardech, mnoho programátorů nemusí být příliš seznámeno se servisně orientovanými principy a mohou využívat pouze své znalosti OOP.
- Kombinovatelnost. Díky standardům webových služeb je kombinovatelnost technicky možná. Z obecnějšího pohledu je poněkud komplikovanější díky nejasnému rozdělení funkcionality.
- Bezestavovost. Žádné dosud implementované operace se neodkazují na nějaký předchozí stav, tato vlastnost je tedy splněna beze zbytku.
- Objevitelnost. Díky nepřítomnosti adresáře služeb není vlastnost objevitelnosti v architektuře přítomná vůbec, přístupové body a cesty k definicím rozhraní musí být zadávány manuálně.

Ačkoliv uvedený příklad reprezentuje přeměnu tradiční architektury na architekturu využívající webové služby bez hlubší analýzy, můžeme si povšimnout, že i tak je dosaženo mnoha výhod, aniž by to přinášelo nevýhody: zvýšila se především znovupoužitelnost a volná vazba, díky níž mohou bez problémů komunikovat systémy postavené na odlišné platformě, stejně jako by nyní bylo mnohem snadnější nahradit některou ze služeb za jiný systém (například postavený na úplně odlišné architektuře).

Pro lepší výsledky by bylo zapotřebí podrobně rozebrat funkcionalitu StreamEditu, streamovacích serverů a transcoderu a rozdělit je na více (potenciálně mnoho) malých služeb. Povšimněme si, že streamovací servery přehrávájí video soubory pro více StreamEditů, nemají ale informace o assetech, pouze o jménech těchto video souborů. To znamená že statistiky mohou být nepřesné, pokud nelze z názvu souboru identifikovat příslušnost ke konkrétnímu StreamEditu. Chyba je v tomto případě v nedostatečném zapouzdření „služby“ streamovacích serverů, které toho „vědí příliš málo“.

Hlavní je správně identifikovat služby a jejich účel, ne absolutní neměnnost jejich rozhraní (jak byla mimochodem navrhována vedením Noteriku pro StreamEdit). Je důležité neměnit technologii (protokol), jímž se na rozhraní přistupuje, protože to vyžaduje složitou rekonstrukci volajících služeb, přidání nové operace však žádný problém není. Je třeba pouze zajistit, aby se služby nepřekrývaly ve funkcionalitě a aby byly podporovány všechny minulé verze rozhraní, které jsou stále někým používány. Je-li služba malá, toto by neměl být výrazný problém. Také je důležité, aby každá služba měla správce svého rozhraní, které pak může zůstat konzistentní i při množství různorodých požadavků na něj. Jsou-li navíc přítomny generické operace jako v případě těch v aplikaci Stats, nemusí být případně zdlouhavější proces přidání nové operace (kvůli formálnímu požadavku na správce rozhraní) na závadu. V případě nutnosti implementovat danou funkcionalitu velmi rychle například kvůli termínu dodaní, se může implementace dočasně umístit na straně služby, která vznesla požadavek na změnu. Nakonec by se ale měla přesunout do systému, který je na danou věc specializovaný, aby byl zachován požadavek na samostatnost a abstrakci služeb.

8 Zkušenosti firem s nástroji na správu požadavků

V této sekci zhodnotíme praktické zkušenosti s prostředky specifikace požadavků. Fakta vycházejí z osobních zkušeností autora z práce na novém informačním systému

Univerzity Karlovy (přehledně popsaného v [51]) a projektech holandské internetové firmy Noterik, dále pak z odpovědí pracovníků firem na autorovy anketní otázky. Otázky se týkaly jak obecně prostředků správy požadavků, tak servisně orientované architektury a jejího návrhu.

Odpovědi na anketní otázky jsou k dispozici od Jana Pavelky z DCIT, který se také podílel na počátcích vývoje informačního systému Univerzity Karlovy (vedl tým, jehož úkolem byl sběr a analýza požadavků a zpracování studie proveditelnosti), senior analytika Tomáše Rubače z Arbes Technologies, analytika Davida Bílka z Unicornu, rakouského business analytika Andrew Kirknessa pracujícího jako kontraktor pro více různých firem (z větších např. ANZ, Telstra, Yarra Valley Water, dále pak pro menší internetové konzultantské společnosti) a také od více pracovníků z Philips Medical Systems a Philips Semiconductors (například Reného Krikhaara a Hanse Thelosena, kteří přímo na otázky ankety neodpovídali, ale bylo s nimi o problematice diskutováno). Ačkoliv množství odpovědí zdaleka nestačí na vytvoření zásadnějších statistik, různorodost vzorku a zkušenosti dotazovaných mohou dát dobrou představu o aktuální situaci v oblasti specifikace požadavků, k ní používaných nástrojů a servisně orientované architektury. Znění otázek i kompletní odpovědi lze nalézt v příloze 1.

Jak poznamenává Jan Pavelka (a podle něj s ním souhlasí i slavný profesor Parnas momentálně působící na irské University of Limerick), zásadní problém specifikace požadavků je komunikace se zákazníkem, tedy činnost, kde softwarové nástroje pomoci nemohou nebo mohou jen omezeně. Přesto se v průmyslu používá mnoho různých prostředků specifikace požadavků, ať už vyvinutých interně pro danou firmu nebo obecných od specializovaných výrobců.

Například ve Philips Semiconductors přecházejí v době psaní této práce z vlastní aplikace na sběr požadavků postavené na Microsoft Accessu k průmyslovému lídru Telelogic DOORS. Ačkoliv vlastní aplikace měla přinést flexibilitu ve výběru podporované funkcionality, ukázalo se, že program je vždy pozadu za tím nejlepším od specializovaného výrobce, od nějž je to navíc možné získat mnohem pohodlněji (také překvapivě často rychleji, obzvlášť v případech, kdy je nově požadovaná funkcionalita v jejich produktu již přítomna) a tedy celkově za nepříliš se lišící cenu. Díky obecnosti a tím i rozsáhlé konfigurovatelnosti nástroje DOORS není výhoda vlastního nástroje natolik znatelná, v podstatě veškerou požadovanou funkcionalitu je možné najít i v od někoho jiného zakoupeném produktu. Ve Philips Semiconductors, především v obchodní divizi Home (kde působí zpovídaný Hans Thelosen), se v posledních letech vývoj softwaru začíná výrazně orientovat na znovupoužitelnost komponent a jejich multilokalitní programování. V takovém prostředí je velmi důležité být schopen spravovat požadavky a žádosti o změnu natolik pokročile, aby se efekt odsouhlasených změn dal vystopovat až k samotném kódu a jeho jednotlivým verzím. Kromě DOORS bude tedy velmi využívaná i jeho vazba na Telelogic Synergy, který ve Philipsu již používají pro změnový a konfigurační management. Možnost požadavky na sebe vázat a sledovat jejich dopad na další fáze softwarového vývoje je nejčastěji zmiňovaná požadovaná funkcionalita. Tomáš Rubač, který používal pouze pro jeho firmu vyrobený proprietární evidenční systém, uvádí, že specifikační software pro něj znamená zásadní zkvalitnění výsledného systému; a to kromě očividné schopnosti vědět, co bylo slíbeno (a vazbě na plán, takže lze brzy zjistit, je-li vývoj nějaké požadované části zpožděný), také díky znalosti, co a jak spolu souvisí. Software je však třeba umět správně používat a využívat, Jan Pavelka si vzpomíná na několik projektů, v nichž působil v roli externího konzultanta, kdy velmi drahý systém na správu požadavků nebyl využíván pro složitější činnosti, než na které by nestačil běžný tabulkový procesor.

A překvapivě často žádný specializovaný software pro podporu specifikace požadavků používaný není. Dotazovaní specialisté velmi často zmiňují, že mnoho projektů, na kterých

pracovali, specifikovali pouze za pomocí textového editoru, případně tabulkového procesoru pro evidenci vazeb a přehledová zobrazení. Z odpovědí lze cítit, že tzv. zakopaný pes je podle expertů z praxe někde jinde: Již zmiňovaný výrok Jana Pavelky klade důraz na správnou komunikaci se zákazníkem, David Bílek pak za klíčové považuje „správné nastavení životního cyklu a donucení okolí požadavky evidovat a řešit“. Andrew Kirkness, ačkoliv pracoval na rozsáhlých projektech pro velké firmy, si dokonce používání specializovaného nástroje neumí představit. Používá pro každou fázi textový dokument, vazby mezi požadavky pak eviduje v tabulkovém procesoru: začne se základními požadavky na vysoké úrovni (například definující hranice systému), pak zaznamená, jak se tyto požadavky vážou na tradiční obchodní požadavky a ty pak na funkční požadavky na systém. Tento způsob práce mu vyhovuje natolik, že prohlašuje, že si „nedokáže představit používat specializovaný software“ a že „někdo by musel pracovat velmi tvrdě, aby ho o nákupu některého přesvědčil“, Jako hlavní vlastnost programů na podporu specifikace požadavků pak dokonce uvedl „nadbytečnou práci navíc“. Zřejmě to není ojedinělý názor, neboť i Jan Pavelka uvádí, že pro zaznamenání úvodní specifikace požadavků natolik rozsáhlého projektu, jakým byl nový informační systém Univerzity Karlovy, byly použity listy tabulkového procesoru pro přehledovou evidenci a textové dokumenty pro podrobné rozepsání specifikace jednotlivých komponent a aplikací s odkazem do přehledu v tabulkách. Specifikace ve formě textových dokumentů zažil i autor práce při práci na jedné z komponent tohoto informačního systému (v tomto případě se jednalo o podrobný technický popis používaných algoritmů), i při třech internetových projektech v nizozemské firmě Noterik (tentokrát byly v dokumentu stručně sepsány uživatelské scénáře užití). I výzkum ITEA projektů z let 2002 až 2004, který zjišťoval informace o prostředcích specifikace požadavků od více jak 100 projektů z oblasti software pro embedded zařízení, z nichž více jak 60% čítalo víc jak 100 000 řádek kódu, ukazuje, že v oblasti specifikace požadavků není používání formálních metodik a specializovaného softwaru příliš časté [52]. Podle jeho výsledků je polovina zkoumaných produktů vyvíjena bez metodiky pro správu požadavků a pouze 25% používá pro tyto účely nějaký specializovaný software. Téměř polovina pak používá právě textový editor Microsoft Word [52, 15].

Zajímavým fenoménem je používání softwaru na nahlašování chyb a správu změn pro evidenci požadavků. Čím kratší iterace prodělává vývojový cyklus, tím se totiž zmenšuje rozdíl mezi požadavkem na změnu (případně hlášením chyby) a požadavkem. Při dlouhém cyklu a v ideálních podmínkách se shromáždí požadavky, systém se navrhne a vytvoří a předá. Poté následují požadavky na změny a případně nové požadavky. Rychlá změna je ale synonymem dnešní doby, a tak požadavky na změny ve stávající specifikaci přicházejí často ještě dříve, než se vůbec stačí implementovat. Požaduje-li zákazník ve zpola vytvořeném produktu přidání nové položky do menu, je to změnový požadavek, nahlášení chyby (jedná-li se například podle zadavatele o samozřejmou funkcionalitu) nebo nový požadavek? Systémů pro správu změn je dostupná celá řada, velmi často jsou to levné (nezřídka jsou úplně zdarma) a lehce nainstalovatelné webové programy. Webové rozhraní znamená velmi důležitou výhodu, pokud chceme výrazně zapojit našeho zákazníka do testování či schvalování vytvářeného produktu – ten si totiž nemusí nic instalovat ani nastavovat, jednoduše se nahlásí přes námi vytvořený účet a může software ovládat, nejčastěji tedy zadávat nové připomínky.

Autor práce má s těmito softwary zkušenosti opět z práce v obou zmíněných prostředích. Při vývoji informačního systému Univerzity Karlovy používali zadavatelé pro komunikaci s dodavateli systém RT: Request Tracker [53]. Produkt umožňuje základní funkcionalitu takových systémů: zadat novou žádost, určit její prioritu a závažnost, žádosti pak různě zobrazovat, filtrovat a přiřazovat různým osobám. Některé ze záznamů v databázi RT pro informační systém UK lze považovat za hlášení chyb, některé za požadavky na změny, jiné jsou ale úplně nové dosud nevyslovené požadavky, některé i na poměrně vysoké

úrovni (jaké by se daly očekávat v některé z úvodních specifikací). Není ale na škodu, jsou-li všechny tyto záznamy na jednom místě – všechny v podstatě znamenají totéž: úkol pro některého z pracovníků. Problematické může být pouze účtování takové práce – rozhodnutí, jestli vyřešení dané žádosti bude zaplaceno zvlášť nebo je v rámci dříve dohodnutého objemu prací. To ale není výrazná překážka tomu mít tato data na jednom místě.

I při práci na webových projektech ve firmě Noterik byl používán obdobný systém: webový Mantis určený na správu chybových hlášení tam byl využíván pro vedení více než právě jen chyb [54]. I zde se často objevovaly úplně nové požadavky, o kterých se pak například muselo vyjednávat, byly-li pokryty úvodní specifikací a tedy i rozpočtem nebo znamenají-li nové náklady. Zde se ukázaly i možné nevýhody takového přístupu. Systém se totiž používal jednak pro komunikaci se zákazníky, pak také ale jako interní úkolový systém a to nad stejnými záznamy. Zákazník tedy zadal svým jazykem formulovaný požadavek a čekal na stejně technické úrovni formulovanou odpověď, mezitím se stejný požadavek ale používal pro interní komunikaci – vyjadřovali se k němu programátoři, architekti a jiní techničtější pracovníci (například řešili vybraný způsob implementace nebo nastalé potíže) – což by mělo být zákazníkovi skryto, protože ho to zbytečně mate. Dobré by tedy bylo oddělit interní úkoly od zákazníkových (tj. odvozovat ze zákazníkových požadavků zadání pro pracovníky). Pouhé oddělení diskusí k požadavkům není dostatečné řešení, neboť žádosti mají i svoje další atributy (stav, priorita, odpovědná osoba apod.) a je žádoucí, aby se mohly nastavovat zvlášť pro oba typy požadavků.

Že používání systémů na správu chyb a změnových požadavků jako prostředek specifikace není ojedinělá praxe dokazuje i odpověď Davida Bílka z Unicornu, který na první anketní otázku (jaké softwary na správu požadavků jste používali) vyjmenoval IBM Rational ClearQuest, Atlassian JIRA a Compuware TrackRecord, kde hlavní účel všech je právě správa defektů a změn v již vytvořeném software.

Odpovědi na otázky týkající se servisně orientované architektury odrážejí stav tohoto konceptu v praxi: na svoje větší rozšíření (pokud nastane), stále ještě čeká. Například Tomáš Rubač se necítil fundovaný otázky o SOA vůbec zodpovědět, ačkoliv firma Arbes Technologies, ve které působí, vyvíjí mnoho systémů pro podporu fungování firem, mimo jiné i ERP systém pro střední a velké firmy. I Jan Pavelka a David Bílek, kteří byli schopni připojit komentáře na toto téma, nemají s praktickým návrhem servisně orientované architektury zkušenosti, nemohli tedy připojit ani žádná doporučení na specifikační software nebo jeho požadovanou funkcionalitu. Odpověď Jana Pavelky na šestou otázku ankety (myslíte, že SOA vyřeší palčivé problémy ICT sektoru) možná charakterizuje postoj firem k tomuto konceptu: Podle něj velmi záleží na tom, jak se princip SOA ujme. K tomu nadějně cituje agenturu Gartner, podle které by se mohla SOA stát standardem pro vzájemnou komunikaci aplikací a vytváření jejich rozhraní do roku 2008. Pokud předpověď vyjde a koncept se ujme, zjednoduší to podle Pavelky život výrobcům software, protože integrace již nebude tak bolestivá, jak je tomu v těchto dnech. Zdá se, jako by firmy opatrně čekaly, jestli se SOA ujme a až pak ji začnou také implementovat. Proč ale mohou být firmy skeptické mohou naznačit odpovědi Davida Bílka z Unicornu. Ačkoliv je podle něj koncept SOA založen na správných principech, je velmi obtížné jej uvést do praxe, problémy jsou prý především ve větších organizacích, kde je obchodní logika firmy rozprostřena ve velkém množství aplikací (jenž se často svou funkcionalitou překrývají) na různých platformách. Nejprve je tedy třeba standardizovat některé operace, vyjasnit procesy a to může být v takových organizacích komunikačně náročné. Ačkoliv podle něj SOA řeší problémy spojené se znovupoužitelností a škálovatelností firemních aplikací, je náročné jí uzpůsobit hardwarovou architekturu. Náročné je prý také řízení a správa požadavků na jednotlivé prvky architektury a jejich samotný provoz, je-li jejich funkcionalita vyžadována mnoha stranami. K návrhu David Bílek doporučuje vytvořit stabilní specifikaci požadavků na architekturu a

rozhraní jejich prvků a také hardwarovou infrastrukturu.

9 Doporučení na prostředky specifikace požadavků

Tato kapitola shrnuje tuto diplomovou práci a vyjadřuje praktická doporučení na správu požadavků a prostředky k tomu používané. Protože specifikace je úvodní fází životního cyklu softwaru, zásadně ovlivňuje celý jeho zbytek. Doporučené praktiky tedy nutně musí přesahovat pouze tuto fázi a souviset se všemi ostatními. Hlavním kritériem při výběru pracovních postupů a jejich softwarové podpory bylo samozřejmě úspěšné dokončení projektu – tedy spokojený zákazník, který obdržel produkt podle dohodnutých podmínek.

9.1 Obecná doporučení

A přemýšlet o specifikaci požadavků, chceme-li vylepšit způsob, jak vyvíjíme software, není od věci. CHAOS Report z roku 1994 od The Standish Group analyzující data od více jak osmi tisíc projektů ukazuje, že pouhých 16,2 % z nich bylo úspěšných a celých 31,3 % bylo dříve či později úplně zrušeno [55]. Podíváme-li se na důvody úspěchu, nalezneme na prvním místě zapojení uživatelů (15,9%), na druhém pak podporu vyššího managementu (13,9%) a na třetím již jasně specifikované požadavky s 13% (ostatní faktory pak mají již větší procentní odstup, začínají na 9,6%). Tři hlavní důvody, proč byly projekty zdrženy nebo stály více, než bylo dohodnuto, zahrnují: nedostatečná zpětná vazba od uživatelů (12,8%), nekompletní požadavky a jejich specifikace (12,3%) a měnící se požadavky a jejich specifikace (11,8%). Důvody pro úplné zrušení projektu pak mají dokonce nekompletní požadavky jako první na seznamu s 13,1%, následuje opět nedostatečné zapojení uživatelů s 12,4%, třetím faktorem jsou pak nedostatečné zdroje s 10,6%. Již toto jsou velmi znepokojivá čísla, stejný průzkum ale od manažerů příslušných firem zjišťoval, jestli podle nich v daném roce selhávalo více projektů než před pěti nebo deseti lety. 46% z nich uvedlo, že projekty selhávají výrazně nebo alespoň o něco více než před deseti lety, pro období před pěti lety pak dokonce 48% [55, 2]. To se skoro dá vykládat jako „myslíme, že situace je velmi špatná a pořád se zhoršuje“.

Podíváme-li se na hlavní příčiny úspěchů a neúspěchů, téměř všechny můžeme shrnout jako špatnou komunikaci. Nedostatečné zapojení uživatelů i nevyhovující specifikace naznačují, že v organizacích je často mnoho vrstev managementu a jiných pracovníků (například oddělení uživatelské podpory outsourcované z úplně jiné země). Těmito vrstvami, kde má každý její člen odlišný pohled na projekt a technické či obchodní znalosti, se pak jen obtížně požadavky zadavatelů předávají nezkreslené. Pokud výrobce softwaru realizuje tradiční dlouhý vývojový cyklus, může specifikace požadavků zabrat netriviálně dlouhou dobu. Mezitím se ale požadavky stále vyvíjejí, stejně jako v době, kdy se obsah specifikace začne realizovat. Takovou změnu je třeba provést ve všech fázích cyklu – je třeba změnit specifikaci požadavků, návrh (neřkuli architekturu), kód i testy, které ověřují jeho správnost, všechny uživatelské dokumentace, případně také způsob nasazení produktu. A správně implementovat všechny změny je v rozsáhlé organizační struktuře, která často zahrnuje pracovníky mnoha firem, velmi komplikované. Když pak po dlouhé době mají uživatelé možnost vidět první verze produktu, mohou být velmi překvapeni, jak se jejich očekávání lišila od předkládané reality. Velmi často od lidí z praxe (analytiků/architektů i uživatelů) slýcháváme, že velmi zřídka panuje s počítačovými systémy velká spokojenost, často implementovaná řešení zklamávají očekávání. Pro autora této práce to znamená jediné: takové projekty jsou pravděpodobně příliš složité. Pokud očekávání velmi přesahují reálný výsledek, je možná požadavků v jednom okamžiku příliš mnoho bez testování a potřebné zpětné vazby od budoucích uživatelů.

Autor může těžit z velmi cenné zkušenosti, totiž z podílení se na projektu, který selhal

(alespoň v původně zamýšlené verzi). Původně navrhované řešení nového informačního systému Univerzity Karlovy se po několika letech vývoje označilo za nepoužitelné, požadovaná funkcionalita bude implementovaná jiným způsobem. Podle již citovaného Jana Pavelky je hlavní příčinou neúspěchu náročnost technické koncepce řešení. Autor by rád připojil svůj úhel pohledu: podle něj je tento projekt jeden z mnoha systémů, které se příliš dlouho vyvíjely, aniž by implementovaná funkcionalita byla ověřena, ať už automatickými nebo manuálními testy, ještě lépe však samotnými uživateli. Ačkoliv autorovo nejbližší vedení na tuto skutečnost upozorňovalo, byl například v centrálním aplikačním serveru (tedy části, na které se autor spolupodílel) dva roky kód, který nikdy nebyl spuštěn někým jiným než jeho autorem, tedy řádně ověřen. To ukazuje, že se příliš mnoho požadavků napsalo na papír a implementovalo, aniž by opravdu existovala aktuální potřeba je ve vytvářeném systému mít. Zdá se, že systém byl poněkud přespecifikovaný. O vhodnosti návrhu celkové architektury se autor necítí fundován vyjadřovat, faktem ale je, že rozhraní jejich prvků se v průběhu vývoje často měnila, což způsobilo, že i menší změny v požadavcích zadavatelů výrazně ovlivňovaly mnoho komponent celého informačního systému. Kvůli rozsáhlosti projektu pak je pro sebeznalejšího a kompetentního projektového manažera nemožné detailně znát všechny jeho části. Pokud ale změny požadavků vnitřní fungování všech částí ovlivňují, je třeba jednotlivým vývojářům buď požadavky předat, čímž může dojít k jejich dezinterpretaci, nebo ustavit komunikaci zákazník-vývojář, což však může narazit na mnoho dalších problémů (vývojář nemá celkový obrázek, používá jinou terminologii apod., úspěch takové komunikace tedy příliš zřejmý není). Podíváme-li se na čísla z průzkumu The Standish Group uvedených výše, můžeme snadno hádat, že tato situace není ojedinělá.

Bylo by bláhové se pokoušet ustanovit nějaký nový způsob komunikace, který by naznačené problémy vyřešil. Protože taková komunikace by musela zahrnovat velké množství pracovníků různých profesí a kompetencí, stejně tak pravidla pro různé činnosti v průběhu projektu, nemohlo by to být vyřešeno jednoduchou sadou pravidel (ostatně v takovém případě by na to pravděpodobně již dávno někdo přišel). A vytvořil-li by se velký politický dokument, jak šířit informace, málokdo by ho podrobně četl, natož dodržoval. Další zdánlivé řešení, totiž že se vše přesně vyspecifikuje a vykomunikuje se zákazníky na začátku a pak se již jen implementuje (kdy jednotlivé vrstvy pracovníků mají dostatečné znalosti na to, aby fundovaně vedly ty, kteří na ně navazují), je ještě větší utopie. Například proto, že požadavky se mohou v průběhu projektu měnit ne proto, že si zákazník právě vzpomněl, co vlastně chce (i když i to je možné), nebo že se analytikům konečně podařilo najít s ním společnou řeč, ale vlivem vnějších faktorů, například změnou situace na trhu. Změna je nevyhnutelná, otázkou je, jak se s ní vypořádávat.

Chceme-li zůstat dostatečně organizovaní, být v těsném kontaktu se zákazníkem a díky dobrému přehledu o projektu velmi přesně rozumět jeho požadavkům (tedy minimalizovat hlavní důvody neúspěchu z citovaného výzkumu), musíme zásadně zmenšit velikost projektů a týmů na nich pracujících. Malý tým (tedy 2-7 pracovníků) nemusí používat příliš složité metody pro vzájemnou organizaci a komunikaci (spíše se nemusí pokoušet je používat, často je to pouze tužba). V malém týmu se zadavatelem může komunikovat přímo vývojář dané části vyvíjeného systému (požadavky tedy nejsou nikým interpretovány), tento vývojář je ale v případě malých projektů schopen vědět jak celkový účel projektu, tak mít velmi dobré ponětí o jeho struktuře a je pro něj tedy snadnější najít s uživateli společnou řeč (nakonec i se svými kolegy, na rozdíl od rozsáhlých týmů je zřejmé, s kým si vyjasnit které části systému). Volání po menších týmech a projektech není ojedinělý hlas autora práce. Podíváme-li se na další veřejně dostupnou část CHAOS studie The Standish Group, tentokrát z roku 1999, zjistíme, že podle jejich výsledků velikost projektů je nepřímo úměrná jejich úspěšnosti [56]. Protože se ve studii opět dočteme obdobné příčiny úspěchů projektů, totiž zainteresovanost uživatelů, podpora managementu a jasně stanovené obchodní

cíle projektu, je závěr „receptu“ očekávatelný: nemějte projekt delší než šest měsíců, dražší než
750 000 dolarů a větší než šestičlenný tým. Výhody menších týmů ověřilo mnoho dalších studií, například výzkum společnosti QSM zpracovávající údaje o 491 projektech, který ukázal, že velikost týmu je nepřímo úměrná produktivitě a přímo úměrná pravděpodobnosti zpoždění i úsilí vynaloženému na stejnou činnost [57]. Podobné názory se doslechneme i od jednotlivých firem z praxe, například od Jasona Frieda ze stále úspěšnějších 37signals přinášejících nové myšlenky na poli softwarového inženýrství [58].

Rozhodnutí, zda použít agilních metod vývoje nebo těch tradičních není až tak jednoznačné. Podle autora práce to závisí na účelu projektu a již osvojených způsobech práce vývojového týmu. I menší projekty často mají mít své rozpočty a plány, které je třeba na něčem postavit. Pokud nějaká firma vyvíjí software na zakázku, je třeba se zákazníkem dohodnout cenu, typicky před tím, než začne s implementačními pracemi. Takovou cenu je ale třeba podle něčeho spočítat: je třeba odhadnout, kolik pracovních sil a na jak dlouho bude třeba na realizaci projektu vynaložit; a k tomu je nutné dopředu vědět – alespoň rámcově – co vše bude do projektu patřit. Menší velikost systému takovou činnost nicméně zjednodušuje. Dobrým přechodem od tradičních k agilnějším metodám by mohlo být nejdřív vyspecifikovat velmi základní funkcionalitu do první verze, tu i implementovat a nechat si schválit (tj. de facto zaplatit) a poté takto inkrementálně pokračovat při rozšiřování funkcionality a vytváření verzí dalších. Riziko by se tak výrazně snížilo, neboť pokud by z nějakého důvodu první prototyp nevyhovoval, neutratilo by se zbytečně příliš mnoho peněz (jako by tomu bylo v případě nevyhovujícího celého systému), implementované řešení by pak navíc mohl použít případný druhý vybraný dodavatel. A pokud by selhal jeden z dalších kroků, ztráta by opět nebyla závratná, jen by se v další iteraci postupovalo zkušeněji správnějším směrem. Jak se zmenšuje rozdíl mezi úvodním požadavkem a žádostí o změnu (díky zkracujícím se vývojovým cyklům, jak bylo ukázáno výše na příkladech z praxe), je důležité být neustále připraven vyvíjenou aplikaci změnit. Při dobré koordinaci se zákazníky by to nemuselo ani znamenat příliš mnoho papírování navíc. Možností je i najmout od dodavatele celý vývojový tým [11], což by mohlo snížit administrativní zátěž, není to však jisté (záleží, jak konkrétně by byl ustaven smluvní vztah). Problematické může také v tomto případě být získávání informací, kdy se zdánlivě zdá, že najatý pracovník má díky přítomnosti uvnitř firmy všechny potřebné informace a není tedy třeba, aby mu byly dojednány managementem nebo poskytnuty v dokumentaci, což často vede ke zmatkům a neschopnosti outsourcovaných vývojářů dokončit svoji práci (jak se autor dozvěděl od podobně pracujících konzultantů a programátorů firem jako NextiraOne nebo Accenture).

Pravděpodobně o něco komplikovanější je použít agilní metody při vývoji kritických aplikací pro konzervativní zákazníky; i to je však možné (jeden z pokusů – z Hewlett Packard – lze nalézt v [59] i s vyjádřenou autorovou lítostí nad stále příliš dlouhou fází sběru požadavků a nadbytečné modelování v UML a uspokojení s brzkým vytvořením prototypu projektu). Požadavek na menší týmy a projekty ale platí i pro tyto projekty.

Ne všechna zadání ale lze vyřešit malým projektem s malým týmem, velké úkoly si žádají komplexnější řešení. Jak ale spojit schopnost řešit komplexní zadání s požadavkem na malé projekty? Autor práce věří, že odpovědí na tuto otázku je servisně orientovaná architektura. Díky ní je možné i rozsáhlý systém rozdělit na mnoho nevelkých služeb, které je možné vyvíjet dostatečně odděleně. Protože SOA přináší standardizaci komunikačního rozhraní nezávislou na platformě, jednotlivé služby mohou být snadno vyvíjeny i různými firmami (například specialisty na danou oblast, velmi častou praxí je dnes kontraktování subdodavatelů). Jak bylo ukázáno výše v této práci, je možné použít agilní postupy i pro návrh celkové architektury, tedy především pro rozdělení funkcionality mezi jednotlivé služby a definování jejich rozhraní – formálních kontraktů. Tato činnost může být prováděna

nevývojáři, lidmi zaměřenými více na obchodní cíle společnosti, což je výhodné, protože – jak bylo také již uvedeno – velmi důležité je zapojení vyššího managementu do této činnosti. Díky použití standardů pro vytvářený middleware (typicky webových služeb), je možné rozhraní definovat z uživatelského pohledu, jeho konkrétní implementace do příslušné technologie (např. poskytnutým WSDL popisem služby) by se měla řídit celosvětovými standardy, které nebude třeba s implementátory vykomunikovávat. Standardizace infrastruktury je další částí již zmíněného doporučení pro úspěšné projekty od The Standish Group. Výzkumníci tvrdí, že podle jejich zjištění 70% aplikačního kódu přísluší právě infrastruktuře [56, 4]. Díky její standardizaci se dají bez problému použít již hotové komponenty a wrappery dostupné pro mnoho programovacích jazyků a platforem a délka a složitost vývoje (a tedy i cena) se tak mohou výrazně snížit. Je důležité vyzdvihnout, že servisně orientovaná architektura výrazně zvyšuje izolovanost týmů vyvíjejících její jednotlivé části (a tím snižuje rozsah zdržující a zmatek přinášející koordinace). Kvůli nejasně definovanému rozhraní centrálního aplikačního serveru informačního systému UK i ostatních komponent specifikace vnitřního fungování těchto částí výrazně ovlivňovala subsystémy ostatní. Vývojáři tak de facto museli číst a rozumět specifikacím částí, se kterými jimi implementované komponenty komunikovaly, protože rozhraní nebyla definována jasně „svrchu“, ale byla do značné míry vytvářena spolu s danou komponentou. To, že se rozhraní v průběhu prací mění, je nevyhnutelné, důležité je ale stanovit jisté hranice, především pak technický způsob komunikace mezi službami. Případová studie z Noteriku to ukazuje, změny WSDL popisu služby problémem nejsou, komplikace přinašejí změny přístupových metod. Je-li navíc přístupová metoda definována jasným světovým standardem, množství nedorozumění může být minimální. Ačkoliv byla specifikace centrálního aplikačního serveru přesná a jasná, byla pro tradiční vývojáře obtížně pochopitelná (tým učitelů a studentů MFF UK byl nucen komunikovat s pracovníky s odlišným přístupem k formalismům a vůbec způsobem práce) a jen těžko byli schopni implementovat přesně vše nutné pro správnou komunikaci. Pokud by se pro rozhraní použilo více standardních – tj. nástroji a jinou dokumentací podporovaných – metod a definovalo se „shora“ (tj. více z uživatelského pohledu), spolupráce by nejspíš nebyla tak komplikovaná.

Můžeme si tedy povšimnout, jak i rozsáhlé projekty mohou být realizovány malými izolovanými týmy, jejichž komunikace nepředstavuje nadměrnou zátěž a zbrzdění. Ačkoliv tým vytvářející architekturu musí sbírat požadavky z velkého množství různých zdrojů, jeho velikost nemusí být závratná. A pokud by velikost byla opravdu enormní, pravděpodobně by nebylo příliš obtížné i tuto činnost rozdělit na dostatečně disjunktní oblasti. Služby by pak měly být navrženy tak, aby je bylo možné vytvářet rychle v malých týmech (například s využitím sdružováním služeb), s výhodou pak s použitím agilních metod.

Mají-li služby kromě povinnosti implementovat formální kontrakt i jiné uživatele (tj. je-li implementováno uživatelské rozhraní, v opačném případě jde pouze o vyhovení kontraktu a vnitřní fungování služby, při kterém jsou si vývojáři sami zákazníky a místo revize uživatelů využívají pouze testů), je třeba s nimi být v úzkém a neustálém kontaktu. (Je nutné podotknout, že subjekt specifikující formální kontrakt představuje pro implementátory služby také zákazníka.) Místo specifikování všeho dopředu autor doporučuje postupovat menšími cykly, kdy na konci každého je *všechna* funkcionalita otestována a schválena zákazníkem. Takto těsná spolupráce zajistí, že bude implementováno přesně a správně to, co uživatel požaduje, a nic navíc. Krátké cykly také udržují motivaci týmu [59]. Každý vývojář (ostatně kdokoliv, který má stanovený termín dokončení čehokoliv) ví, že těsně před odevzdáním projektu efektivita výrazně roste – a to neznamená pouze to, že by se pracovalo více hodin, ale především to, že práce je najednou daleko víc orientována na výsledek, více se programuje a méně rokuje a odkládá. Díky krátkým cyklů je reakce na změnu daleko rychlejší a změněné nebo postupně se vyvíjející požadavky je možné implementovat levněji. Je také

důležité poznamenat, že zkracování cyklů je umožněno vývojem programovacích jazyků a integrovaných prostředí, stejný program v moderním jazyce je možné vyvinout daleko rychleji než před jen několika lety a dá se očekávat, že tento vývoj bude jen pokračovat. Přeprogramování části systému již nebude velmi časově náročné v situaci, kdy je mnoho kódu generováno buď z definic uživatele (například z modelů systému – např. v UML, navíc srovnej MDA v [60]) nebo díky definovaným konvencím a standardům.

Shrnuto, aby byly požadavky ve výsledném systému implementovány podle představ zákazníka, je doporučováno dodržovat následující zásady:

1. Tvořit maximálně sedmičlenné týmy pracující na co nejmenších projektech.
2. Produkty vyvíjet v úzké vazbě na uživatele (zákazníka, zadavatele) a v krátkých cyklech, na konci každého pak veškerou funkcionalitu ověřit testy nebo nechat schválit. Je tedy žádoucí snížit fázi specifikace požadavků na minimum a místo toho ji několikrát opakovat a požadavky revidovat.
3. V případě rozsáhlých systémů vytvořit malý tým mající na starost vývoj servisně orientované architektury a tím rozdělit komplexní problém na více menších úkolů. Jedná-li se o již existující infrastrukturu, lze použít popsaných agilních metod pro její postupnou přeměnu.

9.2 Doporučení na specifikaci požadavků a jejích prostředků

Pohled na specifikaci požadavků je odlišný pro celkovou architekturu a pro její prvky. Architektura – tedy celková koncepce informačního portfolia – je ovlivňována mnoha faktory, je třeba získávat informace z mnoha zdrojů, alespoň v úvodu. Jak naznačuje David Bílek citovaný v předchozí kapitole a jak zaznívá i z úst jných praktiků ([61]), zavedení SOA není ani tak technický jako spíše organizační problém. V případě velkých organizací je i při použití agilních metod v jednom okamžiku ve hře mnoho různorodých požadavků, které je třeba shromáždit a pečlivě evidovat. Jedná se o velmi obecné požadavky, popisující více potřeby firmy než nároky na systém (v rámci specifikace zde tedy mluvíme o fázi zahájení). Při vytváření úvodních představ se u velmi rozsáhlých architektur může vyplatit používat software pro vytváření modelů systémů podle zadaných obchodních potřeb typu CASEwise Corporate Modeller (použitý pro úvodní fáze vývoje informačního systému Univerzity Karlovy) nebo Telelogic FocalPoint. Tato fáze není tak podstatná v menších organizacích (případně pro úkoly menšího rozsahu), kdy se nevelké množství požadavků může evidovat v textových souborech apod.

Pro samotný návrh architektury pak již v části o nástrojích pro specifikaci SOA byl doporučen UML diagram nasazení, případně přímo diagram pro softwarové služby z rozšiřujícího profilu IBM. Kromě těchto přehledových diagramů je třeba specifikovat formální kontrakty služeb a další funkční požadavky na systém, například kdo bude využívat kterou službu, kde má fyzicky běžet apod. Zde opět záleží na velikosti řešení, má-li se pro tyto požadavky použít specializovaný software nebo stačí poznámky k jednotlivým elementům diagramu, případně separátní dokument. Místo dokumentu by autor doporučoval dokumentovat přímo elementy diagramu, pokud to náš UML nástroj jen trochu podporuje, s více dokumenty roste riziko nekonzistence. Pro velká řešení se pak vyplatí pořídit si specializovaný nástroj na správu požadavků, autorovo doporučení směřuje k Telelogic DOORS, který v rozšíření DOORS/Analyst umožní v jednom prostředí vytvářet diagramy i na ně klást požadavky se všemi výhodami, které automatizované prostředky specifikace přinášejí. Toto řešení je i dostatečně škálovatelné, pokud by se menší architektura začala později výrazněji rozrůstat. Lze využít i vazby na prostředky řízení konfigurace, například pro ukládání různých verzí WSDL definic rozhraní. Vždy ale záleží na v organizaci již využívaných produktech; podle nich by se měla používaná rodina produktů doplňovat, aby

byla navzájem dobře propojitelná, je-li to vyžadováno. Naštěstí mnoho produktů specifikace požadavků lze napojit na velké množství UML nástrojů, výběr by tedy měl být dostatečný.

Jednotlivé služby by pak měly být natolik malé a oddělené, že by mělo být možné vyvíjet je za použití agilních metod s vysokou orientací na uživatele (ať už někoho, kdo definuje rozhraní služby nebo osoby, které ke službě přistupují přes její uživatelské rozhraní). Fáze specifikace požadavků by se měla zkrátit na minimum, zásadním stadiem v opakovaných cyklech by měl být částečně vývoj (např. zkušebních prototypů), především pak testování (je-li funkčnost systému formálně definovaná) a zpětnou vazbu uživatelů, kde poslední dvě fáze vyprodukují požadavky pro fázi další (tj. žádosti o opravy chyb nebo rozšíření systému). Takto hrubé požadavky bude potřeba rozpracovat a zvážit, budou-li se ve skutečnosti implementovat. Obecné požadavky na službu ze specifikace architektury by neměly být natolik restriktivní, aby tento postup znemožnily. Požadované rozhraní by mělo být zesložiťováno postupně, tak jak se prototypy aplikace stávají vyspělejšími.

Úvodní specifikace tedy vychází z nástrojů, které byly použity pro specifikaci architektury, k vývojářům služby se pak dostanou typicky jako textový dokument (jako report vytvořený ze specifikačního software nebo přímo UML nástroje). K tomu přibude krátký popis iniciální implementace služby, tedy krátké rozšíření tohoto dokumentu. Další požadavky již přicházejí v průběhu vývoje od mnoha různých osob, je tedy třeba udržet v nich pořádek alespoň natolik, aby se na žádný nezapomnělo reagovat. Protože – jak bylo poznamenáno – se v malých týmech rozdíl mezi požadavky na rozšíření a hlášení chyb smazává, vše, co potřebujeme, je software na správu chyb – tím disponuje i sebemenší vývojový tým a ten za těchto okolností nabízí vše, co potřebujeme i pro požadavky na změny a rozšíření. Například systém Trac umožňuje nadefinovat si vlastní typy hlášení, je tedy konfigurovatelný pro použití pro více typů požadavků na nějakou činnost [62]. Protože Trac má navíc integrovanou wiki, rozhraní na úložiště kódu Subversion a další funkcionalitu a také je to snadno instalovatelný webový nástroj a nic nestojí, může ho autor doporučit jako ideální pro organizování malých týmů, které chtějí mít software, který jim práci usnadňuje, ne ji zdržuje procesy, které tým nemá osvojené a necítí potřebu (třeba jen zatím) je následovat.

Velmi důležité je také, aby uživatelé měly možnost vyjádřit svoje požadavky pohodlně a jednoduše, protože jen tak se vývojářům dostane tolik potřebné zpětné vazby od lidí, kteří jsou pro produkt nejdůležitější, a to i uživatelů technicky tolik nevyspělých. Stejně jako budujeme náš systém tak, aby jeho ovládání bylo intuitivní a pro většinu akcí nevyžadovalo nahlednutí do manuálu, chtěli bychom, aby reakce uživatelů mohla být učiněna rychle a bezbolestně. Také bychom chtěli, aby se tyto požadavky objevily v naší databázi co nejrychleji bez zásahu lidské síly, jen tak budeme schopni spravovat jejich velké množství. Autorovo doporučení tedy směřuje k vytvoření možnosti zadávat požadavky na rozšíření či oznamovat chyby přímo z aplikace, která by pak kontaktovala náš software na správu těchto událostí a záznam by do ní sama vložila. Zvláště jednoduché by to bylo u webových aplikací, kdy by se pouze do kódu stránky vložil odkaz vedoucí na komponentu umožňující oznámení vytvořit a odeslat. Tato komponenta by pak sama odesílala více důležitých informací (především ke které části systému se připomínka váže), uživatel by jich tedy nemusel vyplňovat příliš mnoho. I newebové aplikace by mohly mít podobný subsystém zaintegrovaný; bylo by sice samozřejmě nutné být připojený k Internetu, to ale v dnešní době není nadměrný požadavek. Méně pohodlně, ale dostatečné je z aplikace odkázat na webovou stránku umožňující zadat zpětnou vazbu. Některé aplikace již podobný systém mají, často ale neumožňuje poslat připomínku na cokoliv a kdekoliv v aplikaci. Často se po selhání programu objeví průvodce odesláním zpětné vazby, v tu chvíli ale uživatel není na zpětnou vazbu příliš naladěn, navíc to reprezentuje jen malou část možných hlášení chyb a žádné navrhování rozšíření. Namátkou, prohlížeč Firefox umožňuje nahlásit špatně zobrazenou stránku, komunikační program Skype pak kvalitu provedeného hovoru, možnosti jsou tedy

omezené. Jsou ale i produkty, které již možnost zadání zpětné vazby obsahují, z neinternetových například iTunes (nicméně řešenou odkazem na webovou stránku a prezentovanou jako žádost o hudbu nedostupnou v katalogu), z webových pak například Netvibes nebo verze obchodního systému OxyShop pro společnost Czech Computer. Touha po větší zpětné vazbě je znát i z mnoha jiných softwarů, velký podíl na tom má velké rozšíření Internetu, kdy je snadné připomínky uživatelů získat. V době psaní této práce firma Microsoft zpracovává zpětnou vazbu od uživatelů betaverze nové verze operačního systému Windows – Vista, kterou měl de facto každý možnost si zdarma stáhnout a otestovat (a udělalo to v případě verze beta 2 dva milióny uživatelů) [63].

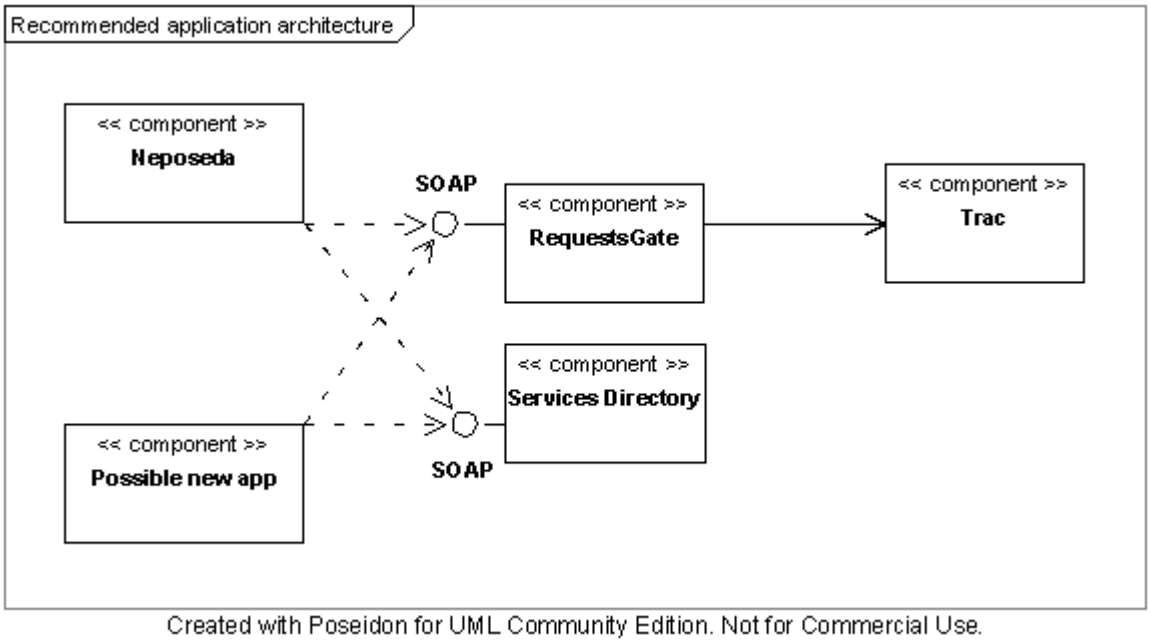
Aby naše databáze žádostí od uživatelů byla přístupná z více aplikací a zároveň jsme nebyli závislí na konkrétním produktu pro jejich správu, autor doporučuje vytvořit jí předřazenou bránu reprezentující její rozhraní. Brána by byla dostupná jako webová služba a přijaté požadavky by zadávala do databáze námi vybraného produktu. Při změně tohoto softwaru by se tak změnila jen tato brána (služba), aplikace by se měnit nemusely. Databáze požadavků by měla být přístupná stejné množině uživatelů, která má možnost zpětnou vazbu zadat, aby si každý mohl najít, jak se s tou kterou jeho žádostí naložilo. Zadal-li však uživatel při vyplňování své zpětné vazby i svůj e-mail, většina systémů mu bude schopná změnu stavu žádosti zaslat. Toto řešení není jen pro veřejně přístupné systémy, může být použito velmi dobře i v systémech s omezeným množstvím uživatelů, například v proprietárním informačním systému. V takovém případě je ještě důležitější, aby zpětnou vazbu bylo možné zadat jednoduše a pohodlně, protože uživatelů zadávajících požadavky by nebylo tolik jako v případě veřejně dostupné služby, a tak by se každý velmi ocenil.

Je nutné zdůraznit, že zohledňování návrhů na změnu a vylepšování od uživatelů neznamená, že vše, o co si někdo požádá, bude také implementováno. Vždy je třeba mít na mysli hlavní účel produktu, zpětnou vazbu od uživatelů pak zvažovat a analyzovat (případně s uživateli vyjednávat, vyjasňovat si stanoviska) a až poté vybrané funkce implementovat.

10 Vzorová implementace doporučení

Jako ukázka, že navrhovaný způsob sběru žádostí od uživatelů je technicky proveditelný a že jeho implementace není nijak náročná, byla architektura uvedena v praxi. Byla vytvořena ukázková webová aplikace obsahující možnost kdekoliv nahlásit chybu nebo požádat o rozšíření či změnu. Tato komponenta aplikace pak kontaktuje webovou službu (také implementovanou autorem práce), která funguje jako předřazená brána systému na správu chyb Trac. Tato webová služba pak vloží požadavek, který ji byl předán, do databáze Tracu. Aby byla zaručena objevitelnost stávajících i budoucích prvků architektury, byl implementován i jednoduchý adresář služeb (viz Obr. 4).

Myšlenka vzorové aplikace, pracovně nazvané Neposeda, je umožňovat komukoliv najít si krátkodobou práci vykonavatelnou přes Internet, spojit se s jejím zadavatelem a nad tímto úkolem komunikovat [64]. Toto je celý koncept, který byl v době zahájení budování aplikace vymyšlen, vše ostatní je ovlivněno námitkami uživatelů, které je napadnou při reálném používání aplikace (nutno podotknout, že autor aplikace je také její uživatel). Ačkoliv v budoucnu tato aplikace bude nejspíše mít rozhraní webové služby, v počátečních fázích, kdy není jasné, jaká data a operace bude přesně poskytovat, a navíc neexistuje žádný potenciální subjekt, který by toto rozhraní používal, by bylo zbytečné ho implementovat. Toto nikterak neodporuje servisně orientované vlastnosti znovupoužitelnosti, neboť ta požaduje budované rozhraní vytvořit jako znovupoužitelné, aniž existuje aktuální potřeba opětovného použití. Tzn. budeme-li v budoucnu budovat rozhraní pro jeden subjekt, je žádoucí, aby bylo implementované jako znovupoužitelné; budovat rozhraní za každou cenu nutné není.



Obr. 4: Implementovaný prototyp servisně orientované architektury sbírající uživatelskou zpětnou vazbu z aplikací

Jediný požadavek na architekturu byl, aby požadavky na změnu, návrhy rozšíření či oznámení o chybě zadané z této a jakékoliv budoucí aplikace rodiny bylo možné spravovat najednou z jedné standardní aplikace k tomu určené. Dalším požadavkem bylo, aby tato aplikace byla snadno nahraditelná jinou, aniž by se musely přepisovat komponenty zajišťující zpětnou vazbu ve všech aplikacích. Požadavky pak měly obsahovat e-mail zadavatele, předmět požadavku, jeho vlastní text, jeho typ – jestli se jedná o hlášení chyby nebo požadavek na změnu (rozšíření) –, identifikaci aplikace a přesného místa, ze které byl požadavek zadán, a konečně datum a čas zadání požadavku. Je zřejmé, že jméno aplikace a její část, ve které se uživatel nacházel, stejně jako datum a čas zadání nemusíme vyžadovat po uživateli, ale je možné ho vygenerovat aplikací automaticky. Prohlížení požadavků a jejich další úprava nebyla (možná jen zatím) požadována mimo vybranou aplikací na jejich správu.

Na základě těchto požadavků byla vytvořena brána – webová služba realizující přístup k (libovolnému) systému na správu chybových hlášení. Té bylo vytvořeno rozhraní nazvané requests_gate s jedinou metodou: *submit* s řetězcovými parametry *reporter*, *summary*, *description* a *component* a jedním logickým parametrem *bug*, který rozlišuje oznámení chyb od požadavků na rozšíření. Parametr component identifikuje aplikaci (v systému na správu změnových požadavků je často tato položka přítomna takovým způsobem, že podle ní lze vyhledávat či filtrovat), kontext v rámci aplikace z důvodu možného velmi odlišného formátu pro různé aplikace v rozhraní přítomný není; je předpokládáno že bude aplikacemi připojen k textu žádosti (tedy k parametru description). Tato služba tedy po obdržení požadavku kontaktuje aplikačně specifickým způsobem vybraný issue management software – v našem případě odesílá HTTP POST požadavek na založení nového „ticketu“ aplikace Trac [65]. Pokud by se Trac vyměnil za jakýkoliv jiný obdobný systém, změnil by se pouze způsob, jakým do něj webová služba Requests gate předává žádosti, do aplikací by nebylo třeba zasahovat nijak.

Pokud by nějaké nové požadavky či nároky případné nové aplikace vyžadovaly bohatší rozhraní, jednoduše by se rozhraní requests_gate přidala další metoda, staré aplikace by stále mohly používat tu původní. V samotné aplikaci Neposeda je na každé stránce možné vložit „bug“ či navrhnout „change or enhancement“, zadat e-mail, předmět a text požadavku.

Aplikace pak do parametru component doplní své jméno a k jeho textu pak kontext, ve kterém byla připomínka zadána (tedy URL).

Aby byla zachována servisně orientovaná vlastnost objevitelnosti (a tím aby nebylo nutné při změně umístění služby Requests gate měnit přístupové body k ní v každé aplikaci), byl implementován i jednoduchý adresář služeb. Jeho rozhraní services_directory obsahuje dvě základní metody, totiž možnost vytvořit nebo změnit záznam o službě a takový záznam získat podle jména služby. Uživatelům je pak k dispozici přehled všech záznamů v databázi adresáře [66]. Záznam o službě (její formální kontrakt) se skládá z jejího jména, jména počítače, portu, cesty v URL, na kterém se nachází formální popis služby, a účel služby vysvětlujícího popisu.

Služby architektury byly implementovány za pomocí standardů webových služeb, jejich rozhraní jsou tedy dostupná přes dokumenty typu WSDL ([67] a [68]). Ty jsou pak mohou být téměř vše, co aplikační programátor potřebuje, aby mohl volat nějakou metodu rozhraní (buď jsou mu objekty v jeho jazyce přímo z WSDL vygenerovány nebo jako v Ruby se z WSDL v reálném čase vyrobí třída schopná metody rozhraní volat).

11 Literatura

[1] Pressman Roger S: Software Engineering: A Practitioner's Approach. 6/e , 2005

[2] Bosch Jan: Design and Use of Software Architectures. Pearson Education Limited, London 2000

[3] Davis Alan M.: Software Requirements: Objects, Functions, States. Prentice Hall PTR, Upper Saddle River, NJ, USA 1993 00500IG

[4] Harwell et al.: What Is a Requirement? Processing 3rd Annual International Symposium National Council Systems Engineering, str. 17-24 1993

[5] Wieringa Roel J.: Requirements Engineering: Frameworks For Understanding. John Willey & Sons Ltd. Chichester, England 1996

[6] Mc Dermid, John A.: „Requirements analysis: Orthodoxy, fundamentalism and heresy“, z Jirotka, Goguen: „Requirements Engineering: Social and Technical Issues“, strany 17 – 40, Academic Press Ltd. 1994

[7] Object Management Group: „UML“ http://www.uml.org/, navštíveno v květnu 2006

[8] Ambler, Scott W.: „Agile Modelling: UML 2 Deployment Diagrams“, http://www.agilemodeling.com/artifacts/deploymentDiagram.htm, navštíveno v květnu 2006

[9] Wikipedia: Unified Modeling Language, http://en.wikipedia.org/wiki/Unified_Modeling_Language, navštíveno v květnu 2006

[10] Wikipedia: Waterfall model, http://en.wikipedia.org/wiki/Waterfall_model, navštíveno v květnu 2006

[11] Král Jaroslav: „Slajdy a další materiály k přednášce Informační systémy“, školní roky 2004 - 2006

[12] Wikipedia: Agile Software Development, http://en.wikipedia.org/wiki/Agile_software_development, navštíveno v květnu 2006

[13] Moore Jim: „ISO 12207 and Related Software Life-Cycle Standards“, http://www.acm.org/tsc/lifecycle.html, 22.června 1999

[14] Turpin Rufus A., Larry F. Jones: „Introduction to ISO/IEC 12207 – Information Technology – Software Life Cycle Processes“, IT Conference, http://www.statcan.ca/english/conferences/it2003/pdf/session18.pdf, 2.4.2003

[15] Manifesto for Agile Software Development, http://agilemanifesto.org/, navštíveno v květnu 2006

[16] Wikipedia: Extreme Programming, http://en.wikipedia.org/wiki/Extreme_programming, navštíveno v květnu 2006

[17] Extreme Programming: A gentle introduction., http://www.extremeprogramming.org/, navštíveno v květnu 2006

[18] Fried, Jason: „Getting Real, Step 1: No Functional Spec“, Signal vs. Noise, blog společnosti 37signals, http://www.37signals.com/svn/archives/001050.php, navštíveno v květnu 2006

[19] O'Reilly, Tom: What Is Web 2.0, Design Patterns and Business Models for the Next Generation of Software oreilly.net, http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, 30. září 2005

[20] The Atlantic Systems Guild Inc.: „Volere“, http://www.volere.co.uk/tools.htm, navštíveno v červnu 2006

[21] INCOSE: „Requirements Management Tools Survey“, http://www.paper-review.com/tools/rms/read.php, navštíveno v červnu 2006

[22] IBM: „Rational RequisitePro Tutorial“, verze produktu: 7, nainstalováno v červnu 2006

[23] Borland: „Borland CaliberRM Tutorial“, verze produktu: 2005, nainstalováno v červenci

2006
[24] Telelogic: „Using DOORS“, verze produktu: 8.0, 12. srpna 2005
[25] "SOA Power Panel" with Jeremy Geelan Live From Times Square, By: SYS-CON TV, 4. prosince 2005, http://www.sys-con.tv/read/159683.htm
[26] Datz Todd: „What You Need to Know About Service-Oriented Architecture“, CIO, January 15 2004, CXO Media http://www.cio.com/archive/011504/soa.html
[27] Erl Thomas: Service-Oriented Architecture: Concepts, Technology, and Design, Pearson Education, Inc. Upper Saddle River, NJ, USA 2005
[28] IBM: „Standards and Web services“, http://www-128.ibm.com/developerworks/webservices/standards/, navštíveno v červnu 2006
[29] Koch Christopher: „The Battle for Web Services“, http://www.cio.com/archive/100103/standards.html, 1.října 2003
[30] La Monica Martin: „Where's the simplicity in Web services?“, News.com, http://news.com.com/Wheres+the+simplicity+in+Web+services/2100-7345_3-5395630.html, 5. října 2004
[31] Winer David: „XML-RPC Specification“, http://www.xfront.com/REST-Web-Services.html, 15. června 1999
[32] Fielding, Roy Thomas: „Architectural Styles and the Design of Network-based Software Architectures“, doktorská dizertace, University of California, Irvine, 2000
[33] Costello Roger L.: „Building Web Services the REST Way“, xFront, navštíveno v květnu 2006
[34] Hinchcliffe Don: „REST vs. SOAP: The Battle of the Web Service Titans“, SOA Web Services Journal, http://webservices.sys-con.com/read/79282.htm, 26. dubna 2006
[35] Král Jaroslav, Žemlička Michal: „Software Confederations—An Architecture for Global Systems and Global Management“, Idea Group Inc. 2003
[36] Král Jaroslav, Žemlička Michal: „Software Confederations and Alliances“, 2003
[37] Král Jaroslav, Žemlička Michal: „Implementation of Business Processes in Service-Oriented Systems“, 2005
[38] W3C Recommendation 24 June 2003: SOAP Version 1.2 Part 1: Messaging Framework, http://www.w3.org/TR/soap12-part1/, navštíveno v květnu 2005
[39] W3C Note 15 March 2001: Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, navštíveno v květnu 2005
[40] UDDI Version 3.0.2: UDDI Spec Technical Committee Draft, Dated 20041019, http://uddi.org/pubs/uddi_v3.htm, navštíveno v květnu 2005
[41] Manes, Anne Thomas: Document vs. RPC style, why is it a big deal?, SearchWebServices.com,
http://searchwebservices.techtarget.com/ateQuestionNResponse/0,289625,sid26_cid494324_tax289 201,00.html, 30 září 2002
[42] Koch, Christopher: „The ABCs of ERP,“ Enterprise Resource Planning Research Center, CIO Magazine, http://www.cio.com/research/erp/edit/erpbasics.html, 10. ledna 2006
[43] ERP/SOA Resource Center: „Service Oriented Architecture (SOA)“, http://www.army.mil/escc/erp/soa.htm, 30.4. 2006
[44] Oracle: „Service-Oriented Architecture: Build, Deploy, and Manage SOA with Best-of-Breed Oracle Technology“, http://www.oracle.com/technologies/soa/index.html, navštíveno v červnu 2006
[45] IBM: „UML 2.0 Profile for Software Services“, http://www-128.ibm.com/developerworks/rational/library/05/419_soa/, 13. dubna 2005
[46] Noterik B.V.: „StreamEdit MAM“, http://noterik.com/downloads/folders_2005/StreameditMAM_UK.pdf, navštíveno

v červenu 2006
[47] Eindhoven Dagblad: EDTV, http://www.eindhovensdagblad.nl/edtv/, navštíveno v červenu 2006
[48] Noterik: W-IPTV backend, http://ed.noterik.com/adserver/, navštíveno v červenu 2006
[49] Ruby on Rails, http://rubyonrails.com, navštíveno v červnu 2006
[50] Noterik: Stats http://stats.noterik.com/export, navštíveno v červnu 2006
[51] Kulhánek Jiří: „Integrace heterogenních informačních systémů“, diplomová práce MFF UK, 2006
[52] Päivi Parviainen: „Embedded software development; technologies and experiences of their use“, VTT Electronics, http://www.tekes.fi/ohjelmat/sweng/sem05/Parviainen%2024.5.2005.pdf, 24. května 2005
[53] Best Practical: „RT: Request Tracker“, http://bestpractical.com/rt/, navštíveno v červenci 2006
[54] Mantis bug tracking system, http://www.mantisbt.org/, navštíveno v červenci 2006
[55] The Standish Group: „The CHAOS Report (1994)“, http://www.standishgroup.com/sample_research/chaos_1994_1.php, navštíveno v červenci 2006
[56] The Standish Group: „CHAOS: A Recipe for Success (1999)“, http://www.standishgroup.com/sample_research/PDFpages/chaos1999.pdf, navštíveno v červenci 2006
[57] Putnam Doug: „Team Size Can Be the Key to a Successful Project“, http://www.qsm.com/process_01.html, navštíveno v červenci 2006
[58] Fried Jason: „Collaborative Technologies Conference speech“, http://www.collaborationloop.com/blogs/ctc-2006-jason-fried-2.htm, 22. června 2006
[59] Gawlas Julius: „Mission-Critical Development with XP & Agile Processes“, Dr. Dobb's Portal, http://www.ddj.com/184405520?pgno=1, 1.prosince 2003
[60] Object Management Group: „Model Driven Architecture“, http://www.omg.org/mda, navštíveno v červenci 2006
[61] IT Conversations: „SOA Reality Check Part II“, http://www.itconversations.com/shows/detail973.html, 11.8.2005
[62] Edgewall Software: „Trac: Integrated SCM & Project Management“, http://trac.edgewall.org/, navštíveno v červenci 2006
[63] Microsoft: „Financial Analyst Meeting 2006, Kevin Johnson's speech“, http://www.microsoft.com/msft/speech/FY06/Johnson_MugliaFAM2006.mspx, 27. srpna 2006
[64] Kubr Jan: „Neposeda“, Simple task management application, http://neposeda.vserver.cz/
[65] Edgewall Software: „Neposeda Trac“, Instance aplikace Trac pro server Neposeda, http://neposeda.vserver.cz/trac
[66] Kubr Jan: „Neposeda services list“, Seznam služeb vzorové architektury, http://neposeda.vserver.cz:3001/
[67] Kubr Jan: „Neposeda Request gate WSDL“, Formální kontrakt služby Requests gate, http://neposeda.vserver.cz:3000/request/wsdl
[68] Kubr Jan: „Neposeda Services directory WSDL, Formální kontrakt adresáře služeb, http://neposeda.vserver.cz:3001/service/wsdl

Příloha 1: Odpovědi na anketní otázky

Položené otázky:

1. Jaké softwary na správu požadavků jste používali a jaké na ně máte názory?
2. Jaké byly nejdůležitější a nejčastější funkce, které jste používali?
3. Co je pro vás u takového softwaru nejdůležitější?
4. Pokud jste žádné dosud nepoužívali, jak požadavky spravujete (popř. jak jste je spravovali před používáním softwaru)? Uvažujete, že nějaký takový produkt používat začněte?
5. Co pro vas specifikační software představuje? Zásadní urychlení správy požadavků, nutné zlo při velkém množství požadavků, nebo jen o něco pohodlnější praci?
6. Jaky je váš názor na servisně orientovanou architekturu? Máte pocit, ze řeší některé palčivé problémy ICT dneška nebo je to jen prázdný pojem?
7. Pokud jste někdy specifikovali nebo přímo navrhovali SOA nebo její prvky, jaký software jste použili? Co byste nejraději použili příště (jaké featury)?
8. Vidíte zásadní rozdíly v procesech specifikace SOA a tradičních systémů?

Získané odpovědi:

Jan Pavelka (DCIT):

Jan Pavelka byl kromě společných otázek dotázán i specificky na projekt informačního systému Univerzity Karlovy. Jeho odpovědi:

Výsledné zadání lze zhruba shrnout takto:

1. zajistit výměnu dat mezi základními agendami (personalistika, mzdy, studium, koleje) tak, aby se aktualizace dat (polo)automaticky přenášely a nedocházelo k redundancím a nekonzistencím,
2. inovovat některé aplikace, zejména zavést centrální personální agendu,
3. doplnit některé chybějící aplikace.

Studie byla svého času vystavena na webu UK. V roce 2004 jsem se k projektu vrátil v roli manažera kvality. Odpovědi na Vaše otázky najdete níže v textu. Jejich podrobnost je limitována mou smluvní povinností chránit důvěrné informace subjektů zúčastněných na projektu.

1. Jak byly sbírány požadavky pro tento systém, byl použit nějaký specializovaný software?

Primárním zdrojem informací byly normativní dokumenty UK a interview s útvary IT a zástupci uživatelů na většině fakult a součástí UK. Pro zaznamenávání získaných informací jsme použili nástroj CASEwise Corporate Modeller, na kterém se nám osvědčila zejména možnost
publikování modelů ve formátu HTML.

2. Co myslíte, že je hlavní příčina neúspěchu původně plánovaného řešení?

Náročnost technické koncepce řešení.

3. Jaký je vztah navrhovaného řešení a plně servisně orientované architektury, pomohlo by jeji použití?

Použití SOA by vyžadovalo implementovat příslušná rozhraní např. v mzdových systémech pěti různých výrobců. Bez hlubší analýzy nemohu posoudit, zda by SOA byla životaschopnou alternativou ke zvolené koncepci.

Následují odpovědi na společné otázky:

1. V průběhu projektu byly požadavky evidovány v MS Excel (přehledově) a MS Word (specifikace jednotlivých komponent a aplikací s odkazy do přehledové evidence).
2. N/A (viz 1)
3. Možnost pracovat se vztahy mezi požadavky. V roli externího konzultanta jsem viděl jsem použití specializovaného softwaru pro sběr a analýzu požadavků na jiných projektech a ve všech těchto případech skončil drahý nástroj jako glorifikovaný Excel. V případě opakovaně nasazovaných produktů (tady se už dostávám mimo rámec projektu IS UK) je nezbytné ošetřit variantní požadavky různých instalací s úzkým vztahem na konfigurační řízení.
4. Nejsem řešitelem projektu.
5. Od specifikačního software bych neočekával žádné urychlení, natož zásadní. Aby byla práce s ním pohodlná, musí ho uživatelé dobře ovládat, což vůbec není samozřejmost - viz odpověď na otázku 3. Podle mne je hlavní problém specifikace v komunikaci se zadavatelem a v tom žádný nástroj nepomůže. Mimochodem, v roce 2001 jsem se na konferenci Requirements Engineering setkal s prof. Parnasem, který byl stejného názoru.
6. Nemám zkušenost a proto nemohu mít fundovaný názor. Myslím, že hodně záleží na tom, jak se SOA ujme, tj. zda se stane de facto standardem pro interfejsování aplikací (třeba Gartneři si myslí, že to bude do roku 2008). Jestli ano, zjednoduší se výrobcům software život, protože integrace aplikací palčivým problémem je.
7. Nespecifikoval, nenavrhoval.
8. ... a proto nemohu vidět.

David Bílek (Unicorn):

1. ClearQuest, JIRA, TrackRecord včechny se dají použít, není problém, pro správu požadavků stačí mít i excel, záleží na rozsahu projektu. Klíčem je nastavit spravně životní cyklus a donutit okolí požadavky evidovat a řešit. Klíčem je vůbec definovat co je požadavek.
2. Založ, uprav, schval, neschval, přiřaď odpovědnou osobu, reporting a různé sestavya statistiky.
3. Snadná použitelnost, stabilita a otevřený přístup datům pro zajištění reportingu.
4. Když se nastroje nepoužívají, většinou se evidují v dokumentech (Word, OpenOffice).
5. Pořádek, který je klíčem k úspěšnému pochopení požadavku a pak řízení jeho životního cyklu až k úspěšné realizaci a nasazení.
6. SOA je teoretický princip založený na správných principech, který je však velmi obtížné implementovat do praxe, zejména ve větších organizacích kde je business logika "rozdělena" do velkého množství různých aplikací na různých technologií,

funkčnost je v těchto případech duplicitní a implementována různými způsoby.
Prvním krokem zavedení SOA je standardizace některých operací, což samo o sobě může být problémem, zejména komunikačním. Zavedení SOA také vyžaduje lepší řízení závislostí mezi jednotlivými požadavky a systémy.
Architektura vysoce dostupných služeb je velmi náročná a je tomu nutné uzpůsobit hardwarovou infrastrukturu.
SOA jako princip muze resit zejména problémy spojené se znuvupoužitelností a lepší škálovatelností funkčního portfolia organizace na druhou stranu zvyšuje náročnost na řízení a správu požadavků, provoz a servis služeb.

7. Nedokážu odpovědět.
8. Sice nevím, co je tradiční systém, ale většina otázek je zodpovězena v otázce 6. Obecně platí, že jasná specifikace požadavků (stabilita) a velký důraz na architektu aplikace, aplikačního software (aplikační servery, DB software) a hardwarové infrastruktury je kritickým prvkem (ne)úspěchu v případě návrhu a implementace.

Andrew Kirkness (kontraktor):

1. None. I have to admit I don't like the idea of requirements management software. Unless you mean managing the documentation like word documents and excel spreadsheets for traceability.
2. N/A.
3. I can't imagine buying any - someone would have to work pretty hard to convince me some software would help.
4. I document requirements for each stage and often use spreadsheets to trace requirements between stages, e.g. start with high level scope/reqs then track how these expand to full business reqs then track these to relevant functional specifications.
5. I can't imagine using software.
6. It means unnecessary overhead to me (unless you mean word docs and spredsheets).
7. Not deeply involved in SOA, like any buzzword, it can help if applied well but with many people it's an empty buzzword.
8. N/A.
9. Yes, but I'm not an expert.

Tomáš Rubač (Arbes Technologies):

1. Vlastní evidenční systém
2. Správa požadavků - plánování do verzí - evidence chyb - evidence verzí - evidence instalací
3. Jednoduchost ovládání - sledování historie - možnost jakýchkoli oprav - automatizace procesů (hromadně)
4. XX
5. Zásadní ZKVALITNĚNÍ výsledného systému - nejde o rychlost, ani o pohodlnost, ale není v tom pak nepořádek a tím pádem se nezapomíná něco slíbeného udělat nebo něco co souvisí s něčím jiným atp. Včas se zjišťuje, že se něco nestíhá atp.
6. XX
7. NE
8. XX