Česká společnost uživatelů otevřených systémů EurOpen.CZ
Czech Open System Users' Group
www.europen.cz

XXVI. konference – XXVIth conference

Sborník příspěvků

Conference proceedings

Hotel Relax
Monínec
15.–18. května 2005

Sborník příspěvků z XXVI. konference EurOpen.CZ, 15.–18. května 2005
© EurOpen.CZ, Univerzitní 8, 306 14 Plzeň
Plzeň 2005. První vydání.
Editor: Vladimír Rudolf, Jiří Felbáb
Sazba a grafická úprava: Ing. Miloš Brejcha – Vydavatelský servis
Vytiskl: IMPROMAT CZ, spol. s r. o., U Hellady 697/4, Praha 4, provozovna
Copyshop Smetanovy sady 6, 301 37 Plzeň

ISBN 80-86583-08-2

Upozornění:
Všechna práva vyhrazena. Rozmnožování a šíření této publikace jakýmkoliv způsobem bez výslovného písemného svolení vydavatele je trestné.

Obsah

Alena Buchalcevová
Jaký má být dnes vývoj softwaru 5

Petr Hřebejk
Using open source in commercial companies 17

Jaroslav Tulach
Test Patterns In Java 29

David Štrupl
Using NetBeans as a Framework for a Network Monitoring Application 63

Thomas Seidmann
Architecture of a Business Framework 71

Jan Rychlík
Rozdílné přístupy komerčních organizací a rozpočtových organizací k zavedení informačního systému 79

Vladimír Rudolf
Trnitá cesta od papírové univerzity k e-univerzitě 89

Václav Pergl
Agilní krabice? 99

Maxmilián Otta
Integrace podnikových aplikací pomocí open-source nástrojů 103

Jan Matějka
Softwarové patenty – minulost, přítomnost, budoucnost 113

Filip Molčan
OpenOffice.org 119

Jan Valdman
Do roka a do dne, aneb roční zkušenosti s implementací univerzitního portálu 125

Daniel Holešínský, Robert Bohoněk, Jiří Šimonek
Testování a ladění výkonnosti J2EE aplikací pro Websphere Portal 131

Oldřich Nouza
Generování zdrojového kódu – pomůcka, nebo překážka? 141

Programový výbor

Felbáb Jiří, Platina Praha
Rudolf Vladimír, ZČU Plzeň
Tulach Jaroslav, Sun Microsystems Praha

Jaký má být dnes vývoj softwaru – business driven, test driven, model driven, architecture driven nebo service oriented?

Alena Buchalcevová
E-MAIL: BUCHALC@VSE.CZ

Abstrakt

Obrovsky rychlý vývoj hardwaru a síťových technologií jde ruku v ruce s vývojem základního softwaru, technologií middlewaru, programovacích jazyků a vývojových prostředí. Významné změny nastávají i v přístupech k procesům vývoje softwaru. Příspěvek se zaměřuje na nejvýznamnější trendy, přístupy a metodiky, které v dnešní době hýbou vývojem softwaru, a které se snaží najít cestu k větší úspěšnosti softwarových projektů a identifikovat přínosy zavedení IS/ICT v podnikání organizace.

1 Nejvýznamnější trendy v oblasti IS/ICT

Články, průzkumy a předpovědi společnosti Gartner patří mezi nejvýznamnější zdroje, které monitorují směry vývoje v různých oblastech, oblast IS/ICT nevyjímaje. Proto jsem při přípravě tohoto příspěvku vycházela z předpovědi společnosti Gartner pro rok 2005 „Predicts 2005: Deploy New Technology, Applications for Success". Velmi potěšující bylo zjištění, že se tyto předpovědi nejdůležitějších trendů pro rok 2005 shodují s tématy jarní konference EurOpen. Společnost Gartner uvádí tyto nejdůležitější technologické trendy [Gartner, 2004]:

- Linux a Open Source software bude stále více prorůstat do podnikové infrastruktury,
- osobní počítače se budou stále více virtualizovat, tj. počítačové zdroje se budou jak hardwarově, tak softwarově rozdělovat na části (partitioning), budou sdíleny a simulovány,
- další masový rozvoj mobilních a bezdrátových technologií,
- bezpečnost se zaměří zejména na prevenci průniků a útoků,

- ještě důraznější bude prosazování hesla „anytime anywhere“,
- boj mezi platformami J2EE a Microsoft .NET skončí provozováním obou platforem a důrazem na jejich integraci.

V oblasti aplikací Gartner předpovídá tyto přístupy k vývoji softwaru:

- pokračující snaha o snižování nákladů povede k růstu offshore outsourcingu a outsourcingu byznys procesů,
- webové služby, které se v mnoha odvětvích a organizacích dostávají za hranice pilotních projektů, budou v roce 2005 hlavním trendem,
- společnosti budou investovat do nástrojů na podporu sdílení znalostí.

1.1 Sladění IS/ICT s podnikovými procesy

Klíčovým trendem, který jde za rámec IS/ICT, je požadavek na sladění IS/ICT s podnikáním. Tento požadavek se objevuje již několik let mezi prioritami v oblasti IS/ICT. Přehled pěti nejdůležitějších priorit v oblasti IS/ICT podle 2003 Worldwide IT Benchmark Report společnosti Meta Group [Metagroup, 2003] pro respondenty z USA a z ostatních zemí světa uvádí tabulka 1.

Tabulka 1 Přehled pěti nejdůležitějších priorit v oblasti IS/ICT

5 nejvyšších priorit pro USA	**5 nejvyšších priorit pro ostatní svět**
1. snížení nákladů	1. zvýšení produktivity
2. business alignment	2. snížení nákladů
3. zvýšení produktivity	**3. business alignment**
4. řízení projektu	4. zlepšení SW procesů
5. zlepšení kvality SW	5. zlepšení kvality SW

Business alignment je odrazem současné ekonomické recese, která významně ovlivňuje i oblast IS/ICT. Výdaje na informatiku se snižují, požaduje se vyšší kvalita, kratší čas a přínosy IS/ICT v podnikání. Právě sladění IS/ICT s podnikovými procesy má být řešením tohoto problému. Jde o to, aby čas a prostředky vynaložené na projekty v oblasti IS/ICT měly přímý vliv na podnikání. IS/ICT je třeba spojit s podnikovými procesy a hodnotit úspěch zavedení IS/ICT podle toho, jak se projeví v podnikání organizace. Nástrojem, jak toho dosáhnout, se zdá být koncept služeb. Služby vystupují jako spojovací článek mezi podnikovými procesy a IS/ICT. Společnost Meta Group je přesvědčena, že jedním z nejdůležitějších trendů v informatice v následující dekádě bude právě orientace na služby [Metagroup, 2003]. Otázka orientace na služby při vývoji IS/ICT není jen otázkou technologie, ale podstatně ovlivňuje i metodiky vývoje a nasazení IS/ICT. Ačkoli se technologie i ekonomické prostředí za poslední desetiletí

dramaticky změnily, metodiky většinou s těmito změnami nepočítají. Většina metodik byla navržena pro prostředí, kde se aplikace vyvíjejí od začátku, ne pro prostředí založené na službách, kdy je třeba řešení poskládat z různých zdrojů.

Architektura orientovaná na služby je postavena na těchto klíčových principech [Bloomberg, 2003]:

- podnikové procesy jsou určující pro služby, které jsou zase určující pro technologii
- služby musí umožňovat agilitu podnikových procesů
- úspěšná Architektura orientovaná na služby se stále vyvíjí.

Atraktivnost služeb spočívá ve zvýšení produktivity IS/ICT řešení, snížení nákladů vývoje a nasazení a zkrácení času uvedení na trh. Dalším cílem je vyšší přínos z IS/ICT řešení. Architektura orientovaná na služby je důležitá pro podniky, protože představuje rámec, který sjednocuje byznys model s technologiemi a realizuje funkcionalitu zajišťující efektivní podnikání.

2 Současný stav v oblasti vývoje IS/ICT

Úspěšnost projektů vývoje informačních systémů není uspokojivá. Podle výsledků výzkumu společnosti Standish Group splňovalo kritéria úspěšnosti v roce 2000 jen 28 % všech projektů na vývoj aplikací. Tato kritéria byla definována jako dokončení projektu včas, se všemi specifikovanými funkcemi a za daných nákladů [Johnson, 2001]. Příčin této skutečnosti je celá řada. Mezi nejvýznamnější bych zařadila:

- turbulentní změny ve společnosti,
- velmi rychlý vývoj informačních a komunikačních technologií,
- složitost vývoje softwaru,
- tlak na rychlost vývoje,
- železný trojúhelník,
- potřeba integrace se stávajícími systémy.

2.1 Turbulentní změny ve společnosti

Současná ekonomika je stále více orientovaná na znalosti, je charakterizována kratším životním cyklem produktů, důrazem na inovace a rychlým technologickým pokrokem. Svět kolem nás se mění a je třeba na změny nejen reagovat, ale dokonce změny vyvolávat, a tak získat náskok před konkurencí. Z pohledu IS/ICT to znamená, že jak při vývoji IS, tak při jeho provozu je třeba do systému promítat změny.

2.2 Velmi rychlý vývoj informačních a komunikačních technologií

Změny, které probíhají v celé společnosti, jsou ještě výraznější v oblasti informačních a komunikačních technologií, neboť právě tato oblast patří dnes k nejdynamičtějším. Obrovsky rychlý vývoj hardwaru a síťových technologií jde ruku v ruce s vývojem základního softwaru, technologií middlewaru, programovacích jazyků a vývojových prostředí. Tento rychlý vývoj nám na jedné straně poskytuje stále lepší prostředky, na druhé straně ale představuje závažný problém, pokud se zabýváme otázkami jako je uchování investic vložených do IS/ICT, kvalifikace vývojářů softwaru a podobnými. Právě na řešení těchto otázek je zaměřena iniciativa organizace OMG Modelem řízená architektura (Model Driven Architecture) MDA. MDA vychází ze skutečnosti, že množství změn v systému klesá s postupem na vyšší úrovně abstrakce. Dopady neustálých změn technologií je možné omezit jen na část modelu – na jeho nižší vrstvy. Při MDA vývoji se nejprve vytvoří Platformově nezávislý model (Platform Independent Model – PIM), který reprezentuje věcnou funkcionalitu a chování systému. Pomocí MDA nástrojů se PIM mapuje na zvolenou platformu (například Corba, Java/EJB, XML/SOAP) a generuje se Platformově specifický model (Platform Specific Model – PSM). Nakonec se generuje implementační kód pro příslušnou technologii. MDA nástroje umožňují také zpětné inženýrství (reverse engineering), a tak je možné vytvořit modely stávajících systémů pro účely integrace aplikací. MDA generátory aplikují současně i návrhové vzory [Buchalcevová, 2003].

2.3 Složitost vývoje softwaru

Na celou historii vývoje softwaru, která není ve srovnání s ostatními odvětvími dlouhá, můžeme pohlížet jako na boj se složitostí. Na jedné straně se do tohoto boje nasazují stále výkonnější nástroje, na druhé straně rostou požadavky na software (rozsah, kvalita, rychlost vývoje, flexibilita, přívětivost a další). Hlavním atributem softwaru je tedy stále složitost jeho vývoje, která je také jednou z příčin velkého počtu neúspěšných softwarových projektů. Na vývoj softwaru má vliv jak prostředí vývoje, tak cílové prostředí. Proměnnými veličinami při vývoji softwaru jsou dle [Scrum, 1995]:

- dostupnost kvalifikovaných specialistů (pro nové technologie, nástroje, metody a domény je malý počet kvalifikovaných odborníků),
- stabilita technologie pro implementaci (nové technologie jsou méně stabilní),
- stabilita a schopnosti nástrojů,
- efektivnost používaných metod,

- dostupnost expertů na věcnou oblast i technologii,
- nová funkcionalita a její vztah k existující funkcionalitě,
- metodika a její flexibilita,
- konkurence,
- čas,
- zdroje,
- další proměnné.

Celková složitost vývoje softwaru je funkcí těchto proměnných, přičemž tyto proměnné se v průběhu projektu mění.

Software má mnoho aspektů, které jej odlišují od jiných produktů, a proto je i proces jeho vývoje odlišný. Tradiční přístupy předpokládají, že procesy při vývoji softwaru je možné plně definovat a konzistentně opakovat. To předpokládá, že je možné definovat a opakovat: problém, řešení, nositele řešení (vývojáře) a prostředí. Tyto předpoklady však podle zastánců agilních přístupů při vývoji softwaru neplatí. V mnoha případech není možné definovat problém na začátku projektu, protože požadavky nejsou přesně specifikovány a nebo se mění. Opakovatelnost řešení předpokládá, že je možné plně specifikovat architekturu a návrh. Také vývojáři nejsou stejní, ale liší se svými schopnostmi a znalostmi. Liší se i prostředí, ve kterém vývoj probíhá. Vývoj softwaru tak probíhá v podmínkách chaosu a je to velmi složitý proces, který nelze předem plně popsat, ale je nutné jej průběžně monitorovat a přizpůsobovat se změnám.

2.4 Tlak na rychlost vývoje

Turbulentní ekonomické prostředí a také vysoká konkurence v odvětvích vedou k tomu, že je třeba realizovat změny rychle. A jestliže je software klíčovým faktorem fungování organizací, pak je třeba také software vytvořit a zavést velmi rychle, dříve než to udělá konkurence. To vystihuje termín „time to market“, který je klíčovým požadavkem při dnešním vývoji a nasazování softwaru.

2.5 Železný trojúhelník

Softwarový projekt je omezen železným trojúhelníkem (obrázek 1). Pokud je přesně definován plán (čas), rozpočet (náklady) a rozsah vytvářeného softwarového produktu (požadavky), nemá řešitelský tým žádný manévrovací prostor, a tak spěje k selhání. Jediné, co se pak může měnit, je kvalita produktu.

První bod železného trojúhelníku odpovídá na otázku „Jak dlouho bude vývoj trvat?“ Nerealisticky těsný plán je známým problémem v řadě projektů.

Obr. 1 Železný trojúhelník

Stejně problematický je ale i několikaletý plán bez interních dodávek, který negarantuje prakticky žádný pokrok. Řešením je realizovat projekt v krátkých iteracích. Primárním produktem musí být fungující software, který splňuje v daném čase nejdůležitější požadavky. Druhý bod trojúhelníku odpovídá na otázku „Co bude vývoj stát?“ Zdroje jsou v projektech většinou špatně řízeny, velmi brzy jsou zařazeny do určitých kategorií a jejich výše bývá na začátku často stanovována uměle. Třetí bod trojúhelníku odpovídá na otázku „Co dostaneme?“ Při tradičním způsobu vývoje se tým snaží definovat většinu požadavků brzy. Tento přístup ale nezohledňuje fakt, že požadavky se mění s tím, jak se vyvíjí znalosti a prostředí. Ve středu železného trojúhelníku je kvalita. Ta řeší otázku „Bude výsledek dost dobrý (kvalitní)?“ Kontroly, které mají odhalovat defekty, jsou tradičním způsobem garance kvality. Ale je to nejlepší přístup? Místo vyhledávání chyb po dodání, by bylo lepší vytvořit bezchybný produkt. Standardy kódování a pravidla modelování pomáhají v dosahování kvality stejně jako vývoj s testováním na začátku a aktivní účast investorů.

Jak se tedy vypořádat se železným trojúhelníkem? Řešením je asi dosáhnout rozumného kompromisu. Nejlepší manažeři si uvědomují, že pro zajištění úspěchu projektu je třeba vzít v úvahu železný trojúhelník a realizovat změny, pokud nastanou. Watts Humprey, jeden z tvůrců CMM řekl: „To, co lidé skutečně chtějí, je vysoce kvalitní systém, který implementuje vše, co chtějí, při nulových nákladech a hned. Vše ostatní je jen dohoda, kompromis.“

2.6 Potřeba integrace se stávajícími systémy

Informační systémy dnes nevznikají na zelené louce. Většina organizací má již automatizovány hlavní oblasti své činnosti. Požadavkem dnešní doby je integrace těchto „ostrůvků“ automatizace do jediného systému. Zatímco dříve byly informační systémy nahrazovány novými, dnes se začíná prosazovat názor, že je třeba stávající systémy, které dobře pracují, propojit s ostatními a zpřístupnit je například přes webové rozhraní. To je jeden z úkolů systémové integrace. Systémová integrace představuje způsob, jak umožnit technologiím ve stále se měnícím informatickém prostředí spolu komunikovat. Jak řekl Richard Soley, předseda a CEO OMG[1], ve své úvodní přednášce na OMG konferenci Integrate

[1] standardizační organizace Object Management Group

2003: „Integrace je problém, který nemůžeme odsunout.“ [Soley, 2000] Integrace není jen technologický problém, ale je to také obchodní problém. S požadavkem integrace systémů úzce souvisí nutnost celopodnikového (Enterprise) pohledu.

3 Metodiky

Složitost tvorby informačních systémů, jejíž příčiny jsem nastínila v předcházející kapitole, se již dlouhou dobu snaží řešit metodiky vývoje informačních systémů. Význam metodik pro vývoj informačního systému dokumentují například výsledky prezentované ve zprávě „2003 Worldwide IT Benchmark Report“ společnosti META Group [Metagroup, 2003], v níž se uvádí, že 51,6 % všech respondentů používá při vývoji informačních systémů metodiku.

3.1 Pojem metodika

Metodika (methodology) představuje v obecném smyslu souhrn metod a postupů pro realizaci určitého úkolu. **Metodika vývoje a údržby IS/ICT definuje principy, procesy, praktiky, role, techniky, nástroje a produkty používané při vývoji, údržbě a provozu informačního systému, a to jak z hlediska softwarově inženýrského, tak z hlediska řízení**. [Buchalcevová, 2005]

Kromě pojmu metodika se můžeme setkat s pojmy proces a softwarový proces. Mnohé metodiky mají slovo „proces“ přímo ve svém názvu. Příkladem jsou metodiky Rational Unified Process, Open Process, Object-Oriented Software Process a další. Existují metodiky hodnocení softwarových procesů (například Capability Maturity Model), mluví se o zlepšování softwarových procesů. Softwarový proces je v kontextu těchto přístupů definován jako sada činností, metod, praktik a transformací, které lidé používají pro vývoj a údržbu softwaru a dalších s tím spojených produktů (projektových plánů, návrhů, testovacích případů apod.). [Paulk, 1993]

3.2 Stav v oblasti metodik v ČR a ve světě

Metodik, které se zabývají vývojem informačních systémů, je velké množství. Tato skutečnost má objektivní důvody, kterými jsou mimo jiné:

- různé technologie a paradigmata vyžadují různé techniky (strukturované, objektové),
- organizace se liší firemní kulturou,
- každý jedinec je jedinečný a má jiný styl uvažování, práce,
- každý tým je jedinečný,
- projekty se liší velikostí týmu,

- projekty se liší důležitostí.

Důležité je, aby metodiky pro vývoj a údržbu IS/ICT byly jednotně popsány a kategorizovány. Charakteristiku předních současných metodik, jejich popis v jednotné struktuře a kategorizaci je možné nalézt v [Buchalcevová, 2005]. Zařazení metodik do dobře definovaných kategorií je velmi důležité, neboť metodiky nejsou srovnatelné v řadě hledisek. Většinou se jedná o metodiky zaměřené jen na určitou fázi vývoje informačního systému, na určitou věcnou oblast, na určitý typ projektu a podobně. Uvedená publikace se snaží také definovat kritéria, jak vybrat vhodnou metodiku pro určitý typ projektu, a postupy pro její přizpůsobení na konkrétní podmínky firmy a projektu.

Problém většiny metodik je také v tom, že se zaměřují na vývoj nového informačního systému. Přitom v dnešní době je hlavním úkolem zejména rozvoj stávajících systémů, implementace typových programových řešení, integrace dílčích řešení do celopodnikového systému.

Podíl firem, které používají při vývoji software formální metodiku, je v České republice nižší než ve světě. Tato skutečnost může mít řadu důvodů, ale jsem přesvědčena, že mezi hlavní důvody patří:

- nedostatek českých metodik[2], neboť většina metodik je v angličtině a nejsou lokalizovány do češtiny,
- metodiky se zpravidla šíří na komerční bázi a české firmy nechtějí či nemohou vydávat prostředky na nákup metodik.

3.3 Rigorózní metodiky a agilní metodiky

V současnosti můžeme sledovat dva hlavní proudy v metodických přístupech, které jsou označovány jako rigorózní metodiky a agilní metodiky. Rigorózní metodiky vycházejí z přesvědčení, že budování IS/ICT lze popsat, plánovat, řídit a měřit. Snaží se podrobně a přesně definovat procesy, činnosti a vytvářené produkty, a proto bývají často velmi objemné. Rigorózní metodiky jsou zpravidla založeny na sériovém (vodopádovém) vývoji. Při tomto způsobu vývoje probíhají jednotlivé fáze jako plánování, analýza, návrh, implementace, zavedení sekvenčně za sebou. Existují ale také rigorózní metodiky založené na iterativním a inkrementálním vývoji[3] Příkladem těchto metodik jsou OPEN, Rational

[2] z českých metodik můžeme uvést například metodiky *Objektově orientované metodiky a technologie* (OOMT) viz [Drbal, 1997], *Multidimensional Management and Development of Information System* (MMDIS) [Voříšek, 1997], *Business Object Relation Modeling* (BORM) viz [Polák, Merunka, Carda, 2003]

[3] Iterativní vývoj představuje opakované (iterativní) provádění jednotlivých fází při vývoji IS. Výsledkem každé iterace je funkční verze systému. Současné metodiky doporučují velmi krátké iterace (dny). Iterativní vývoj může probíhat buď pro celý systém, jehož funkčnost se v jednotlivých iteracích rozšiřuje, a nebo ve spojení s inkrementálním vývojem (systém se vyvíjí po přírůstcích). [KIT, 2003]

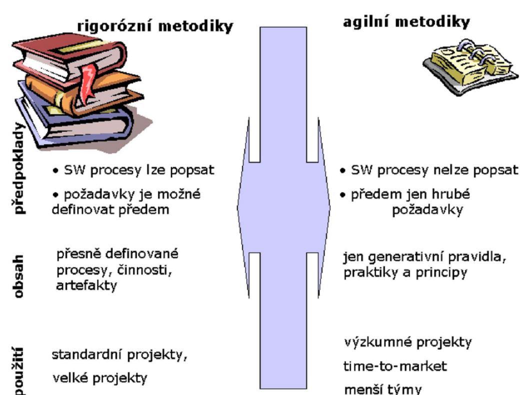
Unified Process (RUP), Enterprise Unified Process (EUP). V rámci rigorózních metodik tvoří samostatnou kategorii metodiky pro hodnocení softwarových procesů (Software Process Assesment). Jsou realizovány zejména v rámci projektu SPICE, který představuje hlavní mezinárodní iniciativu pro podporu vývoje mezinárodního standardu pro hodnocení softwarových procesů. Metodiky hodnocení softwarových procesů jsou založeny na přesvědčení, že kvalita procesu určuje kvalitu produktu, a proto popisují postupy, které umožňují hodnotit úroveň zralosti procesů při vývoji software. Nejznámější z těchto metodik je Model zralosti (Capability Maturity Model).

Změny technologií a ekonomického prostředí, ke kterým v současnosti dochází, a požadavky na rychlé zavedení IS/ICT vyžadují změny v metodikách. Tradiční rigorózní metodiky přestávají v takových podmínkách vyhovovat a začínají se prosazovat metodiky, které umožňují vytvořit řešení velmi rychle a pružně jej přizpůsobovat měnícím se požadavkům. Tyto metodiky jsou označovány jako agilní. Jedná se o různé metodiky, které vznikaly od druhé poloviny 90. let a které prosazují myšlenku, že jedinou cestou, jak prověřit správnost navrženého systému, je vyvinout produkt, nebo jeho část, co nejrychleji, předložit ho zákazníkovi a na základě zpětné vazby upravit. Každá z agilních metodik je svým způsobem specifická, ale všechny jsou postaveny na stejných principech a hodnotách. Proto se sešli představitelé těchto přístupů v únoru 2001, podepsali „Manifest agilního vývoje software" [Fowler,Highsmith, 2001] a vytvořili „Alianci pro agilní vývoj software" [AgileAlliance, 2003].

Agilní metodiky představují ve své podstatě reengineering procesů při budování IS/ICT. Když byl v 90. letech prosazován reengineering v podnikových procesech, dostala jedna osoba (vlastník procesu) plnou kontrolu nad celým procesem. Podobně agilní přístup k vývoji softwaru dává jednomu vývojáři plnou kontrolu nad všemi fázemi procesu vývoje – od přímé komunikace se zákazníkem při sběru požadavků až k realizaci.

Rigorózní a agilní metodiky představují dvě skupiny metodik, které vycházejí z odlišných předpokladů a odlišného pohledu na vývoj software. Výsledkem je jiný obsah a zaměření každé kategorie metodik a jiný okruh projektů, na které je vhodné tyto metodiky aplikovat. Odlišnosti obou přístupů jsou přehledně zachyceny na obrázku 2.

I když jsou východiska, obsah, přístupy i použití rigorózních a agilních metodik na první pohled velmi rozdílné a jejich zastánci vystupují zpravidla antagonisticky, je možné oba přístupy určitým způsobem kombinovat. Rigorózní metodiky je možné odlehčit a aplikovat v jejich rámci některý z agilních přístupů. Velmi zdařilý popis aplikace základních principů agilních metodik v metodice RUP je možné nalézt v [Kroll, 2001]. Dalším příkladem propojování rigorózních a agilních metodik je aplikace metodiky Agilní modelování v RUP, která je popsána v [Ambler, 2001]. Na druhé straně, pokud potřebujeme použít agilní metodiky na větší projekty či projekty větší důležitosti, je třeba je více formali-



Obr. 2 Srovnání rigorózních a agilních metodik [Buchalcevová, 2005]

zovat, zařadit více dokumentace apod. Agilní metodiky jsou v převážné většině zaměřeny na vývoj nového řešení. V poslední době se objevuje snaha aplikovat agilní přístupy i na úpravy řešení a integraci řešení a stejně tak na některé metodiky patřící do kategorie globálních metodik.

4 Závěr

Příspěvek podává obraz současného stavu v oblasti IS/ICT a nejdůležitějších trendů. Je příznačné, že všem těmto tématům jsou věnovány příspěvky na jarní konferenci EurOpen 2005.

Literatura

[AgileAlliance, 2003] http://www.agilealliance.org/articles/index

[Allen, 2002] Allen, P.: *The OMG'S Model Driven Architecture, component development strategies*, Cutter Information Corp., January 2002. Dostupný z WWW: http://www.cutter.com/articles.html

[Allen2003] Allen, P.: *Service–Oriented Architecture Concepts*, Expert's corner LogOn, 6/2003. Dostupný z WWW:
http://www.ltt.de/cgi-bin/down/download.pl?download/experts/allen-07.03.pdf

[Ambler, 2001] Ambler, S.: *Agile Modeling and the Unified Process*, Agile Modeling, 2001. Dostupný z WWW:
http://www.agilemodeling.com/essays/agileModelingRUP.htm

[Ambler, 2002] Ambler, S. W.: *Agile Software Development*, 2002. Dostupný z WWW: http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm

[Beck, 2002] Beck, K.: *Extrémní programování*, Grada, 2002, ISBN 80-247-0300-9

[Bloomberg, 2003] Bloomberg, J.: *Principles of SOA*, Application Development Trends March 2003. Dostupný z WWW: http://www.adtmag.com/article.asp?id=7345

[Buchalcevová, 2002] Buchalcevová, A.: *Agilní metodiky*, In: Objekty 2002, ČZU, Praha 2002, ISBN 80-213-0947-4.

[Buchalcevová, 2003] Buchalcevová, A.: *Model Driven Architecture jako nový přístup k vývoji i integraci aplikací*, In: Systémová integrace 2003, s. 469–476, ISBN 80-245-0522-3.

[Buchalcevová, 2005] Buchalcevová, A.: *Metodiky vývoje a údržby informačních systémů.* Grada publishing, 2005, ISBN 80-247-1075-7.

[CMMI, 2002] *Capability Maturity Model® Integration (CMMISM) – Version 1.1 – Staged Representation*, Technical Report CMU/SEI-2002-TR-012, The Software Engineering Institute, 2002. Dostupný z WWW: http://www.sei.cmu.edu/cmmi/

[Cockburn, 1998] Cockburn, A.: *The Methodology Space*, 1998. Dostupný z WWW: http://alistair.cockburn.us/crystal/articles/ms/methodologyspace.htm

[Cockburn, 1999] Cockburn, A.: *A Methodology Per Project*, Humans and Technology Technical Report, TR 99.04, 1999 Dostupný z WWW: http://crystalmethodologies.org/articles/mpp/methodologyperproject.html

[Drbal, 1997] Drbal, P. a spol.: *Objektově orientované metodiky a metodologie.* skripta VŠE, 1997, ISBN 80-7079-740-1.

[Fowler,Highsmith, 2001] Fowler, M., Highsmith, J.: *The Agile Manifesto*, Software Development, August 2001. Dostupný z WWW: http://www.sdmagazine.com/documents/sdm0108a/

[Gamma, 2003] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Návrh programů pomocí vzorů, Stavební kameny objektově orientovaných programů*, překlad anglického originálu, Grada, Praha 2003, ISBN 80-247-0302-5.

[Gartner, 2004] Predicts 2005: Deploy New Technology, Applications for Success Dostupný z WWW: http://www.gartner.com/resources/124700/124735/predicts_2005_d.pdf

[Highsmith, 2002] Highsmith, J.: *Agile Software Development Ecosystems.* Addison-Wesley, 2002, ISBN 0-201-76043-6.

[Humprey, 1999] Humprey, W.: *Pathways to Process Maturity: The Personal Software Process and Team Software Process.* Dostupný z WWW: http://interactive.sei.cmu.edu/Features/1999/June/Background/Background.jun99.htm

[Jacobson, Booch, Rumbaugh, 1999] Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley, 1999, ISBN 0201571692.

[Johnson, 2001] Johnson, J., Boucher, K. D., Connors, K., Robinson, J.: *Collaborating on Project Success*, Software Magazine, February/March 2001.

[KIT, 2003] *Terminologický slovník KIT.* Vysoká škola ekonomická, Katedra informačních technologií, 2002. Dostupný z WWW: http://www.cssi.cz [12. 11. 2003]

[Kroll, 2001] Kroll, P.: *The Spirit of the RUP*, the Rational edge, 2001. Dostupný z WWW: http://www.therationaledge.com/content/dec_01/f_spiritOfTheRUP_pk.html

[Metagroup, 2003] *Summary of Results 2003 Worldwide IT Benchmark Report*, 2003. Dostupný z WWW: http://www.metagroup.com

[Paulk, 1993] Paulk, M. C., Curtis, B., Chrissis, M. B., Weber, Ch. V.: *Capability Maturity Model for Software*, Version 1.1, Technical Report CMU/SEI-93-TR-024. Dostupný z WWW: http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.024.html

[Polák, Merunka, Carda, 2003] Polák, J., Merunka, V., Carda, A.: *Umění systémového návrhu, Objektově orientovaná tvorba informačních systémů pomocí původní metody BORM*, Grada, Praha 2003, ISBN 80-247-0424-2.

[RUP, 2001] *Rational Unified Process: Best Practices for Software Development Teams*, Rational Software White Paper TP026B, Rev 11/01. Dostupný z WWW: http://www-140.ibm.com/developerworks/rational/library/253.html

[Řepa, 1999] Řepa, V.: *Analýza a návrh informačních systémů.* Ekopress, 1999, ISBN 80-86119-13-0.

[Scrum, 1995] *SCRUM Software Development Process, Building The Best Possible Software.* Dostupný z WWW: http://www.controlchaos.com/scrumwp.htm

[Soley, 2000] Soley, P.: *Model Driven Architecture*, white paper OMG Group, 2000. Dostupný z ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf

[SPICE, 1998] ISO/IEC TR 15504: *Information technology – Software process assessment*, International Standards Organization, 1998. Dostupný z WWW: http://www.sqi.gu.edu.au/spice/

[Voříšek, 1997] Voříšek, J.: *Strategické řízení informačních systémů a systémová integrace*, Management Press, Praha 1997, ISBN 80-85943-40-9.

Using open source in commercial companies

Petr Hřebejk

E-mail: petr.hrebejk@sun.com

Should you care about open source?

This article is about whether you should care about open source and if yes, why? What is Microsoft? A software company which most of the open source guys talk about as evil empire. What is open source? Bunch of guys who Bill Gates thinks are all communists. Neither of this is true. Microsoft really is just a software company. Well, yes they have the monopoly, which definitely is not healthy. But as I will show later it's not Microsoft's fault that they are in a position they are in. If it would not be Microsoft it would be other company but it would be some. Neither is true that all the people participating in open source are communists. They are not. Even if it may not seem so from the distant point of view, except very rare cases, people who write software do it for money. Yes sometimes they simply can't get enough and they write other software (not that they write at work) just for fun. But these fun projects are usually not the main stream even in the open source. But not only that, more and more big companies are going with the open source stream. IBM poured more than 1 billion of dollars into the development of Linux and is giving the Eclipse IDE/platform for free. Sun is sponsoring projects like OpenOffice suite or NetBeans IDE/platform, they open sourced the Solaris operating system. Both companies are very active in the Apache projects. Borland joined Eclipse recently and they opensourced InterBase. Novell bought Suse. Oracle is pretty active in the open source tools space. And you it's easy to find more examples of large companies either participating in open source project or sponsoring some. And of course there are companies/communities based on the open source model only. To name just three look at RedHat, JBoss or MySQL. Of course lots of people are using the open source software and they are just happy. Sometimes even without knowing that they are using open source software. For proof of that just see how big the market share of Linux/Apache is in the server market. Also look at how the number of people using Linux on theirs notebooks/workstations grows. And even if they are still running Windows they usually have at least one open source application on the computer; Mozilla (Firefox,Thunderbird), OpenOffice, Gimp,...

Now, there is a movement producing software in the environment ruled by one big, monopolistic, evil, company, ehich if famous for not hesitating to killing all theirs competitors. How's that? All these projects should be dead for long time, right? But they are not and they are continuing growing theirs market share. If nothing else just this should be a reason why open source is a thing worth of looking at. At least the question: "how this is possible?" definitely has to be asked. For a answers you may for example read the famous Cathedrals and Bazaars article. You will get some there for sure. It explains nicely how it is possible that the open source model can produce some software at all, how people cooperate and how they choose tasks. But in my opinion the most important question remains open. Why the people do that? What drives them? The usual explanation is something like: people like working, people like to please other people, people want to be famous for what they do, people like to be part of a community, etc. However, at least in my eyes, this all simply is not enough to explain why the open source is as big and as successful as it currently is. It also does not explain why companies are joining the movement. Don't tell anyone that: companies like working, companies like to please other companies, companies want to be famous for what they do, companies like to be part of a community. No, companies do not care about things much. Companies like to make money. The better they look and the less they are perceived as evil empires, while doing that, the better. But if they do not look that good or even if people and other companies think that they are an evil empire, no problem, until they are making big enough profit. And even if we sometimes don't like to admit it, the same holds for people. And by the way. it is good so. It makes the world moving.

If we accept the well proved idea that the main reason for working is profit, then there are only few reasons left for the ability of open source to survive. Basically there has to be some future value in going open. Either there is an opportunity to make more money when going open or going open is a way how to defend against loss of money in the future.

Let's look at how the software market looks today. From the open source point of view it mostly looks like Microsoft against the others. Well, in reality most of the companies do both open technologies and Microsoft technologies. But still, the software market definitely does not look like a standard competition driven market. It looks more like a market full of state interventions. But every one knows that, even if there always is space for lobbying, it is not the main cause here. Microsoft is not like Telecom or CEZ here in heart of Europe. The cause is much more simple. The economic science teaches us that when there is a product where the marginal cost (the cost for producing one additional item of the product) is decreasing or constant then a monopoly will develop very likely in the market. Company will not produce more items if the marginal cost is higher than the marginal profit. Most products show growing marginal cost at

some point. At some level of production some kind of cost (work, material, delivery, promotion, capital) will start growing steeply and will push the marginal cost over the marginal profit. The company will stop increasing the number of produced units and there will be a chance for other companies to participate in satisfying the aggregate demand. Not so in software. Cost for producing additional unit is virtually equal to zero. In the Internet age you don't even have to produce a CD. May be you have to create a mirror for your download server but that's too cheap to really be counted. Not even the differentiation in your customer base can't make you worry. May be you can't produce a ideal car for everyone because you would have too many models and it would be too costly. But in software? You simply stuff all the features into your product and the user simply will choose those he likes. Just look at what a spreadsheet can do today. The only problem is that the download size gets bigger. But be realistic, who cares anyway? In such environment the so called natural monopoly will develop, because the competitors will do whatever it takes to kill the other. They are even willing to accept losses for very long periods of time because they know that once theirs monopoly is established they will be able to ÔČĽdictateÔČÝ the price and to earn the monopoly surplus. So Microsoft is not an evil monopoly. It is not the state driven monopoly it is a natural monopoly. This is why I said before that if it would not be Microsoft it would be other company. Microsoft is just the entity which won the race.

But if Microsoft did won over so many companies in last years why the open source still lives? We have to look at the tactics usually used against the competitors First one is the price war. Dump the prices of the products and try to survive longer than your competitors. But this tactics is very hard to apply against someone who's product price already is zero and who does not plan to make it higher at any time in the future. Second tactic is to buy the other company. Not so easy either. In open source there usually is no subject to buy. What is called community is not more and not less than a few hackers spread all of the world. And even if there is some subject which you can buy and if you do you probably won't succeed with the last and more important step. You need to kill or assimilate the rival product. Fortunately open source licenses generally allow so called forking. So what you achieve is that you spend lot of money to buy a competitive product and after some time it will be resurrected by other or even the same group of hackers you just successfully laid off. The last resort is legal. Either you sue your competitor and you bet that your lawyers are better than his. Software patents are useful tool here. Or even better you lobbying in the parliament. You say some bad mouth the competitors and you insist on a law, which will put them out of the business. But again suing an open source community is not that easy. Facing this difficulty most such attempts end up in suing the customer. But rather be careful the customer's money is the money you are faithing for and someone taken to court because of you is probably not

willing to pay for your products in the future. Going the political way may be very dangerous for the image. Political fight against something which may look like a charity from outside is probably not going to have much support in the public. You may end up with people wearing t-shirts with the source code of the banned product printed on it.

It's easy to find examples of how all the before mentioned tactics failed again the open source. I think what's working best is simple again. It's pure marketing. Pouring money into marketing campaigns supporting the product is something what the open source can hardly do and what potentially has a big impact on the customer's decisions. On the other hand it is very expensive. Saying: "Yes, I know the product and I consider buying in the future" (usual check mark option in a questionnaire for evaluating marketing campaigns) and to really paying the money for it instead of downloading free alternative are two different stories.

If you take one kind of applications and if you don't expect the rules of business to change drastically then you should be able to guess that there will only be one winner in the end. It may be open or commercial. In some kinds of applications free software may win in some the commercial products may be the winner. However, the current trend seems to be that more and more software is falling down into the commodity category. I use the term commodity software for such kinds of apps people are not willing to pay for any more or they expect the price to be very low. Operating systems, development tools, databases, web and application servers and even office suites more or less fall into that category already.

After analysis of the economic aspects of the software market it should be clear that there is a reason to give open source a try even if you are working for a commercial company. "OK, but it would mean to give our products for free and that would kill our company in the long run." Not at all. Giving open source a try does by far not mean opensourcing the products. It may or may not be the last step. The decision whether to take the last step or not should be backed up by deep and detailed analysis of what it can bring and what it will take. Some preliminary knowledge and experience is necessary here. The next few paragraph should be a recipe for how to start getting that knowledge and experience.

Start with using open source tools inside your company

Lots of companies already do. You definitely can save some money on operating system licenses by installing Linux. You may start with servers if using application server or web server you may decide to go and install Apache, tomcat,

JBoss or Sun Application server on a Linux box. Also if you are running a database server it does not have to be Oracle. You may be served good enough with something like MySQL or Postgres.

Other kind of server needed by almost every software company is a version control system. Again there are viable alternatives to the commercial products in the open space e.g. CVS or Subversion. Same holds for bug tracking systems, see Bugzilla for example. All these transitions may be relatively easy to proceed with. Maintaining servers is usually task performed by small group of people and they should be fine with bit of experimenting. All these products have conventional (Windows based) clients or a web service, which guarantees that the developer productivity will not be at risk. The standard pushback you can face here is something like: "We rely on these products and if we go the open way we will not get the support." This is not true, what you don't get is the paid hot line, but it may not hurt much. People on hot lines are usually pretty cold and they often cannot help you really. When trying to fix a problem in an open product you will soon realize that simply copy pasting the error message into Google.com will give you more than enough support to fix the problem immediately. Not to mention that you Saving the hot-line and telephone fees.

Other candidate for savings are development tools. Why should you pay for something like compiler today? It is said that the best things are for free. In software it rather reads the good enough things are for free. Sure you might get lot better performance with some paid compiler, but 95 % of applications do not require it and you should be OK with something like gcc, javac, jikes, make, Ant, Maven, etc. Using these tools enables to create things like nightly/continuous build serves bothering people with emails when they break the build. We will get to that later. But you have to be a little bit careful here and make sure that you don't write your products in some proprietary language owned by a company. There is rich choice of safe (means portable)languages e.g. C, C++, Perl, Java, Lisp, SQL, and lot more. Even compiling and running C# on Linux should be possible with a little bit of care. Writing something in Gupta SQLWindows is probably not a good decision though. When you choose a language or languages you probably will need some more tools (like an IDE for example) to be able to do that nice time saving things like refactoring, debugging, profiling, and so on. Also in this space you will find lot of free and good preforming products.

Maybe you will also decide to save some money on your favorite office suite and maybe you will want to try some new tools widely used in the open source space. I really recommend looking at Wiki for collaboration on documents for example.

It may seem that all the cost savings mentioned above are just a nice bonus and staying with a paid product should not harm anyone. But that's a false hope there always is at least one competitor who might have these saving already in place. And a competitor with a lower cost is a dangerous competitor because he

can afford selling his product at lower prices. For the consequences see above.

Start using the open methodologies

I admit, methodologies is a little bit exaggerated. Start small. Read a book about extreme programing and maybe one more about agile programming and decide that something like that is a process you will never implement in your company. Who would like to program in pairs? (Sometimes it is fun) But you have to read the books anyway. First it you will have an interesting conversation topic and second there can be parts of the methodologies which might be useful. For instance writing tests (and running them from time to time) is a good idea. It really helps to have confidence when doing changes in the code. And even other techniques from the open methodologies can be useful for particular projects.

Start "living" openly in the closed environment

If you already have things like CVS server in place and you are using some build system. Dedicate one machine to performing the CVS checkouts and the builds continuously. If the build fails the machine will send an email to a mailing list all developers are subscribed to. Set up a policy which says that breaking a build i.e. making the code uncompilable is a very very bad thing and that no one should do that. OK, no one should do that three times in a row. Broken code base stops all the people from working and the time is money. So everyone is required to run a clean build before check in. After you successfully fought with all the initial pushback and your code is no longer broken make the rules a bit harder. Add a test suite run after the build. This suite should test that the main features of your application are still working. This will add one more safety net. Not only the source base will always be compilable but it also always will run somehow without crashing throwing exceptions. It obviously means that everyone is required to run the testsuite before checkin. And in turn it means that your build scripts and tests framework must be good enough to run on the server and on every developers workstation.

But the build/test server should not be the only machine which will send emails to developers. Version control and bug tracking systems are able to send emails as well and they should. This is one of the main features of the open source development way. It is called living in a fishbowl. Everyone should be able to look at any part of the code and watch what is happening to it. Seeing source is important but seeing the changes in the code is even more important. When someone does a checkin an email is sent to all developers showing the diff. Then there are lot of eyes which can catch the bugs soon enough. It also makes

it easier for the gurus to watch the work of less skilled members of the team. Making a change to the code should be publicly visible action so people will likely not do something what no one should rather see. Mails from bug tracking system help to track the bugfixing efforts. If you are brave enough you might even consider moving the bug tracking system outside of the company and let your users to enter bugs, search the database and subscribe to the lists to be notified that the bug they really hate was already fixed.

Go outside

Become a member of some open source communities. The best way how to do that is to choose the most important open source application you use. May be it will be the one you need an update for most often. Register into the mailing lists of the product. Start discussing things. Sometimes you may ask for help. Sometimes you may give an advice to someone who is currently in trouble. If you find a bug in the product try to find a way how to reproduce it, file a bug in the bug tracking system and be kind to the developers when they ask for some additional info.

Of course if you feel strong/smart enough you may want to try fix a bug in the product yourself. You don't need to worry much. You won't be granted the write access into the CVS repository anyway. You will have to send a patch to one of the developers who already ave the commit right. The developer will check your patch for correctness and if everything will be OK he will integrate the patch instead of you. (He might require you to write not only a the patch but also a test for it. If so, do it.) After few successful patches you may become a regular commiter. Having at least one such person in your company might be crucial if the product you use is so called mission critical application for you.

Start offering open source solutions to your customers

That's the next step. If you feel you have enough knowledge about the products you may want to start offering it to your customers. This may seem a little bit strange. When you offer a choice between a commercial and free product and the customer will decide for the free one, you will loose the provision you usually earn from selling something. Yes, but you still may charge for installation and support. If someone requires 24 by 7 support he can't get it neither from the distant producer of a commercial application nor from the community. In such case a collocated support company is needed, which is an opportunity to make

money on free products. Not charging for the product itself just makes the deal a bit more interesting for the customer and gives you a competitive advantage.

How easy it is to offer free software or products build on free software to customers really depends on which kind of software are you producing. Nowdays more and more software is server based with web based clients. This setup makes the real software behind the web browser virtually invisible to users. However producing thick clients or desktop applications using free software may be a bit harder as most companies still demand Windows based software. But this does not by far not mean that there is no way how to use open source for such products. There is lot of technologies available which successfully run on Windows and Linux.

What you have to watch here carefully however is the various licenses used in the open source space. Some licenses (like the widely used GPL – General Public License) will pose some restrictions on your product. However, there are less restrictive licenses (MPL – Mozilla Public License, SPL – Sun Public License, LGPL – Lesser General Public License, BSD – Berkley Software Distribution, etc.). Some products are also dual licensed, which means that you may use it freely for noncommercial purposes but you have to pay a license fee for commercial usage (SleepyCat's Berkley DB for example). The licensing problems unfortunately make using of free software little bit more complicated than just a grab and go process.

Last step: making you software open source

There is always the possibility to open source your own software. But as I already mentioned you should think twice here. I definitely do not want to discourage anyone from such step, but you still have to be profitable company, right? In next few paragraphs I will describe what you can get from opensourcing a piece of code and about what you probably won't get. You should carefully weight the pros and cons.

First thing is to look at how the market with given kind of applications looks like. It has to be said here that it means how the global market looks like. There is nothing like north bohemian market niche for accounting apps. In the Internet age the software market is almost always global. (Notice the difference between the software market and the market for software services. A north bohemian market niche for maintaining accounting application may very well exist.) If you are the only player on the market. Means that you application has more than 90 % of the market you probably do not need to care about opensourcing it. If there are more players on the market the decision may depend on few other aspects. The application should be splittable into a base product and some value added add-on(s). Most of the money, the reason why people buy the application,

should lay in the add-ons. In this case you may want to open source the base and build a community around it and still make the money on the added value. This might seem easy but in most cases most users will be happy enough with the base functionality and won't pay for the add-ons. You should also be sure that the entry barrier for writing the important add-ons is high enough. If not you may end up watching an enthusiastic developer giving the same functionality for free after a while. Of course licenses, which make your life harder when using the free software, may help here to eliminate some of the dangers. So you might for instance think about dual licensing your product. Other important aspect is: is there already a free alternative of the software you produce and how is it doing? If there already is such a project it might be very hard to introduce a new one and win against the existing one. What is money for commercial products is a community for open source. You will have to fight for people who use, test and codevelop the product. Developers/users time is the scarce resource here. On the other hand if there is no free product and you are not the only player in the market there always is a danger that one of your competitors will do the step sooner than you and that he will succeed. When you analyzed the market and you are still thinking about making at least part of the product open source you might want to look below at what it takes to develop and open product in order disbelieve some myths around the open source and to avoid the later disappointment.

What can open sourcing your products give and take

First myth about open source is that short after publishing the source there will be flourishing community of users and developers, who will be working for free on the product. There won't be any. Look at http://sourceforge.net. It currently host tens of thousands of projects. But only few of them are really active. The fact is that most of the projects will die sooner than they reach the 1.0 version. This is caused by the fact that there is a big difference between clicking on the project's web page and really supporting the project. 100 000 will visit the web page, 10 000 will spend 10 minutes there, 1000 will download the product, 100 will start using it, 10 will fill a bug report and 1 will become a developer. Of course the numbers are not exact but it gives a good perspective of how it works.

Even if you manage to create a community and even if you start dominate the market, you still need to consider the cost of keeping the community satisfied. It is very hard to please all the people and it does cost money. You have to train people how to take care about the community, how to behave on mailing lists,

and that sometimes they have to fix an obscure bug because one of the strong supporters filled it.

You also have to do the guerrilla marketing i.e. making sure that your product is constantly mentioned on relevant forums and webpages. But notice that it does not cost a dollar to download another product, give it a try and maybe change the camp. Even worse users usually do not support just one product when there is a choice. They will download both and use both based on which of them is better for given task.

Another myth is that you will get lot of bug reports, RFEs – Requests For Enhancements and emails with feedback which will help you to improve the product. You definitely will get some. May be lots of them. (If so the question is how you will process them all) The quality of however may be questionable. The people, who yell most loudly on the forums are usually those guys who do not have anything better to do. Smarter folks have less time to discuss things and they will only tell you if there is a very serious problem.

There always is some politics about the open source projects. You have to plan things openly and let people comment. And even if you might be able to drive the project the way you want or need it may make part of the community upset. In case you are building some add-ons or commercial products on the open base you might have hidden agenda which you can't discuss in public.

The worst end of the politics around open source projects is forking. This is double edged sword of the open source. It gives you the security. If you're using open project which dies or which goes in a direction you dislike you can fork and keep maintaining and using the forked version. On the other hand looking how somebody forks your project and the majority of the community decides to go his way may be a bitter experience.

And last final warning here. Don't try to follow an the open source manifestos or extreme open source texts. You might end up thinking that talking to your colleague in the hallway is a sin and that you rather should have written an email. If you find something interesting or potentially useful evaluate it. If somebody tells you something what sounds like ideology feel free to ignore him.

But there of course is not only the dark side. As mentioned above going with the open source movement does not necessarily mean that you would open source your products. Using the open source way of working. Following the usual structure of open source projects. Using the open source development tools can change processes in a commercial enterprise positively. The most important factor of success of a software company today is how effective is the communication and how effectively the company 's cumulated knowledge is used. Having employees communicating on mailing lists and bug tracking system makes the communication cheaper. (Of course there is still place for face to face discussions). But it also makes it possible not to have the whole development placed

in one building. You may have people working from home and you may start even start off shoring the development.

Machine generated emails from version control systems, automatic build machines and bug tracking systems will help to increase the amount of shared knowledge. Shared knowledge is generally good think. It makes people replaceable. Not necessarily in the meaning of laying people off and hiring other ones. Rather you may feel safe when somebody gets sick or leaves. But even more importantly you may create more flexible organization structure. When people are less specialized you may move them freely between various projects.

Summary

A successful company must maximize profits and minimize the risks. Open source can help with both. First: open source products total cost of ownership is usually lower than the TCO of it's commercial counterpart. Second: the open source methodologies effectively lowers the cost of internal and external communication and makes the products more sustainable.

Used sources/recommended reading

[1] Raymond, E. S.: *The Cathedral and the Bazaar*, http://www.catb.org/ esr/writings/cathedral-bazaar/cathedral-bazaar/

[2] Merrill, R. (Rick) Chapman: *In Search of Stupidity: 20+ Years of High Tech Marketing Disasters*, Apress http://www.google.com/search?hl=en&lr=&q=open+source&btnG=Search

Test Patterns In Java

Jaroslav Tulach

E-mail: jaroslav.tulach@sun.com

Abstract

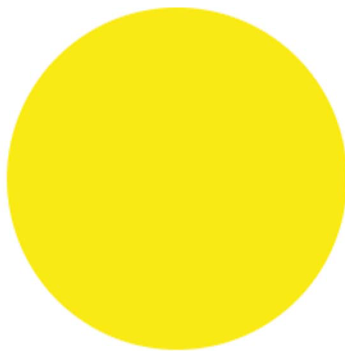
Testing is an important part of software development. Effective testing is a key factor in reducing total cost of maintenance of any application over its life time. It reduces the cost and time of development, can increase savings on quality assurance and of course on sustaining. Knowing when to invest in better design, in post development quality assurance, in manual tests or in automatic testing forms a basic difference between successful and unsuccessful software projects in these tough and competitive days.

In this paper we start with general motivation and present automatic tests as a form of functional specification. It will then quickly deep down into hardcore JUnit test examples showing various forms of regression tests verifying algorithm complexity, memory management, data structure sizes, deadlocks, race condition behavior and tests randomly generating new test cases, simulating user clicks in the UI, API signature tests, etc.

We assume that you know what tests are and that tests are important and useful part of software development. This presentation will show how it looks like when it comes to practical usage of a test framework. You will learn techniques, tips and tricks for testing various aspects of real world J2SE applications. We will also give examples of the savings and improvements that were achieved by increased usage of extensive automatic testing in the development of NetBeans platform and IDE.

What it means quality of an application?

There are many possible answers to question "what makes an application to have good quality?" Depending on one's standpoint the application can be requested to have slick UI, natural work flow, be acceptably fast, not crash from time to time, etc. These are all good expectations, and let's include them under one general category – *specification*. If we are good UI designers, if we can understand the user needs, then we can create good *specification*, which describes how our application should look.



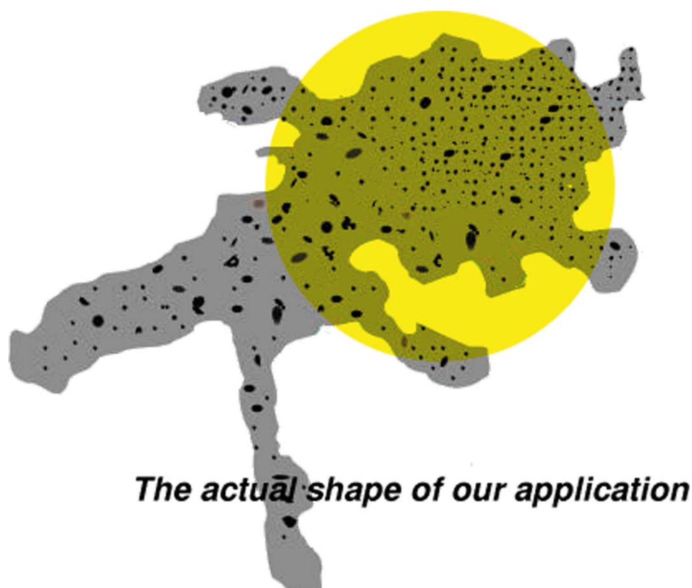
How we think our application looks like

However this does not mean that our users will be satisfied with the quality of what they get. A good enough specification is just a half of what they see. They also need good enough **implementation**. Any expectations we put into our application can or even are (in ideal state) expressed in our *specification*, but before they get to the user they have to be implemented in **code** and it is very likely that the **code** will not follow the *specification* fully. There will be differences between the **code** and the *specification*.

The amount of the differences between our **code** and our **specification** is the measurement of quality. If the final application is not doing what we originally intended it to do, then it is not good enough. Its *quality* or maybe better term is our *confidence* in it is lowered with every difference from the expected behaviour.

Sometimes the application *does not do* what we expect it to do, sometimes it *does more*. Both situations are dangerous, but only one of them is easy to find. One can read thru the spec and test (manually, automatically) if everything that is requested is really implemented. So by carefully testing one's application, one can minimize the places where the **code** offers less than expected. But even this has its limits:

Over the time, with new and new releases, regressions occur. The functionality of the **code** is changing, it starts to do new things and alas, it also stops to do what it used. Of course one can execute the manual test procedures once more with every release, but that is first of all very expensive as people have to try all specified features from all previous releases, and the accuracy of such findings often is not good enough. As a result the shape of the application **code** is changing from release to release as **amoeba** changes its shape over time. That is why we call this behaviour the amoeba model and in the rest of the paper we give advices for ways to fight with its unwanted implications.

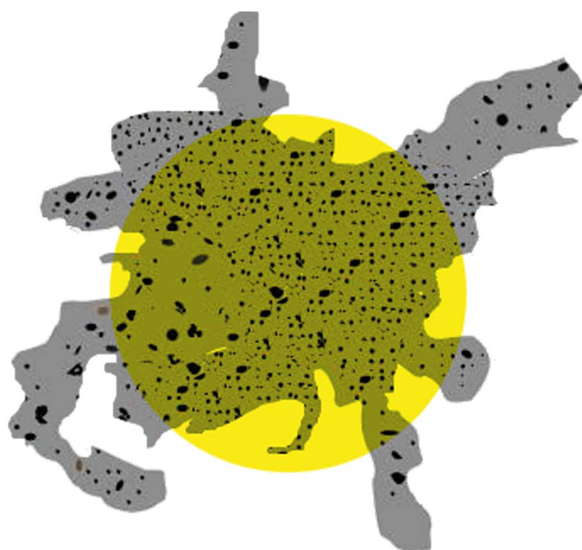


Tests As A Form Of Specification

This paper claims that writing automated tests is beneficial for the quality of an application. But as we are not really XP programmers, we are not going to insist on tests being a must. We believe they are beneficial, especially in certain areas, but sometimes there are other useful and effective ways to achieve good enough quality.

One of the most important things automated tests do, is that they show the *intended use and behaviour* of the applications code. We all know how hard it is to maintain code inherited from other person — read it all, analyze what it does and if a bug is reported, just guess whether that is really a bug or some weird feature (e.g. intended behaviour). As a result, when fixing such code, one has to be affraid whether another hidden feature will not get broken. This whole suffering can be greatly *simplified by automated tests*. If written correctly, they contain the expected calls to the application code and the expected results. One can always check them to find out whether certain use of the code was anticipated or is just a side effect that happens to work.

Automated tests help to fight with the amoeba behaviour. Nearly every change to the application code shakes the shape of what its code does, fixing something and breaking something else at once. By having a chance to execute the tests with an integration of a change in code, one does not prevent the breakages, but at least ensure that the anticipated intended behaviour of an application remains unchanged.



Shape of amoeba after next release

By running various coverage tools (NetBeans use emma) one can find what part of the application code is covered and which one is not. This allows to find and delete code that is not needed or improve the automated coverage for parts of such code and receive more of the benefits associated with having automated tests.

An important social aspect of tests is their support for arrogance. Well, in fact it is not real arrogance, it is more *effective use of manpower*. As by having automated and isolated tests, one can refuse bugs that other want to assign to him by finding or writing tests which mimic the buggy behaviour and proving that the behaviour of ones code is ok and the bug must be somewhere else. This provably lowers the number of hours spend with a debugger and thus more effectively uses time given to programmers.

On the other hand, there are areas which usually do not benefit much from attempts to provide automated tests. User interface is one of such examples. It changes often and automated tests (although possible, see http://jemmy.netbeans.org) are hard to write and are not fully reliable. Sometimes it might be more simple to invest into manual testing. But we have to mention that the NetBeans project has set of fifteenish UI tests to verify basic functionality of the NetBeans IDE and they are executed continuously and they helped us to catch various regressions, but often also just cause false alarms. In spite of all their drawbacks, they are still accepted as valuable.

From a general point of view, the automated tests can be seen as *a kind of specification*. Sometimes it is enough to write UI specification or UML document, for certain aspects it is better to also provide automated tests coverage and sometimes it just does not make sence to do anything else than automated test. We'll give some examples later.

Pragmatism vs. Religion

When talking about testing one has to be very careful. People tend to have strong opinions and discussions easily get pretty heated. And not only heated, often also confused. The whole methodology is still evolving and things just have not settled yet. Due to that let us dedicate a separate section to clear up possible definition confusion.

The interest in automated testing significantly increased during recent years and there is no doubt that this was caused due to all the efforts of the extreme programming movement. The influence is so big, that many people associate testing and **XP** together and feel that if you write tests you are doing **XP** or that if you do not want to do **XP** you should not write tests. This is false impression. **XP** provides a lot of useful tools (like junit) and explicitly talks about the importance of writing unit tests by developers of the application code, but it is much larger methodology which (according to the **XP** proponents) has to be followed fully or not at all. So even we propose usage of tests, whether we do **XP** *or not* **XP** *is not the question* that matters. The important thing is to get confidence in application code we produce.

Another misleading question that is often asked is whether we write unit or functional tests? Well, we use junit and xtest as base for writing and harness for running our tests, but that does not mean the tests are unit ones. As we will demonstrate later, sometimes it is useful to test the application code in isolation, sometimes it is more meaningful to setup as close to production environment as possible. This depends on the aspect we want to verify — in the first case we concentrate more on functionality of a small unit, in the later we check how it behaves and cooperates with other components of the system. Both of these aspects may seem different, but we can use the same tools to check them (e.g. junit and xtest). Again, it does not matter whether we write functional or unit tests, what is important, is to force the amoeba's edge of our application to stop shaking.

Often people disagree when it comes to question who should write the tests and when? Some say that when the application code is written, it should be passed to quality department which will write tests for it. Others (including **XP** folks) insist on tests being written together with the application code. We believe that both opinions have some merits. If an application is tested by someone who

has not code it, it will sure get tested in new and unexpected ways and new and surprising areas where the behaviour of the application (its amoeba shape) does not match the specification will be discovered. Also, it does not make sence to write some kind of tests in advance (for example the deadlock test which we will discuss later) and it is better to wait for a bug to be reported and write the test to simulate the bug and as a proof that it really has been fixed. On the other hand, when tests are written by different folks, it can lead to the infamous we vs. they separation: People writing application code will not understand the tests and those who write the tests will not fully understand the application code and if a test fails these two sides start to blame each other while nobody will really know whether the bug is in test or in application code. This can get really frustrating especially if the failures are intermittent and unreproducible in debugging environment. That is why we share the **XP** opinion that at least some tests shall be written by the application code developers. Beyond the fact that then there is now someone who understands both, the code and the tests, the other reason is that the *application code is different* if written with *testability in mind*.

The biggest improvement of the application code that happens when its developers also start to write the tests is that the application code will no longer be one big monolithic chunk of mutually interlinked code. Instead, for the sake of easy testability and also for the benefit of the application code design, its code gets split into smaller units that can exists separately. As such they can then be tested on its own. So instead of having to start the database, fill it with predefined databases and then launch the whole application to just verify that an invoice entered by a user in a dialog is processed correctly, one separates the invoice handling code from external environment by abstractions. So the invoice code no longer talks directly to database or requires user interface. Instead of that, it relies on an abstract interface to handle its communication with surrounding environment. In the test, then a fake artificatial testing environment is created and which handles all the invoice processing calls, but without all the references to the rest of the application. This is called separation of concerns and there are various supporting techniques one can achieve that (see mock objects, dependency injection using spring and many others), but in NetBeans we are using Lookup. Our code is full of abstraction interfaces like:

```
abstract class DialogDisplayer {
  public abstract void notify(String msg);

  public static DialogDisplayer getDefault() {
    return Lookup.getDefault().lookup(DialogDisplayer.class);
  }
}
```

which separate the caller of its methods from the actual implementation. When

one writes:

```
DialogDisplayer.getDefault().notify("Hello world!");
```

it is unknown which implementation will actually get called when the code is executed. This allows us to provide different implementation in the whole NetBeans application (a one that actually shows a dialog to the user) and different when testing the code, which can either print the message or record it for the rest of the test to check that the expected call really happened. This separation of concerns gives us confidence that our units of code are working correctly. It is true that this is does not imply that the whole application is working correctly, but *if there was no confidence in the smaller parts, there could be none in the whole application.*

The separation of concerns is not the only change that happens when the same people who write the tests can also change the application code. Very often new, otherwise unneeded methods that verify internal state appear. These usually stay package private as the tests are packaged in the same java package and are friends, so no externally visible changes are done, still the code gets more verifiable.

Another change is that the code gets more predictable. Until the only device that consumes output of the application code is the user, it does not matter much whether an action happens now or after few milliseconds later. The human will observe nothing (actually our usual style to solve problems was to post the action to `SwingUtilities.invokeLater` as the human could not realize the delay). This is a coding style that just cannot work with automated testing. The test is as quick as computer and will observe any non-determinism the application code does. So the testing affects the code by either forcing it to remove such non-predictable constructs or by introducing a rendezvous methods that allow the tests and application code to synchronize on the expected state and verify that the behaviour matches the expectations. This is indeed much better then the opposite style of post-poning an operation to happen later and hoping that the human will not realize the delay, once again this helps the application the stiff its amoeba edge.

Properties of Good Tests

The basic and probably only property of any test is that it has to be *useful.* As we are proposing tests as the right way to improve the quality of application code, it has to in some way contribute to its quality. It has to help us shape the amoeba in the way we want.

Regardless of any textual specification and talks made on corridors, sometimes it is not easy to find out what parts of the application code are supposed

to do. Tests help with that. They *show the intention.* If the same programmer who codes the application also provides a test for its behaviour, it is clear that his expectations on input values, on order of calls, throwing exceptions, etc. are going to be visible in his tests. Later, when another person looks at the code, it can sort out more easily what works just by a chance and what is the essencial and desired behaviour of the application code.

It is good if clever people develop your application and are able to fix bugs. It is however much better if your application is being maintained by *careful* people — e.g. those that care about the application's future a lot and rather invest more work now to prevent the same or similar problem to reappear later. Those that are just more careful and do not want to spend their life just by patching, patching, and patching broken code. In such environment, every bug can be seen as a possible report that the behaviour of the application does not match our expectations. Such bug is a chance to get the shaking amoeba edge of the application code closer to the specification. But only if one together with the fix also prevents regressions (by writing test) the shape of amoeba is going to be stiff enough to not regress and recreate the same bug in future.

If we have enough tests and we want them to really prevent regressions, we need to run them regularily to discover their failures. Whether they shall be executed once per release, every week, every day or instantly depends on the size of your project and on the time the execution of tests consumes. This depends on how soon you want to catch the regression. The the sooner it is discovered, the less painful is the identification of the change in application code that caused it. In NetBeans we have a set of tests, that take about five minutes and are supposed to be executed before every commit to our shared source base, we call them commit validation. Every developer is supposed to verify that his changes are not *really bad* and do not break some basic functionality of the NetBeans IDE. By warning that something is broken before it really gets integrated we minimize the amount of work needed to investigate what is causing the regression. However, these tests cannot include all of our tests that would run for hours, so we also have a suite of tests executed daily that is using the xtest harness to run the tests enumerated in our master configuration file and a each module's individual configuration file. This ensures that every regression is reported within 24 hours after integration and that is why one needs to evaluate only the changes made in previous day to find out the cause.

Of course, if people use tests written by someone else to verify whether their own work is correct (as we do in commit validation), the tests *must be reliable.* Failure must mean that something with my changes code is wrong. If this is not true, then people start to loose trust to the whole system and as such, it is better to remove randomly failing tests from validation suites as quickly as possible.

Althrough **XP** suggests to avoid that, sometimes certain assumptions are

made about the whole system and not just individual parts — e.g. source file. Such *intentions* are really hard to discover by reading single source file. This forms a maintanance problem as even if you have clever people that can read and understand the code, they may not realize that it is necessary to also locate some distinct source file and it as well. Only one change then and a regression appears! Of course if the assumption about the system is expressed in automated test, you will get notification. In some sence the tests can be used to link *physically distinct sources* that are logically connected together. This gets especially important when one does *hacks* around bugs in foreign code. Writing a hack is ugly, but if that is the only solution to some bug, there can be strong push to do it. If it works, ok. But as usual with our amoeba, we want to be sure that the hack works with new versions of code we own. And as usual an automated test for the hack functionality is the cheapest way to ensure that. We used this *hack verification test* for check that our workaround for certain weakness in default implementation of clipboard works. Not only that the test verified the behaviour on JDK 1.4, but also caught a change in the clipboard handling in a beta version of JDK 1.5 and we could negotiate some changes with the JDK team to make it work again.

So the only property of good test is its usefulness. It can help us to show the intention, prevent regressions, verify connections of unreleated code or even hacks into foreign code and sometimes it can serve as an example (we have a shoping cart application with few checks around). Does not matter how, but if it helps to bring the amoeba's edge closer to our expectations, then it has to be good.

Testing Framework

When searching our memory we can realize that we wrote tests long, long time ago. The evidence consists of all the `main` methods in our application sources hidden in comments or sometimes even left in the code that setup the part of the application in a specific way so one can just verify his code and do not be forced to launch the rest. This indicates that programmers really like to write tests and like the separation of interest, but still, for writing automated tests, it is better to use an exisiting framework than to encode the testing logic into main methods spread everywhere. And if we need testing framework in Java, then we very likely need junit. It is the flag ship of the **XP** in Java, and even it is not masterpiece of art, it works well, is easy to use for simple cases and higly customizable if one needs more complex behaviour.

Tests are written in the separate classes, in the same packages and are usually named as the class they test with postfix Test:

```
public class MyClassTest extends TestCase \{
  private int variable;
  private static int counter;

  static {
    // one and only initializations
  }

  public MyTest (String name) {
    super (name);
  }

  protected void setUp () throws Exception {
    // do some kind of setup
    variable = counter++;
  }

  protected void tearDown () throws Exception {
    // clean up and possibly post verifications
  }

  // The tests
  public void testFirstInvocation () throws Exception {
    assertEquals ("First means 0", 0, variable);
  }

  public void testSecondInvocation () throws Exception {
    assertEquals ("Second is 1", 1, variable);
  }

}
```

When executed by a harness, the static initialization is called once, then an instance of the test is created for each method with prefix test. The sequence of `setUp`, test method, `tearDown` is then called on each instance. The test is expected to perform some computations and then verify if the result is correct by using various predefined asserts like `assertTrue`, `assertEquals`, `assertSame` or unconditional `fail`. If all tests passed, the harness succeeds, otherwise it reports an error.

The junit is simple and extensible. That is why the NetBeans is using our own extension call `nbjunit` which instead of `TestCase` provides NbTestCase with a few additional useful asserts. The extension can be easily used by anyone, just run `NetBeans 4.1`, select Tools, Update Center in menu and download it. Then you can add the library to any of your project test classpath and start to use it.

A typical NetBeans not only uses NbTestCase, but also Lookup to handle the sepration of concerns, so the static one time only initializer registers test's own implementation of the `Lookup` providing the environment needed for the test:

```
public class MyTest extends NbTestCase {
  static {
    System.setProperty ("org.openide.util.Lookup", "MyTest$Lkp");
  }

  public MyTest (String name) {
    super (name);
  }

  public static final class Lkp extends org.openide.util.lookup.AbstractLookup {
    public Lkp () {
      this (new org.openide.util.lookup.InstanceContent ());
    }

    private Lkp (org.openide.util.lookup.InstanceContent ic) {
      super (ic);
      ic.add (new DD ());
      ic.add (...);
    }
  }
}
```

Of course junit is flexible enough, so one can use other extensions, and possibly also a different harness (recently there was a lot of buzz around TestNG), but as our example are going to use the `NbTestCase`, we thought it is reasonable to explain our specific terminology.

When there is enough tests

While writing tests, people can ask, how much of them should be written? The simple answer is to write tests, while they are useful. The more precise, more complex and less clear answer is going to be answered in this chapter.

There are various tools out there that help to measure test coverage. We have selected emma for measuring the coverage of our application code by our tests. When invoked (for example from the popup menu of any project from NetBeans.org) it intstruments the application code and invokes automated tests on it. While running it collects information about all called methods, visited classes and lines and then it shows summary in a web browser.

Counting coverage by visited methods is very rough criteria, but it can be surprisingly hard to get close to 100 %. But even if you succeed, there is no guarantee that the resulting application code works correctly. Every methods has a few input parameters, and knowing that it succeeded once with one selection of them, does not say anything about the other cases.

Much better is to count the coverage by branches or lines. When there is a `if (...) { x(); } else { y(); }` statement in code of your method, you want to be sure that both methods, x and y will be called. The emma tool supports

this and by helping us to be sure that every line is visited, it gives us confidence that our application code does not contain useless lines.

Still, the fact that a line is visited once, does not mean that our application code is not buggy.

```
private sum = 10;
public int add(int x) {
  sum += x;
}
public int percentage(int howMuch) {
  return 100 * howMuch / sum;
}
```

It is good that each method if both methods get executed, and fine if we test them with various parameters, still we can get error if we call `add (-10); percentage(5)`, because the `sum` will be zero and division by zero is forbiden. To be sure that our application is not vulnerable to problems like this, we would have to test each method in each possible state of memory it depends on (e.g. each value of `sum` variable) and that would give us the ultimate proof that our application code works correctly in single threaded environment.

But there is another problem — java is not single threaded. A lot of applications starts new threads by itself, and even if they do not, there is the AWT event dispatch thread, the finalizer thread, etc. So one has to count with some amount of non-determinism. Sometimes the garbage collector just kicks in and removes some unneeded objects from memory, which can change the behaviour of the application — we used to have a never ending loop, which could be simulated only if two mozilla browsers and evolution client was running as then the memory was small enough to invoke the garbage collector. This kind of coverage is unmesuarable.

That is why we suggest people to use code coverage tools as a way to sanity check that something is not really undertested. But it is necessary to remind ourselves that however big the coverage is, it is not prevent our application code fully from having bugs. So we, in order to help to fight the strange moves of application amoeba shape, suggest to write a test when something gets broken — when there is a bug report, write a test to verify it and prevent regressions. That way the coverage is going to be focused on the code where it matters — the one that really was broken.

AWT Testing

User Interface automated testing is hard. Hard especially in Java that runs on variety of platforms, each with slightly different behaviour — sometimes keyboard focus follows mouse, sometimes it does not, swing comes with different

look and feels and our automated tests, in order to be useful, has to overcome this and work reliably. And this is hard to achieve.

Rule #1 is clear: Avoid AWT testing. Separate your application code into two parts — application logic and actual presentation. Test your models separately from the UI — writing automated tests for them shall be possible, as they are the models and are independent from the presentation (of course if the design follows model-view-controller separation, but swing does). For example instead of trying to write a test for your checkbox, write a test for its ToggleButtonModel, make sure it works and then let the UI to simply delegate to that model (this is what we did when writing StoreGroup and its tests). That gives you confidence in your code as the logic is tested and the UI is dumb simple that manual test once a release is enough to guarantee it works as it should.

Sometimes it is possible to avoid showing UI components, but one has to run the test inside of AWT dispatch thread, because the code follows the Swing threading model (e.g. everything in AWT thread). The simplest any library independent approach for this is to use `invokeAndWait`:

```
public void testSomething () throws Exception {
  javax.swing.SwingUtilities.invokeAndWait (new Runnable () {
    public void run () {
      callMethodThatDoesTheTest ();
    }
  });
}
```

The logic of the test method is in `callMethodThatDoesTheTest` and it is executed in AWT. The above example however does not handle exceptions correctly and if we try to, the code gets a bit more complicated (see run method in NbTestCase) and that is why we allow tests to just override `runInEQ` and the NbTestCase handles the rest for them automatically:

```
public class MyTest extends NbTestCase {
  protected boolean runInEQ () {
    return true;
  }
}
```

See for example CookieActionTest.java that needs to run in AWT event thread as it tests Swing-like action implementation.

In some cases, the test code itself runs outside of AWT event queue, but somewhere inside the application code certain actions are posted to AWT. In such case it may be handy (and often necessary) to wait for the execution to finish, before continuing in the test. Then following method may be useful:

```
private void waitEQ () throws Exception {
  javax.swing.SwingUtilities.invokeAndWait (new Runnable ()
```

```
            { public void run () {} \});
}
```

It posts empty Runnable into the AWT event thread and waits for it to finish. As the queue of runnables is FIFO, the runnable is scheduled at the end after all tasks posted by the application and when it is finally executed one can be sure that all delayed tasks of the application in AWT event queue are over as well. See DataEditorSupportTest.java for an example of a test that needs to wait while the application code finishes some actions posted to AWT event thread.

There are situations when the generic mock object approach can be useful for AWT testing as well. For example in order to test UI environments that do not support custom cursor definition the UtilitiesTest defined its own AWT Toolkit that does not support custom cursors. In another example (see CloneableEditorUserQuestionTest.java another mock object for `DialogDescriptor` is used to fake the communication with user. It replaces the `DialogDescriptor`, which in production environment shows dialog and interacts with a user by showing a UI component with a headless implementation that returns immediately preset values and thus allows automated verification of application code that itself communicates with humans.

If you cannot split your code into logic and UI and you absolutely have to write automated tests, then use Jemmy. It is a junit extension that operates on realized UI components and allows automated navigation on dialogs. An excellent introduction to jemmy can be found at Jemmy Testing Toolkit presentation.

Algorithm Complexity Tests

A very important but fragile piece of functionality that really deserves automated testing is performance. Nearly everyone finds out that something seems to be slow in the application. The natural resolution is to get the profiler, apply it on the application, find out what is wrong and fix it. This works, and we all do it, but the question is how much is this effective. As far as we know the profiling is usually done when the application code is ready and all features are implemented. When a performance improvements are applied at that time, it is clear that they will stay till the release (as all other integrations are already done). That means that we shaped our amoeba for the release, but what will happen by the next one, will not the amoeba shape change again?

Of course it will! After the release (and all the profiling effort) a new round of feature integration starts and for a certain point in time nobody will care about the performance. The new code for sure changes expectations of the old one and very likely negates the performance improvements made in the hectic part during the end of previous release. So it is the time to take profiler, find

what is wrong, provide improvements that make the application state acceptable enough and start the whole vicious circle one again. Are you surprised? You should not, it really works this way. Is there a better way? Yes, let's analyze it.

During the hectic profiling time, when one uses profile to find hotspots and provide speed ups to them, one invest a bit of time to write a *speed test* to demonstrate what is wrong. That test will first of all serve as a proof that the performance problem has been fixed, but (most importantly) also as a continuous reminder that the intention of this code is to be fast and as a warning that some day this intention was broken. This will prevent the amoeba to regress and safe a lot of work when new release is being profiled.

The basic idea behind speed test is simple. Just execute the same algorithm on data set of different size and compare that the time matches our expectations (e.g. it is constant, linear, quadratic, etc.). This can easily be written in any test harness, including plain junit. The following test is trying to access the middle element of a linked list and checks that the access time is constant — well it checks whether the slowest time is three times slower that the first one:

```
public class WhyIsTheAccessToListSlowTest extends TestCase {
  private int size;
  private List toCheck;
  private long time;
  private static long one = -1;

  protected void setUp () {
    size = Integer.valueOf (getName().substring (4)).intValue();
    toCheck = new LinkedList (Collections.nCopy ("Ahoj", size));
    time = System.currentTimeMillis();
  }

  protected void tearDown () {
    long t = System.currentTimeMillis() - time;
    if (one == -1) {
      one = t;
    } else {
      if (t > one * 3) {
        fail ("The time is just too long");
      }
    }

  private void doTest () {
    for (int i = 0; i < 10000; i++) {
      toCheck.get (size / 2);
    }
  }

 public void test10 () { doTest (); }
 public void test100 () { doTest (); }
 public void test1000 () { doTest (); }
 public void test10000 () { doTest (); }
```

This works and really can discover that access to middle of linked list is too slow, but such measurements can be influenced by various external events: first a garbage collector can be invoked during one of the tests and effectively stop the execution, making the time measurement too big. Or, a hotspot compiler can step in and decide to compile the application code to make it faster. As a result one of the tests will just take longer time as the hotspot compilation will slow it down, and the later tests executed after it will be faster as they will run the compiled code, much faster than the interpreted one. We have really observed such random failures and that is why we created `NbTestCase.speedSuite` a junit like wrapper around our test cases that can execute the tests more times to eliminate the influence of garbage collector and hotspot compiler. The results are excellent, just by allowing the test to restart itself few times in case of failure the indetermistic factors of external environment were eleminated and we we had no random failure in our speed tests for more than a year. Here is the previous test rewritten in our speed suite style:

```
public class WhyIsTheAccessToListSlowTest extends NbTestCase {
  private int size;
  private List toCheck;

  public static NbTestSuite suite () {
      return NbTestSuite.speedSuite (
         WhyIsTheAccessToListSlowTest.class, /* what tests to run */
         10 /* ten times slower */,
         3 /* try three times if it fails */
      );
  }

  protected void setUp () {
    size = getTestNumber ();
    toCheck = new LinkedList (Collections.nCopy ("Ahoj", size));
  }

  private void doTest () {
    for (int i = 0; i < 10000; i++) {
      toCheck.get (size / 2);
    }
  }

  public void test10 () { doTest (); }
  public void test100 () { doTest (); }
  public void test1000 () { doTest (); }
  public void test10000 () { doTest (); }
```

An example of such test from NetBeans code base can be found for example at DataShadowSlowness39981Test.java and we can confirm that it helped us prevent regressions in our application amoeba edge.

Memory Allocation Tests

One of the least specified things in conventional project documentation is memory model of your application. It is not that surprising given the fact that there is no generally known metodology how to design the memory model of an application. However it is very surprising as writing long time running application without ensuring that they manage their memory effectively and do not allocated more and property deallocate what is not needed, one can hardly write an application that is supposed to run for days.

Again, the classical model is to code the application, start profiler and search for possible memory leaks. When found, return back to the code, fix it and on and on and on, until the application is in a releaseable shape. And again, as described many times, this is not effective if one plans to release more than one version of the application. As after a time all the improvements from the profiler hunting phase which helped to ensure the amoeba shape is better will regress. Unless makes sure they are continuously tested.

The standard junit does not offer much in this area, but we have to write few extensions for our NbTestCase both based or supported by our memory inspection library called Insane.

The first thing one has to fight against regarding memory management in modern object oriented languages like java are memory leaks. The problem is not that an application would address unknown place in memory, that is not possible due to garbage collector, but sometimes (also due to garbage collector) objects that one would like to vanish, still remain in memory. If certain operation always leaves garbage after its execution, after a few executions one can find, that the free memory is shrinking and shrinking and the whole application is slower and slower. For this the NbTestCase offers method `assertGC`:

```
Object obj = ...;
WeakReference ref = new WeakReference (obj);
obj = null;
assertGC ("The object can be released", ref);
```

If you believe that after some operation an object shall be no longer needed in memory, you just create a `WeakReference` to it, **clear your reference to object** and ask the `assertGC` to try to release the object from memory. The `assertGC` tries hard to force garbage collection of the object, it does few `System.gc`, allocates some memory, explicitly invokes finalizers and if the `WeakReference` is cleared, successfully returns. If not, it invokes the insane library and asks it to find a reference chain that keeps the object in memory. Possible failure could then look like:

```
junit.framework.AssertionFailedError: Represented object shall disappear as well:
private static final java.lang.ref.ReferenceQueue
```

```
         org.netbeans.modules.adaptable.SingletonizerImpl{\$}AdaptableRef.QUEUE->
java.lang.ref.ReferenceQueue@17ace8d->
org.netbeans.modules.adaptable.SingletonizerImpl$AdaptableRef@bd3b2d->
org.netbeans.modules.adaptable.SingletonizerImpl$AdaptableRef@14653a3->
java.lang.String@130c132
         at org.netbeans.junit.NbTestCase.assertGC(NbTestCase.java:900)
         at org.netbeans.modules.adaptable.SingletonizerTest
               .doFiringOfChanges(SingletonizerTest.java:177)
         at org.netbeans.modules.adaptable.SingletonizerTest
               .testFiringOfChangesOnAllObjects(SingletonizerTest.java:116)
         at org.netbeans.junit.NbTestCase.runBare(NbTestCase.java:135)
         at org.netbeans.junit.NbTestCase.run(NbTestCase.java:122)
```

which can be read as there is a static field `QUEUE` which points to `ReferenceQueue` and it thru two `AdaptableRef` holds the `String` whcih we wanted to garbage collect in memory.

Another thing that may affect performance of an application code is the size of its data structures. If you known that a certain object is going to be simultanously kept in memory in thousands instances, you do not want it to occupy 1000 bytes or more. You want to minimize its size. Again, this can be observed in profile, or this can be well in advance thought decision, but the usual problem remain — we need to ensure that from release to release the size constraint will not regress. For that our NbTestCase provides `assertSize` check:

```
class Data {
  int value;
}
Object measure = new Data();
assertSize ("The object is small", 16, measure);
```

It uses the insane library to traverse the graph of all objects referenced from the `measure` variable and computes the amount of occupied memory. Then it compares the value with the expected one and if lower or equal, it passes. Otherwise it fails, printing sizes of individual elements to let the programmer analyze the failure:

```
junit.framework.AssertionFailedError: Instance is small: leak 8 bytes over limit
               of 64 bytes
  org.netbeans.modules.adaptable.SingletonizerImpl{\$}AdaptableImpl: 1, 24B
  org.netbeans.modules.adaptable.SingletonizerImpl{\$}AdaptableRef: 1, 32B
  $Proxy0: 1, 16B
        at org.netbeans.junit.NbTestCase.assertSize(NbTestCase.java:937)
        at org.netbeans.modules.adaptable.SingletonizerTest.
               testProvidesImplementationOfRunnable(SingletonizerTest.java:58)
        at org.netbeans.junit.NbTestCase.runBare(NbTestCase.java:135)
        at org.netbeans.junit.NbTestCase.run(NbTestCase.java:122)
```

So it can be seen that the `AdaptableImpl` references `AdaptableRef` and `Proxy` which together with their fields consume 72 bytes, which is more than expected 64.

The size of simple `java.lang.Object` instance which has not fields is 8 bytes. When adding one field with integer or reference to other object the size increases to 16 bytes. When adding second such field, the size stays at 16 bytes. The third one however increases it to 24 bytes. From that it seems that it make sence to optimize round up the number of fields in an object to two, and we are sometimes, in really sensitive places, really doing that. However it has to be noted that the computed sizes are logical ones, the actual amount of occupied memory depends on the implementation of virtual machine and can be different, but that shall be ok, as the test of logical size expresses the intention of the programmer which is independent on the actual virtual machine architecture.

We have found both `assertGC` and `assertSize` very valuable in stiffing the edge of our application's amoeba. By writing tests using these asserts we can *specify* the expected behaviour of our application. So these tests became part of our *functional specification*, and not only that, being automated tests, they are *active specification* that verifies its validity everytime we execute them.

Randomized Tests

Most of the ways to test the application code we have discussed are useful when you find out that something is wrong and you want your fix to last and stiff the amoeba shape closer to the desired look of the application. Tests usually does not help much in discovering differences between the specification and reality. The one exception however are randomized tests — they help to test the code in new, unusual way and thus can discover new and unusual problems of the application code.

The basic idea is simple, just use random number generator to drive what your test does. If, for example, if you support operations `add` and `remove` use the generator to randomly specify their order and parameters:

```
Random random = new Random ();
int count = random.nextInt (10000);
for (int i = 0; i < count; i++) {
  boolean add = random.nextBoolean ();
  if (add) {
     list.add (random.nextInt (list.size (), new Integer (random.nextInt (100)));
  } else {
    list.remove (random.nextInt (list.size ());
  }
}
```

This will not invent new ways to call your code, just new order of calls and even that can reveal surprising problems, because not all combintations of operations have to be anticipated by the programmer and some of them may lead to failures.

Important feature of each test is its reproducebility or at least clear failure report. It is fine that we know there is a bug in our code, but if we do not know how to reproduce it, we may not be able to analyze it and fix it. The reproducebility of random tests is even more important, as in fact, we do not know the sequence of computations that is really being performed. First step to achieve it is to do not create random generator blindly, but use initial seed which, when passed repeatadly generates the same sequence of numbers. If you look at the implementation of the default `Random` constructor, you will find out that the initial seed is set to the current time, so we can mimic the behaviour by writing:

```
private void doRandomOperations (long seed) throws Throwable {
  Random random = new Random (seed);
  try {
      // do the random operations
  } catch (AssertionFailedError err) {
      AssertionFailedError ne = new AssertionFailedError (
          "For seed: " + seed + " was: " + err.getMessage ()
      );
      throw ne.initCause (err);
  }
}
public void testSomeNewRandomScenarioToIncreaseCoverage () throws Throwable {
  doRandomOperations (System.currentTimeMillis ());
}
```

which knows the initial seed and prints as part of the failure message if the test fails. In such case we can then increase the coverage by by adding methods like

```
public void testThisUsedToFailOnThursday () throws Exception {
  doRandomOperations (105730909304L);
}
```

which exactly repeats the sets of operation that once lead to a failure. We used this approach for example in AbstractMutableLazyListHid.doRandomTest.

One problem when debugging such randomized test is that by specifying one number a long sequence of operations is defined. It is not easy for most people to imagine the sequence by just looking at the number and that is why it can be useful to provide better output so instead of calling `doRandomOperations (105730909304L);` one can create a test that exactly shows what is happening in it. To achive this we can modify the testing code not only to execute the random steps, but also generate more usable error message for potential failure:

```
private void doRandomOperations (long seed) throws Throwable {
  Random random = new Random (seed);
  StringBuffer failure = new StringBuffer ();
  try {
```

```
    int count = random.nextInt (10000);
    for (int i = 0; i < count; i++) {
      boolean add = random.nextBoolean ();
      if (add) {
        int index = random.nextInt (list.size ();
        Object add = new Integer (random.nextInt (100));
        list.add (index, add);

        failure.append (" list.add(" + index + ",
               new Integer (" + add + "));\n");
      } else {
        int index = random.nextInt (list.size ();
        list.remove (index);

        failure.append (" list.remove(" + index + ");\n");

      }
    }
  } catch (AssertionFailedError err) {
      AssertionFailedError ne = new AssertionFailedError (
          "For seed: " + seed + " was: " + err.getMessage () +
               " with operations:\n" + failure
      );
      throw ne.initCause (err);
  }
}
```

which in case of error will generate human readable code for the failed that like:

```
list.add(0, new Integer (30));
list.add(0, new Integer (11));
list.add(1, new Integer (93));
list.remove (0);
list.add(1, new Integer (34));
```

We used this technique to generate for example AbstractMutableFailure1Hid.java which is long, but more readable and debuggable than one seed number.

The randomized tests not just help us to prevent regressions in the amoeba shape of our application by allowing us to specify the failing seeds, but also, which is a unique functionality in the testing, it can help to discover new areas where our shape does not match our expectations.

Reusing Tests

The `junit` framework provides a lot of freedom and allows any of its users to customize it it in nearly unrestricted ways (so we could create the NbTestCase). The standard way of writing tests by prefix method name with `test` does not need to be followed and one can create its own style. Sometimes that is useful, sometimes necessary, but often the build-in standard is enough, because it

is well thought and offers a lot. Even multiple reuse of one test in different configurations.

The simplest way of reusing a test is to let it call a protected factory method in one class and overwrite it in a subclass with different implementation:

```
public class WhyIsTheAccessToListSlowTest extends NbTestCase {
  private List toCheck;

  // blabla

  protected void setUp () {
    size = Integer.valueOf (s.substring (5));
    //
    // calls the factory method to create the actual
    // instance of the list
    toCheck = createList (size);
  }

  // here comes the testing methods
  // imagine some that use the toCheck field initialized in setUp

  /** The factory method with default implementation.
  */
  protected List createList (int s) {
     return new LinkedList (Collections.nCopies (s, "Ahoj"));
  }
}

public class WhyIsArrayListFastTest extends WhyIsTheAccessToListSlowTest {
  protected List createList (int s) {
    return new ArrayList (Collections.nCopies (s, "Ahoj"));
  }
}
```

This example creates two sets of tests, both running over a `List` but in both cases configured differently. This could be also done in a more advanced way by useage of factories and manual test creation as we did in AbstractLookupBaseHid.java, AbstractLookupTest.java, ProxyLookupTest.java, but for a lot of cases, the build in inheritance in junit is enough. If used one can easily get twice as much tests, covering twice as much scenarios.

Writing one test and using it more configurations can be very useful when one has a family of various implementations of the same interface that can be assembled by the final user into various more and more complicated configurations. Imagine for example that one writes an implementation of `java.io.InputStream` and provides a test to verify that it works correctly. But in real situations, the stream is not going to be used directly, it will be wrapped by `java.io.FilterInputStream` or its subclass. That means we want to write another *layer of test* that executes the same operations as the previous

test, but on our steam wrapped with `FilterInputStream` and yet another by a `FilterInputStream` with overriden some methods. If these tests work, we will have bigger confidence that our implementation will really work on various configuration. But then we realize that usually the stream will also be wrapped with `java.io.BufferedInputStream` for performance reasons. Well, that is easy, to ensure that everything will work smoothly, we just create new *layer* and configure the test to use `BufferedInputStream`.

We have used this technique in implementation of `javax.swing.ListModel` with a very special semantic, that itself is pretty complex (LazyListModel-Test.java) and required a lot of testing, but then it needs to be propagaged thru various layers thru our APIs with unchanged semantics, so we added three additional *layers* (LazyVisualizerTest.java, LazyVisualizerOverLazyChildrenKeysTest.java, LazyVisualizerOverLazyChildrenKeysAndFilterNodeTest.-java). Whenever there was a failure, we could immediatelly find out which part of our code needs a fix. If all four tests failed, then the problem was in the basic alghoritm, if the basic test passed, and one of the more complicated did not, we immediatelly know which layer in our code needs to be fixed. This was very handy for debugging. One did not need to step through behaviour of all tested code, it was enough to just pay attention to code in the problematic layer.

This kind of setup has also been found very valuable for randomized tests. Whenever there was a failure, we recorded the seed and created a fixed test repeating the behaviour of the failed random test. And again, if all four suites failed, we know that the basic algorithm is bad, if some of them worked, we knew which part of the application investiage to find the problem.

A surprising place where the *layered* style tests can be helpful is the eternal fight between people who want more reuse of the code and people that are affraid to allow it. We usually require at least three different uses of a certain functionality before we even consider to turn it and maintain it as an API. The reason is that the reuse is associated with significant additional costs. One needs to find a reasonable generalization that suites all the known uses and one has to be more careful when developing such publicly shared part (e.g. write more tests), otherwise things can quickly get broken and the whole benefit of code reuse is gone. That is why we sometimes suggest to *copy the code* instead. We know that *copy based programming* is not nice and that is has its own problems, but sometimes that is really more convenient. Especially when its biggest problem — i.e. the possibility that the copied code gets *out of sync* — can be easily overcomed by a test. Supposed that the original code is well tested, then it is not hard, when copying the code to modify its test to use factory for creation of the tested object and together with the code create also a test that changes the setup to create object matching the new scenario and executes all the old tests on it. We used this in TopComponentGetLookupTest which sets up its environment in `setUp` method and defines set of tests that shall pass. The

test is then extended by different module's ExplorerUtilCreateLookupTest. This effectively prevents the code from getting out of sync. If any future fix in the original code changes the behaviour or adds new feature covered with tests then the next run of automated tests will fail on the copied code. As the inherited test executes the same operations on the currently out-of-sync copied code as we will be properly notified to update our copy of the application code.

More *layers* of the same tests is a very valuable and powerful pattern that not only helps to form the amoeba into more desirable shape, but also effectively uses the invested work into writing one test, as it exploits its power in more situations and help to more quickly analyze the area that caused the test fail.

Testing Foreign Code

Whenever one designs an interface that others can implement, one exposes himself to possible problems caused by wrong implementations. It is very likely that at least one implemetor will not do everything correctly and something will go wrong. We have faced that with our generic wrapper virtual file system api. It provides a generic API to access resources of regular operating system files, in ZIP and JAR archives, version controlled files, ftp archives, and many more. Clients work with just the API and if there is a bug it ends up reported against the generic framework, regardless of the actual implementation that is often responsible for the faulty behaviour. To prevent this another step of *layered* tests can be used as a very good solution.

The provider of the API that allows other implementors to plug-in, can write a generic set of tests that describe the properties that each implementation shall have. These tests, contain an abstract factory interface that implementors shall provide together with their implementation of the application code. Their factory sets up the environment for their code to work and the rest of the test then verifies on the presetup object that all the required properties are satisfied. This is often refered as **TCK** — test compatibility kit. In a way a *layered* test, but with a generic interface, not knowing all the implementors that are going to reuse it.

The already mentioned virtual file system library has such **TCK** which heart is formed by a factory class with one create and one cleanup method:

```
public abstract class FileSystemFactoryHid extends NbTestSetup {
  protected abstract FileSystem[] createFileSystem(String testName,
               String[] resources) throws IOException;
  protected abstract void destroyFileSystem(String testName) throws IOException;
}
```

All the test use these methods to get the `FileSystem` to operate on and the various implementation provide their implementation and select the test sets

that shall be executed. So the access to ZIP and JAR resources can do:

```
public class JarFileSystemTest extends FileSystemFactoryHid {
  public static Test suite() {
     NbTestSuite suite = new NbTestSuite();
     suite.addTestSuite(RepositoryTestHid.class);
     suite.addTestSuite(FileSystemTestHid.class);
     suite.addTestSuite(FileObjectTestHid.class);
     suite.addTestSuite(URLMapperTestHidden.class);
     suite.addTestSuite(URLMapperTestInternalHidden.class);
     suite.addTestSuite(FileUtilTestHidden.class);
     return new JarFileSystemTest(suite);
  }
  protected void destroyFileSystem (String testName) throws IOException {}
  protected FileSystem[] createFileSystem (String testName, String[] resources)
              throws IOException{
    return new JarFileSystem (createJarFile (test, resources));
  }
}
```

Other types of file system plugins do similiar things, just create different instance of the filesystem.

As usually this helps to fight with an application amoeba. Morever **Test Compatibility Kit** helps to improve regular activities a software engineering organization needs to solve. For example it simplifies the lifecycle of a bug report, it makes it much easier to find out which part of the system is buggy and lowers the number of reassignment that a bug needs to find it right owner. We really lowered the number of assigning to you for evaluation bug transfers by introducing **TCK**s. In some sence a **TCK** supports the best incarnation of programmer's arrogance — until you have an implementation, also use the **TCK** or your bug reports will not be taken seriously.

Deadlock Test

Fighting with deadlocks is a sad destiny of any multithreaded application. The problem field has been under extensive research because it causes huge problems for every writer of an operating system. Most of the applications are not as complex as operating systems, but as soon as you allow a foreign code to run in your application, you basically have to fight with the same set of problems.

In spite of the huge research efforts, there was no simple answer solution found. We know that there are four necessary and also sufficient conditions for a deadlock to be created:

1. **Mutual exclusion condition** — there has to be a resource (lock, execution queue, etc.) that can be owned by just one thread

2. **Non-preemptive scheduling condition** — it is not possible to take away or release a resource already assigned by anyone else then its owner
3. **Hold and wait condition** — a thread can wait for a resource indefinitely and can hold it indefinitely
4. **Resources can be aquired incrementally** — one can ask for new resource (lock, execution queue), while already holding another one

But we do not know how the code that prevents at least one condition to appear shall look like and definitely we do not know how to do a static analyse over a source code to check whether a deadlock can or cannot appear.

The basic and in fact very promising advice for a programmer in a language with threads and locks like Java has is to *do not hold any lock* while calling to foreign code. By following this rule one eliminates the *fourth* condition and as all four must be satisfiable, to create a deadlock, we may believe we found the ultimate solution to deadlocks. But in fact, it is sometimes very hard to satisfy such restriction. Can following code deadlock?

```
private HashSet allCreated = new HashSet ();

public synchronized JLabel createLabel () {
  JLabel l = new JLabel ();
  allCreated.add (l);
  return l;
}
```

It feels safe as the only real call is to HashSet.add and it is not synchronized. But in fact there is a lot of room for failures. First problem is that JLabel extends JComponent and somewhere in its constructor one aquires awt tree lock (JComponent.getTreeLock()). And if someone writes a component that overrides:

```
public Dimension getPreferredSize () {
  JLabel sampleLabel = createLabel ();
  return sampleLabel.getPreferredSize ();
}
```

we are in danger of deadlock as the `getPreferredSize` is often called when a component is painted and while the awt tree lock is held. So even we tried really hard to not call foreign code, we did it. The second and even less visible problem is the implementation of HashSet. It uses `Object.hashCode()` and `Object.equals` which again can call virtually anywhere (any object can override them) and if the implementation acquires another lock, we can get into similar, but even less expected, problem.

Talking about possible solutions for deadlocks would provide enough materials for a its own article, so let us return back to topic of this one — writing tests.

In Java, the solution to deadlocks are often easy. Whenever the application freezes, the user can produce thread dump and from that we can get the description of the problem. From there there is just a step to fix, just lock on another lock, or use `SwingUtilities.invokeLater` to reschedule the code in question from the dangerous section sometime later. We used this style for few years and the result is that our code started to be unpredicatable and we have not really fixed much of the deadlocks as when we modified the code to fix one, we often created new one. My favourite example are changes made in our of our class on Jun 26, 2000 and Feb 2, 2004. Both tried to fix a deadlock and the second one effectively returned the state back, prior the first integration. That means we have successfully shifted the amoeba shape of our application code to fix a deadlock in the year 2000, and four years later we just shifted in once more to improve in one part, but regress with respect to the 2000's fix. This would have never happened if together with the first fix, we also integrated a test!

A test for a deadlock!? Yes, a test for a deadlock. However surprising that may sound, it is possible and often not that hard (we often write a test for deadlock in about two hours, we never needed more than a day). Beyond the automated nature of such test, it also gives the developer confidence that he really fixed something, which is not strong contract to the esoteric nature of deadlock fixes when they cannot be reproduced by anyone. Also, when there is a test, one can choose simpler solution that fixes the problem, then to invent intelectually elegant, but in fact complicated one. The result is that the *art of deadlock fixing* turns into regular engineering work. And we all want our applications to be developed by engineers, are we not?

Writing test for a deadlock is not that hard. In our imaginary situation with `createLabel` we could do that by writing a component, overriding `getPreferredSize`, stopping the thread and waiting while another one locks the resources in oposite way:

```
public class CreateLabelTest extends TestCase {

  public void testSimulateTheDeadlock () {
    MyComponent c = new MyComponent ();
    c.validate ();
  }

  private static class MyComponent extends JComponent
  implements Runnable {

    public synchronized Dimension getPreferredSize () {
      JLabel sampleLabel = createLabel ();

      new Thread (this).start ();
```

```
      wait (1000);

      assertNotNull ("Also can create label", createLabel ());

      return sampleLabel.getPreferredSize ();

    }

    public void run () {
      assertNotNull ("We can create the label", createLabel ());

      synchronized (this) {
        notifyAll ();
      }
    }
}
```

The test works with two threads, one create a component and validates it, which results in a callback to `getPreferredSize` under awt tree lock, at this moment we start other thread and wait a while for it to acquire the `createLabel` lock. Under current implementation this blocks in the JLabel constructor and as soon as our thread continues (after 1000 ms) we create the deadlock. There can be a lot of fixes, but the simplest one is very likely to synchronize on the same lock as JLabel contructor does:

```
public JLabel createLabel () {
  synchronized (JLabel.getTreeLock ()) {
    JLabel l = new JLabel ();
    allCreated.add (l);
    return l;
  }
}
```

The fix is simple, much simpler than the test, but without the test, we would not fix the shape of our amoeba. So the time spend writing the test is likely to pay back.

Often the test can be written by using already existing api, like in our case the `getPreferredSize` method (for example our test. Only in special situations one needs to introduce special method that helps the test to simulate the problem (we used that in our howToReproduceDeadlock40766(boolean) called from PositionRef.java. Anyway deadlock tests are pure *regression tests* — one writes them when a bug is reported, nobody is going to write them in advance. At the begining it is much wiser to invest in good design, but as we explained sooner, as there is no really universal theory how to prevent deadlocks, one should know what he wants to do when a deadlock appears, for that we suggest that test is the best way with respect to our amoeba shape.

Testing Race Conditions

While certain problems with multiple threads and their synchronization are hard to anticipate, as deadlocks mentioned earlier, sometimes it is possible and useful to write a test to verify that various problems with parael execution are correctly handled.

We have faced such problem when we were asked to write a startup lock for NetBeans. The goal was to solve situation when user starts NetBeans IDE for the second time and warn him that another instance of the program is already running and exit. This is similiar to behaviour of Mozilla or Open Office. We decided to allocate a socket server and create a file in a well known location with the port number written to it. Then each newly started NetBeans IDE could verify whether a previously running instance is active or not (by reading the port number and trying to communicate with it).

The major problem we had to optimize for was to solve situation when the user starts more NetBeans IDEs at once. This can happen by more clicks on the icon on desktop or by draging and dropping more files on the desktop icon of our application. Then more processes are started and they start to compete for the file and its content. The sequence of one process looks like this:

```
if (lockFile.exists ()) {
  // read the port number and connect to it
  if (alive) {
     // exit
     return;
  }
}
// otherwise try to create the file yourself
lockFile.createNewFile();
DataOutputStream os = new DataOutputStream(new FileOutputStream(lockFile));
SocketServer server = new SocketServer();
int p = server.getLocalPort();
os.writeInt(p);
```

but it can be at any time interrupted by the system and instead of executing all of this as atomic operation, the control can be passed to the competing process which does the same actions. What happens when one process creates the file, and other tries to read it meanwhile, before a port number is written to it? What if there is a file left from previous (killed) execution? What happens when a test for file existence fails, but when trying to create it the file already exists?

All these questions have to be asked when one wants to have really good confidence in the application code. In order to get the confidence we wanted we inserted a lot of check points into our implementation of locking so the code became a modified version of the previous snippet:

```
enterState(10, block);
if (lockFile.exists ()) {
  enterState(11, block);
  // read the port number and connect to it
  if (alive) {
     // exit
     return;
  }
}
// otherwise try to create the file yourself
enterState(20, block);
lockFile.createNewFile();
DataOutputStream os = new DataOutputStream(new FileOutputStream(lockFile));
SocketServer server = new SocketServer();
enterState(21, block);
int p = server.getLocalPort();
enterState(22, block);
os.writeInt(p);
enterState(23, block);
```

where the `enterState` method does nothing in real production environment, but in test it can be instruct to block in specific check point. So we can write test when we start two threads and instruct one of them to stop at 22 and then let the second one to run and observe how it handles the case when a file already exists, but the port is not yet written in.

This approach worked pretty well and inspite how skeptical opinions we heared when we tried to solve this problem, we got about 90 % of behaviour right before we integrated the first version. Yes, there was still more work to do and bugs to be fixed, but because we had really good automated tests for the behaviour we really implemented, our amoeba edge was well stiffed and we had enough confidence that we can fix all outstanding problems.

Analyzing Random Failures

Those 10 % of random failures mentioned in the previous part emerged, as usually, into more work than just next 10 % of additional tests and few fixes. They inspired this whole new part, as dealing with failures that happen just from time to time and usually on a computer that you do not own, requires more sophisticated technique to be used for their tracking.

The problem with parael execution is that there is really not much help anyone can get if he wants to use it correctly. The metodology is either weak or missing or just too concentrated on specific case, the debuggers are not really ready to push the debugged applications to their parael limits, so in order to really move somewhere, people resort to the oldest solution — to `println` and logging. The old approach is pretty simple — add log messages into your code, run it few

times, wait until it starts to misbehave and then try to figure out from the log file what went wrong and fix it. In case of automated tests a similar approach can be used. Enhance the application code and also the tests with logging, and if the test fails, output all the collected log messages as part of the failure report.

We have achieved this by writing our own implementation of `ErrorManager` (which is a NetBeans class used for logging and error reporting), but one can do this in any test by using `java.util.logging` and implementing its `Handler`. The implementation has to be registered at the begining of the test and has to capture all logged messages and in case of failure make them part of the failure message:

```
public class MyTest extends NbTestCase {
  static {
    System.setProperty ("org.openide.util.Lookup", "MyTest$Lkp");
  }
  public MyTest (String name) {
    super (name);
  }
  protected void runTest () {
    ErrManager.messages.clear ();
    try {
      super.runTest ();
    } catch (AssertionFailedError err) {
      throw new AssertionFailedError (err.getMessage() + " Log:\n" +
              ErrManager.messages);
    }
  }

  public void testYourTest() throws Exception {
     // invoke some code
     ErrorManager.getDefault().log ("Do some logging");
     // another code
     ErrorManager.getDefault().log ("Yet another logging");
  }

  public static final class Lkp extends org.openide.util.lookup.AbstractLookup {
    private Lkp (org.openide.util.lookup.InstanceContent ic) {
      super (ic);
      ic.add (new MyErr ());
    }
  }

  private static final class ErrManager extends org.openide.ErrorManager {
    public static final StringBuffer messages = new StringBuffer ();

    public void log (int severity, String s) {
      messages.append (s);
      messages.append ('\n');
    }
    public void notify (int severity, Throwable t) {
```

```
      messages.append (t.getMessage ());
      messages.append ('\n');
    }
  }
}
```

The logging can be done by the test to mark important sections of its progress, but the main advantage that your code shall be full of `ErrorManager.log` or (if you use the standard Java logging) `java.util.logging.Logger.log`. The test then collects messages from all places and in case of failure provided complete and detailed (well, depending on how often and useful the log messages are delivered) description of the failure which then can be analyzed and either fixed or the logging made more detailed to help to track the problem down (as we did for example in ChildrenKeysIssue30907Test.java).

Sometimes people are reluctant to analyze random failures in tests as something that does not affect the production code. In fact, it may turn out that the problem is in the test and the code is ok, but relying on this is usually false hope. Without deeper understanding of the problem, it can be in the application code and even it is not reproducible all the time, if it occurs it can have huge consequences. If we want to have enough trust to the behaviour of our application and make its amoeba shape less amoebic, logging in application code and logging friendly tests turn out to be very useful tool.

Summary

If you got this far, you very likely saw a lot of convincing examples that writing automated tests is a good thing. Tests can be seen as a form of specification that helps to stiffen the Amoeba's edge (at least from one side — what is supposed to work really works). Tests can significantly reduce the cost of maintanance of an application. Tests form a nice and in our opinion necessary add on to other types of specification and shall be used together with documentation, use cases, UI specs, javadocs everytime one is *serious* about the application's future. This all has been pointed out many times, so let us tell you *why not to write tests*.

One of our friends had worked in an optical fibre company and one day his manager told him: boy, our company is going to pay you a trip to Malaysia. Not bad, it is a long flight from Europe, but the manager forgot to tell him that this was the last thing they decided to pay him. He was sent to Malaysia to teach cheaper workers how to do his work and as soon as he returned back he was fired and replaced by them.

By writing automated tests you are contributing to shared knowledge of the project you are working on and in fact you are making yourself *more easily replaceable*. This is good for the project, but as shown on the fibre example, it

can be dangerous for the workers. Is it even useful to have successful product when it is no longer yours? Maybe not, but our NetBeans project experience shows, that without continous improvements to quality the whole project would be obsoleted long time ago. Tests are a must if we want to keep our jobs. That is why most of us write them. And if someone does not? Well, we at least know why. He may be afraid of loosing his job in favor of someone else. But maybe he is just not good enough and has to be afraid...

What is your attitude? Do you care about future of your project or are you just affraid to loose your job? Are you going to write automated tests now?

Using NetBeans as a Framework for a Network Monitoring Application

David Štrupl

E-mail: david@solutions.cz

Abstract

NetBeans project is quite well known for producing open source integrated development environment (IDE). Less known is the fact that the core of the application can be used as an application framework for general desktop application development. It is described what had to be done to adapt the framework to be usable for a project from a different application domain — Nokia network monitoring tool.

Introduction

We will first briefly introduce NetBeans IDE and the developed application. We will try to show the similarities and differences between the two. The next sections will describe how we have adapted the NetBeans framework to fit into our target application requirements. We will conclude with a feasibility assessment and a brief outlook to the future plans.

NetBeans

NetBeans is an open source IDE written in Java having quite long history (when we take the age of Java itself into consideration). It evolved from a students project through commercial IDE to finally land into the open source world. As of the latest release (4.1) the functionality covers a lot of different areas ranging from J2ME client programming (phones), through J2SE desktop programming to J2EE server development.

As the program has many unrelated functions the project quite logically converged to a pluggable modular architecture. I have intentionally used the word

converged since it is a live project that needs to accommodate all architectural changes while producing end user usable products.

The idea to use parts of NetBeans for different types of applications is not new and have been used even before the project started to proudly present itself as the application framework. Today the core NetBeans developers are aware that they provide parts that don't necessarily end up in an IDE type of application. But as there is quite strong commitment to backward compatibility the application framework is slightly more suitable for the type of application similar to the original (and still the most important) one.

NetAct

NetAct^{TM} is a network and service management software developed by Nokia. It can manage both the network and services in a centralized manner, meaning that the operator can view all network element failures, service quality indicators and traffic from one single screen. The system allows alarm handling, reporting capabilities as well as network optimization and configuration. The domain of the program is commonly abbreviated as OSS (Operations Support Systems). It is a system for managing GSM/GPRS/EDGE and WCDMA networks.

While the whole NetAct comprises lot of different parts where probably the most important functionality is performed on the servers the most visible part is the GUI client application running in the service control rooms of the mobile operator. And it is this client application where the NetBeans framework is being used.

Comparison

We will try to compare the two applications in this section. The main similarities and differences will be described together with possible consequences for the application developer.

Similarities

Let's first check what do our application have in common.

GUI Integration

As both systems are large programs they need to accommodate lots of different tools that have to work together. Let's look at an example from each domain

showing that both programs need to contain complicated user interface elements that work together to achieve the user task.

IDE: The developer starts by writing some code. He needs to instantly see the documentation for the used libraries. While writing the code the structure of the source code is instantly updated and visualized. After the code is written the compiler and deployment tools are used. The user then needs to debug the running program while still having access to the previously mentioned information. The debugging session presents yet another stream of fresh and constantly changing data.

NetAct: The operator receives an alarm from a network element in alarm list. There are lot of types of alarms so he needs to read alarm manual for instructions on how to handle alarm. Then it is needed to run configuration performance reports for the network element and for example telnet into the network element with the given instructions. If it is not possible to resolve the problem from the control room a trouble ticket is created for a field engineer to physically visit the element.

As can be seen from the examples both programs have large scale user interface — lot of interacting views and tools are presented to one user. In both cases the GUI front end must be ready to show all the information in some clever way allowing the user to concentrate on the task at hand but not loosing the overall information about the status of the whole system.

Complex interaction

Both the IDE and the network monitoring application needs rather complex user actions in a very dynamically changing environment. The work has to be done on the user's machine in order to present the dynamic data with a reasonable feedback.

The central feature of the IDE is the source code editor. While ASCII text editors are very common in all operating systems the editor used by the developers has usually lot of additional features. Besides basic editing it provides code highlighting for different types of languages (Java, XML,...), code completion, code folding, error highlighting and loot of other extensions. While the user works mainly in the editor he has several helper views visualizing different aspects of both his source code and the computing environment (running servers, available services etc.).

When working with the NetAct application the main user interface elements are the Network views. They are graphical display of network on map or hierarchically organized views of network elements. The helping views are indicators for alarm states, trouble tickets, notes, properties of objects. Similarly to the IDE the views are rather dynamic since they are constantly updating according to the data received from the servers.

In both applications we need the GUI elements to not only be dynamic in their nature but also to work reasonably well with each other. The requirements are really similar for both applications: while the views are coming from different parts of the application they have to understand common operations: being able to accept drag and drop transfer, have similar popup menu behavior, be able to enable/disable toolbar buttons.

Customization

Both NetBeans IDE and the NetAct face similar requirement: being able to build and/or deploy slightly different versions of the products based on the user preference.

The IDE for a mobile phone application developer will look slightly different that the IDE used by the J2EE developer. NetBeans solve this requirement mainly by being able to download and run add-on modules. These modules will provide the necessary features that might not be present in the packaged product. As the architecture is pluggable it is also not a big problem to package a product based on the basic IDE with additional modules. The add-on modules are able not only to add features but also to provide customization of the look — it is possible to provide different set of icons, texts and colors making a branded product.

In NetAct the customers have lots of in-house tools that want to integrated with the GUI of the main NetAct client. Not only the customers but also service people from Nokia are able to add additional functionality tailored for different customers. This is in a way very similar to the branding requirement described above for the IDE.

Differences

Although there are some similarities there are also very notable differences between the two applications.

Overall architecture

The IDE is an application that is installed on the users machine and that is able to operate in isolation. While there is a support for different kind of client functions for several types of servers (version system client, J2EE client, database client, web services client) the main application executes on the user machine without any need of additional services.

NetAct is rather considered to be a “Rich client” of some server functionality. The servers are J2EE application server clusters plus database server and

communications server in the backend. As mentioned above while the client is complicated GUI program the data that it tries to visualize are stored on remote servers. Without the server side the NetAct client is useless.

Installation

Another big difference is the way the user gets to a state where he can start using the application. In the case of the IDE the usual scenario is this: the user either downloads or purchase a CD and installs the product. After the product is installed he can either use it or use the Auto update functionality to pull newer versions of parts of the system. All in all the user is responsible for installing the pieces of the software.

With NetAct the Java Web Start is used to bring current version of the client to the users machines. All the updates are automatically downloaded and installed. The system administrators decide for the users what will be installed and how. This is absolutely necessary since the client stubs for EJBs must match the server-side version. And as mentioned earlier the application is "just" a client to the server components. Also the time when the parts of the systems need update are controlled from the server because the server parts will evolve and the client must keep pace with them (unlike in the IDE where it does not really matter what version of the IDE the user is using as long as the final product is deployable on the appropriate version of the target runtime).

Robustness and Security

As the IDE is used to make a software developer productive it is mainly his responsibility to install and maintain it. There are no special requirements for security — the program runs with all the permissions granted by the operating system to the current user. Also the robustness requirements are modest — in the case of the IDE malfunction it usually affects one user (there can be exceptions to this rule).

In NetAct both robustness and security must play bigger role when designing the system. In case of malfunction the immediate effect can for example lead to turning off the phone network for a region. This also brings bigger concern for security: the system must be able to grant permission to use various functions to sets of users.

Users

There is also some difference in the users of our applications. The users of the IDE are (usually) people competent in using the computers. After all if you

want to make a program for the computer you should have seen some programs before.

In NetAct the average computer competence can vary. The main users are network engineers and shift workers watching the status of the mobile network and raising alerts if something does not run as usually.

Changes

The changes to the NetBeans platform were needed mainly to overcome the described applications differences.

JNLP — Java Web start

The NetBeans module system heavily relies on the installation structure of the application. The installation procedure determines all the files location. The directory structure remains fixed after installation.

In NetAct Java Network Launching Protocol (JNLP) is used to bring the parts of the application to the user and start it. JNLP is designed to make it simple for the user to automatically retrieve a new version of each file when there is one on the server. While JNLP makes it easy for the user to always have current version of all the files, the files themselves are stored and managed by the JNLP client software (Java Web Start). The NetBeans module system has to be adapted not to rely on the installation information.

There was one nice feature of the NetBeans module system that was not supported by the Java Web Start. In NetBeans the module system creates classloaders allowing code isolation of the modules — the result is that different modules can use different versions of the same library (two versions of the same class can be used by one virtual machine at the same time). This feature collided with the system used by the default JNLP client (Java Web Start). Java Web Start puts all resources on the application classpath — effectively turns off the nice isolation feature of the NetBeans module system.

Fortunately enough it was possible to hack the classloaders in NetBeans to achieve the code isolation feature even when started from Java Web Start. Of course such feature is useless for the IDE user since the installation and update structure of both application differs.

GUI

As mentioned in the section about similarities the GUI elements used in both applications are similar. It is no surprise that a lot of NetBeans GUI code could

be directly reused in the NetAct client.

Look and Feel

As all today's applications strive to look originally both of our programs are no exception. While NetBeans tries to be Look and Fell neutral the NetAct has chosen to use its own Look and Feel. But as NetBeans introduces some new GUI elements we had to make sure we are able to render them in the custom Look and Feel.

To achieve good results we had to do small tweaks in both NetBeans and the custom look and feel.

Web-style navigation

I would like to mention one features of the GUI that is not present in the NetBeans IDE but that was found useful in the NetAct context. Almost every user of todays computer uses a web browser and bookmarks are part of each of them.

The user is able to create and manage bookmarks that allow him to start a tool with a certain state. State in this sense includes the content displayed by the tool and the layout (e. g. width of table columns). It is important to understand that bookmarks store only criteria for displaying content, not the content itself.

Besides the user defined bookmarks we have implemented also Forward/Backward Navigation. Whenever a 'bookmarkable' action takes place in a tool (i. e. a major content change – no undo/redo functionality), a bookmark is generated automatically and added to a bookmark stack that can be used for navigating forward/backward. The difference between user created bookmarks and navigation bookmarks is that the latter do not necessarily store the full state of the tool.

Summary

Although NetBeans is coming from different application domain it is beneficial to use it as an application framework when writing network monitoring tool. The effort needed to overcome the differences is significantly lower than would be the effort required to write the framework from scratch.

Architecture of a Business Framework for the .NET Platform and Open Source Environments

Thomas Seidmann

E-mail: Thomas.Seidmann@cdot.ch

Abstract

This paper contains a description of the architecture and components of a software framework for building enterprise style applications based on the .NET platform. The process and achieved results of porting this business framework to an open source platform based on GNU/Linux/Mono is described as well.

1 Introduction

Many software companies focused on building potentially large distributed data-driven applications are faced similar problems during the life cycle of their projects: the technology they use as a base for their solutions does not provide all the necessary abstractions or their abstraction level is too low. Usually they end up with a framework, in many cases even more of them in various projects, often depending on the particular project members involved in them. This fact leads to software maintenance problems as well as internal incompatibilities. To fill this gap we've developed a business framework (in further text referred to as 'framework') for so called enterprise application as a part of a broader product called Cdot-InSource, which comprises in addition to the business framework also services like education, project analysis, design and jump starting the implementation and consulting. The business framework part of Cdot-InSource is based on the .NET Framework platform authored by Microsoft and can be used as a foundation for a wide variety of applications, which might (but don't necessarily have to) employ Web services.

The outline of the rest of this paper is as follows. In section 2 we'll give an overview of the overall architecture of the framework contained in Cdot-InSource. Section 3 contains the description of the building blocks contained

in the framework. Section 4 explains the blueprints for building and testing real-life applications using the framework. In section 5 we'll explain extending possibilities of the framework with respect to platform changes and additional requirements. Finally, in section 6 we'll described the objectives and effort of porting the framework to an open source platform as well as the current state and lessons learned so far. We conclude the paper with some thoughts of working in a "virtual company" and its social consequences.

2 Architecture of the Business Framework

The architecture of the framework can best be explained based on applications it is intended for. We are looking at typical multi-tier applications with a relational DBMS used as a data store, a hierarchy of business services built on stateless components, a web service tier used as an entry point to the service layer and a set of client (front-end) applications. Figure 1 illustrates this architecture and also gives some hints about the platform support and contains some forward references to the next section.

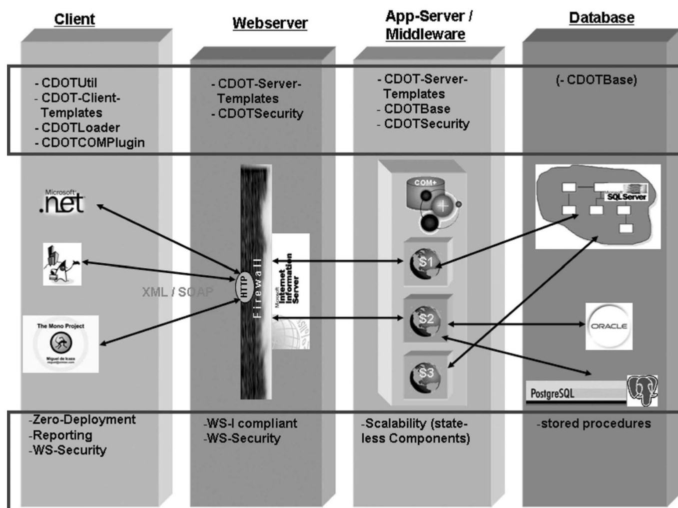


Figure 1 Cdot-InSource architecture

The **web service tier** exhibits high adherence to current standards as formulated by the WS-I consortium. For example instead of inventing own authentication mechanisms, WS-Security is used for this purpose. When using Microsoft's ASP.NET implementation, this tier is normally hosted in the Internet Information Server (IIS), although other alternatives exist as well (see section 6). In addition to authentication, web services are usually responsible

for authorization, although this task may be also delegated to the business services. It is of crucial importance that no business logic is implemented in the web service code itself, but delegated to the business services instead. The web service tier may form a hierarchy of layers (by aggregating simpler services into complex ones), but normally this won't be the case.

The **business service tier** can and usually will be structured into some hierarchy of layers with data access components (normally using DBMS-native data providers and protocols for accessing the data store) at the bottom and components performing complex business logic at the top. As already mentioned before, components of this tier are supposed to be stateless in order to achieve scalability of the whole system. There may be exceptions to this rule, for example in case of some special singleton objects, but generally state should be kept either in the database or in the client tier. To allow for higher scalability and/or integration with external systems the business service tier may (even partially) utilize middleware services like COM+ for functionality like distributed transactions or message queuing. Hosting of the business service components can be accomplished in a variety of ways starting from in-process with respect to the web service code, through an operating system service up to hosting in the middleware services. The same goes for the communication between these two layers, which can be done with .NET remoting or middleware-provided protocols (for example DCOM).

The primary **client tier** platform is, as in the case of the previous two tiers, the .NET framework, thus effectively forming a thin .NET client (also known as Windows Forms client). .NET offers advanced techniques for distributing portion of client code over a network, which can be used for zero (sometimes called 'no touch') deployment of front-end programs. As shall be seen in section 6, efforts are being undertaken for enabling platforms other than .NET to be able to execute .NET managed code.

Passing data between the various tiers is based on the notion of ADO.NET datasets, which represent an in-memory database with excellent serialization mechanisms for communication as well, even via web services. Thus instead of inventing own value object types, datasets (both strongly typed and untyped) are used for this purpose. The support for Java clients, although depicted on figure 1, is currently limited, due to the lack of dataset functionality. This problem of toolkit compatibility is tackled as well.

3 Components of the Framework

Flowing from the previous section, all program code contained in the business framework consists of MSIL, so called managed code. The basic components of the framework in the context of Cdot-InSource are depicted on figure 2, forming

the central part of the picture. These components can be described as follows:

- `Cdot.Security` contains code augmenting the WS-Security implementation as well as an authorization sub-framework for performing RBAC (Role Based Access Control) independently from the one contained in .NET. This component is used by the client, web service and business service tiers.
- `Cdot.Base` represents an abstraction layer above ADO.NET for database access offering data provider independence to some extent, thus making the task of porting to a different DBMS platform easier. This component should be used only by program code of the business service tier.
- `Cdot.Office` contains glue code and abstraction shims for interaction with the operating system shell (Windows, X11) and some office productivity software systems (MS Office, OpenOffice). Both the use by the client tier as well as the business services is thinkable.
- `Cdot.Util` is the biggest component in terms of class count and is currently dedicated to .NET (Windows Forms) clients. This reaches from zero deployment support code through specialized user input widgets (controls) to a user interface configuration and internationalization sub-framework.

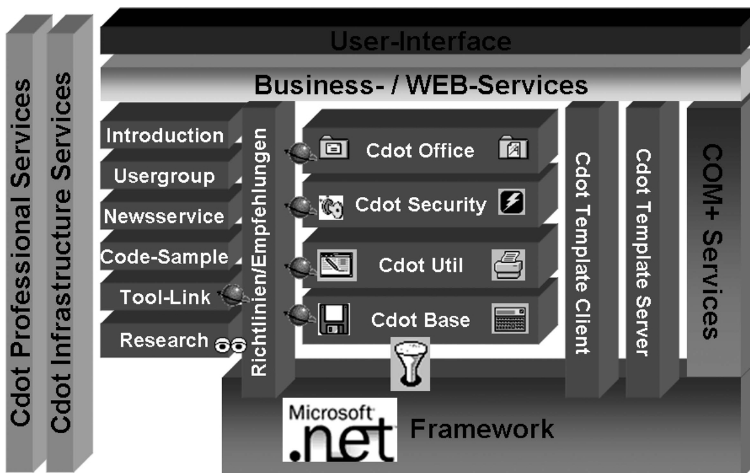


Figure 2 Cdot-InSource components

Not shown on the figure are various third party components the framework relies in addition to the .NET platform. A complete enumeration of such components is clearly outside of the scope of this paper. Let us just mention on noticeable component: a grid library. Data driven enterprise applications often need a way to represent data in a tabular form using some kind of a data

grid. Cdot.Util contains a shim layer offering virtually the same programming interface on top of various underlying grid implementations, which simplifies their replacement in client code. One of those grid implementations (actually the preferred one) resides in a third party component library. Another example of a third party component is a report generator, which we integrated into the framework instead of writing our own one.

4 Building and Testing Enterprise Applications based on the Framework

In the previous section, figure 2 contains an important architectural component in the right central part of the picture – templates for service- (server-) and client-side code which serve as a blueprint for building applications. Although not covering every last detail of the framework, these templates represent a major help in starting a new project from the ground. Besides showing the way to structure the code in the various layers of an application they also stress out the importance of including testing code from the very beginning of the coding phase based on the principle of unit testing. A third-party (open source) unit testing tool called NUnit [1] is used for this purpose. Including testing code is deemed important particularly in the service component code and the templates show the way of accomplishing this task.

Unit testing is employed also inside the framework itself for the purpose of regression tests. Next section contains some notes on this.

In addition to templates and (simple) code samples for the various components of the business framework a complex sample is available, which should serve as a demonstration of the use of the whole framework. This sample contains the implementation of all tiers as described in section 2.

5 Extensibility of the Framework

The business framework of Cdot-InSource shouldn't be viewed as a static piece of code. Instead, it is a dynamical system evolving in time; additions and modifications are performed on a daily basis. Of course, since users of the framework are depending on the stability of their platform, some stringent rules have to be followed by the framework developers: incompatible changes should be avoided as far as possible and regression tests should be run after every significant change of the framework code to insure that previous functionality is still guaranteed. New functionality must come hand in hand with corresponding new testing code.

[1] http://www.nunit.org

6 Porting the Framework to the MONO Platform

While .NET is widely accepted as an application platform, there are exceptions to this. The call for an alternative to platforms authored by Microsoft can sometimes and someplace be clearly heard. Until these days the only possible usable answer was the switch to the Java platform. The state of the MONO project [2] changes the view of things slightly. This project is aimed at offering an open source alternative (binary compatible) to .NET, developed mainly on top of GNU/Linux, running on a wide range of operating system platforms including even Windows.

The promising state of the MONO project lead directly to the idea of using the very same business framework on this platform in addition to .NET. We will denote the adaptation process as porting even though the aim is to use exactly the same MSIL files (assebmlies) on both platforms.

The porting process comprises following problems to be solved:

1. Enabling the Web service and application server components to execute on MONO.

2. Enabling the client components, which normally are Windows Forms application components, to be able to execute in the Mono environment.

The first item turned out to be pretty straightforward with MONO version 1.0. We have managed to get the web service and business service tiers of an application to run on GNU/Linux, utilizing the Apache web server for the former. The business service tier in this particular application was originally hosted in a Windows service, so we've adapted it to execute as a background process (daemon) employing .NET remoting for the communication with the web server. The only real problem turned out to be the WS-Security implementation contained within MONO, which we had to modify and augment with missing functionality.

The second item is much more complicated and tricky, since the state of Windows Forms emulation in MONO by means of the Windows emulator Wine [3] is very immature and also due to the fact, that there is a number of third party components used with Cdot-InSource which must be taken as-is without the possibility to adapt them to MONO. This item is subject to work-in-progress right now.

[2] http://www.go-mono.com
[3] http://www.winehq.com

7 Conclusions and Future Work

The Cdot-InSource business framework is evolving in time, but the platform is evolving and changing as well. Already today the extent of the next major overhaul of the .NET platform in Windows Longhorn can be glimpsed. It is the job of a good business framework to cope with such platform changes and offer the access to technologies as WinFX or Indigo with little or no impact on client code to the framework.

An interesting aspect of the business framework apart of its actual functionality and contents is the relationship to customers. It is not meant as a shrink-wrapped off-the shelf product, but instead it is based on a partnership model with customers. That way, with a relatively small number of customers (a maximum of several tens), it is possible to react on their needs and let flow the fulfillment of theses needs into the framework. It seems, that this customer relationship model is quite attractive and appreciated, since we've encountered positive feedback on this in many places.

Rozdílné přístupy komerčních organizací a rozpočtových organizací k zavedení informačního systému

Jan Rychlík

E-mail: rychlik@civ.zcu.cz

Klíčová slova: implementace IS, životní cyklus IS, definování cílů, způsob financování IS, úloha managementu, uživatelé IS

1 Úvod

Zavádění informačních systémů (IS) je poměrně složitou činností, projektem. Dnes už jsou známy bohaté zkušenosti z této oblasti nejen ze světa, ale i z domácího prostředí, takže už snad každý, kdo se do dobrodružství zavést nový IS pustí, si dobře uvědomuje důležitost organizačního zajištění celého projektu. Také klíčoví pracovníci firem, kteří se tvorbou a zaváděním IS profesionálně zabývají, si jsou dobře vědomi rizik, které z podcenění organizačních aspektů vyplývají. V případě neúspěchu IS však nakonec nese největší ztráty organizace, která IS požadovala. Zkušenosti jsou však získány především z oblasti zavádění IS v komerční sféře. V rozpočtových organizacích, ať už se jedná o veřejnou správu na straně jedné či univerzity a školství na straně druhé, však dochází při implementaci IS k určitým odlišnostem. Vzhledem k tomu, že k zavádění IS přistupují dodavatelské firmy již jako k rutinní záležitosti, chtěl bych v předloženém příspěvku upozornit na některé aspekty, o kterých se domnívám, že by jejich přehlédnutí mohlo vést i k neúspěchu IS. Pokud jsou sledovány rozdíly v implementaci IS komerčních a rozpočtových organizacích, je nutné ještě vymezit, za jakých dalších předpokladů bude srovnání provedeno. Tímto předpokladem je, že IS bude dodán dodavatelskou firmou, a to za samozřejmého předpokladu, že firma je plně profesionální, tj. nejen že zná danou problematiku, ale má i vypracovanou vlastní metodiku implementace IS. Odlišnost lze očekávat jen v těch fázích projektu, ve kterých je zodpovědnost na uživateli IS. Příspěvek ukazuje, že se jedná o čtyři oblasti, a to definování cílů projektu, způsobu financování projektu, přístupu managementu, a chování koncových uživatelů.

2 Úspěšné zavedení IS a definování cílů projektu

Předložený příspěvek se nezabývá definicí vymezením pojmu *informační systém*, i když se zabývá pojmem *úspěšné zavedení IS*, a to proto, že vymezení pojmu *úspěšné zavedení IS* již souvisí s metodou zavádění systému.

2.1 Pravidla pro zavádění IS

Pravidla pro zavedení IS se ustálila do etap v terminologii IS označených jako *životní cyklus IS*. Existuje více různých životních cyklů (např. [Kra98, RichSo96]), které se mezi sebou liší zejména ve způsobu zahrnutí zpětných vazeb od používání již hotového či částečně hotového systému k opětovné analýze a návrhu systému, ale pro účely tohoto příspěvku lze uvažovat jen jednoduchou modifikaci zahrnující nejdůležitější fáze:

- specifikace cílů,
- specifikace požadavků,
- analýza a návrh systému,
- kódování,
- testování,
- předání a užívání.

V průběhu etap je ještě nutné udržovat dokumentaci a připravovat data pro test. Pro bližší přiblížení bychom fázi *specifikace cílů* mohli zkráceně označit jako odpověď na otázku *proč* systém zavádět, fáze *specifikace požadavků* odpovídá na otázku *co* musím udělat, aby cíle byly splněny, a fáze *návrh systému* říká *jak* to bude provedeno.

Autor článku se domnívá, že některé fáze má zcela na starosti dodavatel systému, jako *analýza a návrh systému, kódování* a také řízení *testování*. Fáze *specifikace požadavků* je důležitou etapou, ve které se odběratel dohodne s dodavatelem na náplni dodávaného systému, a v teoreticky ideálním případě by byla formulována smlouva a její předmět. Není na škodu, je-li tato fáze alespoň zakončena oponenturou. Co by však mělo být výhradně záležitostí odběratele a tím i uživatele budoucího systému je *definování cílů* a samozřejmě *užívání* systému. I tyto fáze však ovlivňují, a to významně úspěšnost zavedení IS. Úspěšným zavedením IS budeme rozumět systém [Kra98], který

- je samozřejmě dokončen,

- splňuje cíle projektu,
- byl dokončen v čase stanoveném harmonogramem nebo alespoň v rozumně a kontrolovatelně prodloužené době,
- dodržel rozpočet nebo jej překročil v rozumné a kontrolované výši.

2.2 Příčiny neúspěchu IS

Podíváme-li se na podmínky úspěšného zavedení systému, je zřejmé, že zásadní podmínkou je dokončení projektu. O ostatních třech podmínkách lze alespoň diskutovat, ale nedokončení projektu je jasné a nepřehlédnutelné. Přestože výše uvedené postupy, jak dosáhnout úspěšného zavedení IS jsou známé, není nedokončení projektu až tak výjimečným jevem. V [Kra98], kde je problematika příčin neúspěchu i úspěchu zavedení IS poměrně dobře rozebrána, je např. uvedeno, že v roce 1995 vláda USA utratila 81 miliard dolarů za nedokončené projekty, což představovalo 31 % finančních prostředků na SW projekty. Poznamenejme, že uvedené statistiky jsou už poněkud zastaralé, ale i dnes např. Oracle i SAP uvádí ve svých prezentacích sice trochu jiná procenta, avšak příčiny stále zůstávají.

V seznamu příčin, proč k tomu došlo, jsou zajímavé příčiny na prvních místech:

- nejasné definování cílů, nekompletnost a nejasnost požadavků (22 % z neúspěšných projektů),
- nedostatek zájmu a podpory ze strany uživatelů (12 %),
- nedostatek zdrojů, tj. podhodnocený rozpočet a krátké termíny (11 %),
- nerealistická očekávání (10 %).

Srovnáme-li vyjmenované příčiny s fázemi životního cyklu IS, je patrné, že došlo k podcenění právě těch fází, které jsou záležitostí odběratele.

Na druhou stranu jsou zajímavé i statistiky a příčiny úspěšného zavedení projektu. Ve statistice výše uvedeného průzkumu je nejvýše uvedený důvod úspěšnosti IS zainteresovanost uživatelů (18 %) a na druhém místě podpora managementu odběratele (16 %). Statistika uvedená ve [Vra99] dokonce uvádí, že z úspěšně zavedených IS na vysokých školách v ČR se na úspěšnosti podílí 40 procenty podpora managementu. Pro zajímavost, stejná statistika říká, že jakost systému se na úspěšném zavedení podílí jen 20 %.

Na úspěšném zavedení projektu se tedy kvalita dodaného SW podílí jen minimálně. Rozhodující je správné a jasné definování cílů, podpora a aktivní zapojení managementu do řízení projektu a zapojení koncových uživatelů.

2.3 Definice cílů

Náměty a postupy pro definování či formulaci cílů je opět oblast již teoreticky propracovaná, velmi dobré doporučení lze nalézt v [Kra98]. V ideálním případě by měl vzniknout dokument „Stanovení cílů projektu“ (nebo také „Projektový záměr“), který nejen odpovídá na otázku „proč“, ale dotýká se i dalších souvislostí. Podrobnějšími zásadami formulace cílů se v tento příspěvek nezabývat, jen zdůrazněme požadavek, který má dokonalá definice cíle splňovat, a to jeho měřitelnost. Jestliže se např. v definici cíle objeví slovo „zkrátit“ (obvykle časový termín), musí být současně řečeno o kolik nebo na jakou hodnotu.

Formulovat cíle tak, aby byly měřitelné je důležité hned ze dvou důvodů. Jedním důvodem je umožnit vyhodnocení úspěšnosti systému, po spuštění systému lze jasně určit, zda cílů bylo dosaženo či nikoliv. Druhý důvod je ten, že cíle jsou formulovány přesněji, a tak lze mnohem snáze realizovat vlastní projekt, zejména fázi specifikace požadavků a návrh systému. Není nic horšího, než když se ujasní teprve při testech systému, co si pod obecně vymezeným cílem představoval zadavatel a co dodavatel.

3 Rozdíly v chování komerčních organizací a organizacích rozpočtových

3.1 Definování cílů

Cíle IS v komerční sféře jsou výrazně jiné než u organizací rozpočtových. U komerční organizace jde samozřejmě o zvýšení zisku, případně o posílení konkurenceschopnosti. K tomuto může IS napomáhat podporou např. v těchto oblastech[1]:

- zkrátit dobu rozhodování,
- poskytnout ucelená data o podniku a jejich historii,
- podchytit požadavky zákazníků,
- podchytit vývoj trhu, sledování obchodních charakteristik,
- urychlit inovaci výrobku nebo služby,
- zkrátit dobu od nákupu surovin nebo komponent do prodeje finálního výrobku,
- snížit skladové zásoby.

[1] Záměrně jsem neuvedl slovo „cíle“, protože tak, jak jsou formulovány, obecně nesplňují základní požadavek měřitelnosti, i když v konkrétních případech by nebylo obtížné krátkým upřesněním měřitelnosti dosáhnout.

Na cíle týkající se těchto a podobných oblastí jsou profesionální firmy dodávající IS zvyklé a umí se v problematice rutinně orientovat.

Cílem zavádění IS ve veřejné správě určitě není zvýšení zisku či posílení konkurenceschopnosti, protože tam tyto pojmy vůbec nemají smysl[2]. Na takové prostředí nejsou dodavatelské firmy zvyklé, pokud se přímo nespecializují na oblast státní správy a samosprávy. Obvykle se snaží v rámci analýzy prosadit funkce systému, které mají již z komerčních oblastí hotové a které v prostředí veřejné správy nejsou až tak důležité, a to na úkor propracování potřebnějších speciálních funkcí. Předejít tomuto nedorozumění lze jedině jasnou formulací cílů systému. Domnívám se, že formulace cílů by měla být u rozpočtové organizace mnohem pečlivěji vypracovaná než u organizace komerční.

Oblasti, ze kterých by mohly být cíle definovány, mohou být např.:

- zpracování velkého objemu dat,
- uspořádání informací,
- řízený přístup k informacím,
- pojmenování událostí, požadavků, prostředků, peněz,
- zkrácení doby vyřízení požadavku,
- zavedení GIS systémů s návazností na katastr, uložení sítí, sledování dopravní situace apod.

Již výše bylo uvedeno, že cíl musí být definován tak, aby byl měřitelný. Musím upozornit, že vydávat výše uvedené či podobné formulace za definice cíle není úplně vhodné, protože nejsou měřitelné. Dost často užívaná formulace „zefektivnit činnost úřadu" je sice pěkná politická deklarace, ale rozhodně špatně definovaný cíl projektu. K tomu, aby se mohla stát definicí cíle by bylo nutné ještě určit, jak budeme efektivnost úřadu vyhodnocovat, tj. jaké veličiny budeme měřit, jakých hodnot nabývají před zadáním požadavku na pořízení IS a jaké hodnoty by měly mít po jeho zavedení. Ještě mnohem nevhodnější je formulace „optimalizace procesů úřadu", kde nejen že je obvykle špatně chápán pojem proces, ale pojem „optimalizace" vyžaduje kromě měřitelnosti stanovit i účelovou funkci, u které pak hledáme maximum nebo minimum.

3.2 Způsob financování

S definováním cílů souvisí i financování celého projektu. Finanční požadavky zavedení IS se obvykle skládají ze čtyř komponent:

[2]Snad jen „zisk" jako účetní pojem.

- dodávka a instalace systému,
- náklady na činnosti prováděné vlastními silami,
- roční údržba systému,
- následný rozvoj systému.

A na konci těchto úvah musí být položena otázka: „Stojí mi to za to?“ A zde opět vidím rozdíl v chování komerčního subjektu a subjektu, jehož základním zdrojem financí jsou dotace.

Majitel komerčního subjektu velice dobře ví, jak mu jdou obchody a kolik si může dovolit investovat do informačních technologií. K tomu pak hledá vhodný IS a dodavatele. Samozřejmě, že si provede průzkum trhu, např. poptávkou u různých dodavatelů a hledá jakési optimum mezi funkčností a vhodností systému a celkovou cenou. A pokud by cena byla náhodou nižší než předpokládal, je jen příjemně překvapen. U komerčního subjektu začíná životní cyklus IS zcela jasně marketingem s očekáváním návratnosti investice do IS vložené. V komerční organizaci existují peníze pro financování projektu zavedení IS před definováním cílů.

Rozpočtová organizace postupuje obráceně. Nejprve si řekne, co chce, a pak teprve k tomu shání peníze. Dokonce výběrové řízení nesmí zahájit dříve, než má zajištěné finanční krytí. Tento postup poskytuje mnohem větší prostor pro nesoulad mezi cenou produktu a jeho skutečnou potřebou. Pokud je dodavatel IS dobrý obchodník, snaží se samozřejmě prodat to nejlepší, co může nabídnout, a samozřejmě také za patřičnou cenu. A tady je veliká úloha na straně odběratele, aby se rozhodl, jestli to, co dostane, není již v kategorii zbytečného luxusu. Na druhou stranu se však může stát i podhodnocení dodávky IS se všemi důsledky (viz např. v odst. 2.2 uvedený 11% podíl na nedokončených projektech). U rozpočtovaného subjektu začíná životní cyklus IS definováním cílů a rozhodující je, zda investice budou na projekt odněkud přiděleny, a ne, jestli se v budoucnu vrátí.

Souhrnně řečeno, nepodceňovat fázi marketingu ani u rozpočtové organizace, i když návratnost investic změřit nelze, a v žádném případě ji nepřičlenit k analýze, kterou provádí dodavatel systému.

Není výjimkou, že se dodavatelé IS se svými produkty nebo svými řešeními rozpočtovým organizacím přímo vnucují, což je v současnosti ještě umocněno možností čerpat prostředky z evropských fondů. Jinými slovy, nejen že definice cílů provádí dodavatel IS a ne uživatel, ale i úvodní záměr je formulován dodavatelem. V těchto případech bývá návratnost investic ještě diskutabilnější.

A ještě jedna poznámka. Jestliže se subjekt ať komerční či rozpočtový rozhodne pro řešení svého problému nějakým IS, obvykle podle svých schopností a času sepíše do poptávky co požaduje, aby mohl od nabízejících firem dostat

cenu nabídky. Vybere dodavatele, a ten pak dle své metodiky implementace upřesní cíle. Někteří dodavatelé tuto fázi implementace nazývají přímo *cílový koncept*, někteří *operační analýzou*, ale v každém případě je teprve nyní znám plný rozsah prací na projektu a lze naprosto korektně stanovit cenu. To, že se to neděje u komerčního subjektu je jen na základě jeho přání obvykle podchyceného ve smlouvě, ale u rozpočtové organizace to ani nelze. Ta v žádném případě další peníze v tak krátké době nesežene a na druhou stranu se již jednou účelově přidělené peníze nevrací. Výjimkou jsou jen skutečně velké IS nebo ostře sledované IS, kde na definici požadavků a analýzu či přípravu projektu je vyhledán jiný dodavatel než poskytovatel cílového řešení.

3.3 Přístup managementu

Již v odstavci 2.2 byla zmíněna důležitost aktivního zapojení managementu do projektu zavádění IS. U komerční organizace to nemusí být problém, pokud se podle této zásady sestaví realizační tým a pokud se informace o průběhu projektu do managementu dostávají.

Problém nastává v orgánech veřejné správy a obdobně je tomu i na vysokých školách [Vra99]. Doba od vzniku myšlenky zavést nějaký IS do jeho rutinního používání je dlouhá, z principu chování rozpočtových organizací mnohem delší než v komerční sféře. Předpokládejme, že v jednom roce záměr vznikne a požadavek na finance se zařadí do rozpočtu na příští rok. I když jsou finance schváleny, fyzicky se obvykle objeví na účtě o něco později, vypíše se výběrové řízení a v druhé polovině roku se začne s implementací. Pokud se jedná o systém, který bude jen customizován, obvykle se do konce roku s jeho implementací skončí. Pokud však bude vyvíjen, určitě jeho implementace přesáhne do dalšího roku. Charakter činností úřadů veřejné správy (stejně jako ve školství) vykazuje periodičnost s periodou jednoho roku. Celý první rok provozu systému je tedy nutné chápat jako zavádění IS a skutečně v této době dochází ještě k dolaďování systému, zejména ke změnám organizačním[3]. Teprve další rok je systém v rutinním provozu a bude přinášet očekávané výhody tak, jak bylo stanoveno v definici cílů. Velmi důležitou otázkou je, zda volení zastupitelé (či akademičtí funkcionáři na vysokých školách) se přínosu IS ve své funkci ještě vůbec dočkají. Tak, jak bylo ukázáno, během jednoho volebního období to prakticky ani nestihnou, jen ponesou zodpovědnost za všechny potíže a počáteční pesimismus, který implementace IS přináší. Toho si jsou zejména radní ať městských tak krajských orgánů intuitivně velice dobře vědomi, a proto se zaměřují spíše na jiné, a to krátkodobé cíle. Pokud se přesto management organizace do zavádění IS pustí, je velmi důležité zajistit ve vedení projektu kontinuitu.

[3]Pokud by k organizačním změnám nedošlo, pak je třeba se vážně zamyslet nad tím, že systém byl buď špatně navržen, nebo je špatně používán.

3.4 Přístup koncových uživatelů systému

To, jestli zavedený systém bude úspěšný, závisí na způsobu jeho používání, tedy na uživatelích. Koncové uživatele (prostřednictvím jejich vybraných zástupců) je nutné zapojit už i do fáze analýzy a návrhu systému. Jednak upřesní konkrétní činnosti a v podstatě na základě jejich výpovědí lze popsat současný stav. Stejně tak důležité je ale i to, že už si pomalu začínají zvykat na připravovaný systém, který obvykle přinese i jinou organizaci práce. Samozřejmě, nesmí se zase jejich popisy pracovních procesů přeceňovat, protože je nemohou vidět v celé šíři. Chtěl bych v této souvislosti upozornit i na nebezpečí, které může vzniknout tím, že za danou oblast je v pracovním týmu jen jeden vzorový uživatel. Nikdy není na škodu v době analýzy zajít na jednotlivá pracoviště a pohovořit se všemi potenciálními uživateli.

Ve fázi analýzy rozdíl v chování mezi uživateli IS komerční sféry a veřejné správy není. V čem se ale dle mého názoru liší, je v běžném používání systému. Uživatelé IS v komerční sféře používají systém jako rutinu, snáze akceptují nějaké nedostatky v podobě pracnosti a svěřené prostředky (tj. PC s nainstalovanými aplikacemi) používají výhradně pro určené účely. Uživatelé systémů v úřadech veřejné správy sice také používají IS podle nejlepšího vědomí a svědomí, ale vedle toho projevují větší stupeň jisté zvídavosti. Kdybych měl charakterizovat uživatele IS na svém domácím pracovišti, tj. univerzitě, použil bych i termín „vyšší stupeň neukázněnosti“. Důvod je v tom, že pracovníka v komerční sféře IS přímo „živí“, zatímco v rozpočtové sféře jej „neživí“, tj. jeho plat se nezmění, když IS den, dva nebude funkční. A poznamenejme, že i tento závěr je přímo v souladu s definováním cílů projektu.

4 Závěr

4.1 Závěr pro rozpočtové organizace

Organizace veřejné správy nebo i vysoké školy musí u projektů zavádějících IS dbát nejen na definování cílů zaváděného IS, ale musí si být i vědomi, že profesionální firmy, zavádějící IS jsou z komerční sféry zvyklé na poněkud odlišné podmínky. Právě přesné definování cílů na začátku projektu je nutným předpokladem pro úspěšné zavedení IS, umožní dodržet jak harmonogram tak rozpočet a jsou to záležitosti, které nelze od dodavatelské firmy „koupit“, protože by ji při této příležitosti „prodali kdeco“. To, jestlli je cena dodávaného IS odpovídající lze zjistit jen srovnáním s obdobným IS již někde realizovaným a vyčíslit návratnost nákladů do IS je velice obtížné. Rovněž je nutné počítat s nestabilitou v podpoře managementem organizace související s volbami a s předvolebním obdobím. Také je nutné nějakým způsobem trochu „ukáznit“ koncové uživatele.

4.2 Závěr pro dodavatele IS do rozpočtových organizací

Rozpočtové organizace mají svoje specifické vlastnosti a z toho se odvíjejí nejen relativně zdlouhavá počáteční řízení, ale i způsob financování. To v daleko větší míře neodpovídá skutečně vynaložené práci a je otázkou jednání, zda je vyšší nebo nižší než je předpoklad. Je nutné si uvědomit, že jasné definování cílů je nakonec skutečně záležitostí uživatele a dodavatelská firma by měla do této role budoucího uživatele IS tlačit a ne ji přebírat. Snadno by se mohlo stát, že by se problémy řešily až při testování systému. Jasné vymezení cílů ze strany uživatele IS zabrání zbytečnému vynakládání práce ze strany dodavatelské firmy.

4.3 Závěr pro nezávislé čtenáře

Zavádění IS do rozpočtových organizací naráží na problémy spojené s formulací cílů. Management rozpočtových organizací je volen jen na určité období a má obvykle jiné starosti než zavádět IS. Představu o tom, co by IS měl dělat, mají jen někteří koncoví uživatelé, i když na různých stupních řízení, a bez zásahu managementu není snadné naformulovat cíle společné pro celou organizaci. Na druhou stranu nějaký IS organizace mít musí. Jednak je potřeba mít ekonomický systém (alespoň slušné účetnictví) a jednak i provozní systém. Pro vysoké školy to je různě důmyslná evidence studentů, pro veřejnou správu pak evidence papírů (podpora spisové služby, práce s dokumenty,...). Ze strany dodavatelů dnes již začíná přetlak nabídek, a tak se může snadno stát, že dodaný a implementovaný systém zase už tak očekávaný přínos nemá. Soudného čtenáře musí mimo jiné napadnout i otázka, zda by nebylo vhodné řadu požadovaných administrativních úkolů zjednodušit či dokonce odstanit, než je řešit pomocí IT. Pro úspěšné zavedení IS v rozpočtové organizaci je nutné bdát na součinnost pracovního týmu složeného ze zástupců dodavatele i uživatele a na rozdělení rolí poněkud více než v komerčních organizacích. Pokud se však implementace IS plně nezdaří, zase se až tak moc neděje. Rozpočtová organizace žije dál a dodavatel IS má aspoň referenci.

Seznam použité literatury

[Kra98] Král, J.: *Informační systémy, Specifikace, realizace, provoz.* Science, 1998, ISBN 80-86083-00-4.

[RichSo96] Richta, K., Sochor, J.: *Softwarové inženýrství.* Skripta, ČVUT, Praha 1996.

[Rych04] Rychlík, J.: *Porovnání implementace IS v komerčních ogrganizacích a ve veřejné správě.* Sborník konference IMPA2004, Darovanský Dvůr, MIM Consulting.

[Vra99] Vrana, I., Búřil, J., Černý, A.: *Metody zavádění informačních systémů na univerzitách, Metodická příručka.* Výzkumný úkol RS 98011, Praha 1999.

Trnitá cesta od papírové univerzity k e-univerzitě

Vladimír Rudolf

E-mail: dolf@civ.zcu.cz

1 Předuniverzitní doba kamenná

Doba dinosaurů (myšleno ve výpočetní technice, kde tímto jménem byly nazývány velké sálové počítače) zmizela zároveň s komunistickým režimem. Pamětníci na tuto dobu vzpomínají s nostalgií, rozlišovali se opravdoví programátoři a pojídači koláčů [11], sálové počítače obsluhovaly mladé a hezké operátorky a většina uživatelů byla smířena s výpočetní otočkou několika hodin (což je nutilo programovat mnohem pečlivěji a s rozmyslem).

Založení Západočeské univerzity v Plzni (28. září 2001)[1] budu považovat i za jakýsi mezník ve vývoji IT ve vysokoškolském prostředí v Plzni a dovolím si nazývat dobu od založení Západočeské univerzity až do současnosti novověkem. Tento příspěvek se již dále bude zabývat pouze novověkem.

2 Startup – level zero

O úrovni výpočetní techniky prvního půl roku novověku, tedy konec roku 1991, svědčí úryvek z [12]:

> *V současné době jsou všechny lokální sítě v jednotlivých budovách realizovány na bázi technologie IEEE802.3/Ethernet 10Base2 (nazývaný též „Cheapernet“ nebo „Thin Ethernet“). Vzhledem k nezbytnosti současného používání různých protokolů vyšších vrstev (jako TCP/IP, IPX a DECnet) je na úrovni linkové vrstvy implementován protokol Ethernet v2. Subsítě jednotlivých lokalit ů tedy lokální sítě v jednotlivých budovách – jsou vzájemně propojeny pronajatými pevnými sériovými linkami s přenosovou rychlostí 19.2 kbit/s. Univerzitní síť je přímo propojena do mezinárodní počítačové sítě EUNET pomocí komutované telefonní linky s přenosovou rychlostí 2400 bit/s přes uzel VŠCHT v Praze.*

[1] Do 28. září 2001 jsme existovali pod názvem Vysoká škola strojní a elektrotechnická a Fakulta pedagogická.

Byla to doba kvasu a hledání, kdy nebylo na pořadu dne používat nástroje pro efektivní a optimální řízení univerzity. Bylo mnohem důležitější univerzitu „rozjet“ a za pochodu se učit, jak řídit lidi a procesy. Na trhu bylo minimum hotových softwarových produktů (mám především na mysli software ekonomický a správní pracující v souladu s českým prostředím a legislativou), IT trh žil především přesouváním krabic a prvořadý úkol se zdálo být dohnat hardwarovou ztrátu za vyspělými zeměmi. Hardwarovou ztrátu se podařilo minimalizovat poměrně brzy a stoupající grafy ukazovaly výrazný extenzivní růst. Kvalitativní růst bylo těžké postihnout pomocí grafů a nebylo se moc čím chlubit, pokrok na poli kvality a efektivity využívání výpočetní techniky rozhodně nevykazoval exponenciální charakteristiku.

3 CIV a oblasti pokrytí

Centrum informatizace a výpočetní techniky (během doby se používaly i různé jiné názvy, toto je poslední platný) mělo velikou výhodu v osvíceném vedení univerzity, které v počátcích novověku nezrušilo v revolučním kvasu útvar starající se o výpočetní techniku, ale zachovalo centrální univerzitní charakter oddělení IT a nedopustilo jeho roztříštění do fakultních IT oddělení (výjimku stále tvoří pedagogická fakulta, která si na anachronismy zřejmě potrpí). CIV od počátku zodpovídal za IT především v oblastech:

- ekonomická agenda
- administrace studia
- podpora výuky
- zavádění nových technologií
- komunikační infrastruktura

Jedná se o hlavní směry, kromě toho se staral o podporu uživatelů, řešil spoustu bezpečnostních incidentů, nastavoval zapomenutá hesla, staral se o servis a řešil další každodenní starosti. To však není téma tohoto příspěvku, zde bych rád ukázal vývoj v jednotlivých, výše vyjmenovaných, oblastech.

3.1 Cesta k elektronickému úřadu

Srovnáme-li vývoj komunikační technologie (viz 3.4) s používáním softwarové podpory pro řízení univerzity, zdaleka nenajdeme podobný extenzivní vývoj. Potíže se zaváděním nových systémů do zaběhlé praxe a odpor k novotám již vyjádřil Nicolo Machiavelli a jeho názor je nutné beze zbytku aplikovat i na zavádění informačních systémů:

A nesmíme zapomínat, že nic se nezačíná tak obtížně a nic nepřináší tak pramalou naději na úspěch, jako zavádění nového systému. Tomu, kdo jej zavádí, jsou nepřáteli všichni, kteří se dříve měli dobře. Jeho stoupenci ho hájí opatrně a vlažně, i kodyž si vábudoucnu od něho mnoho slibují. Opatrní jsou proto, že se obávají odpůrců, a nedůvěřiví proto, že je válidské přirozenosti nevěřit novotám, dokud člověka zkušenost nepřesvědčí nezvratnými důkazy o jejich užitečnosti.

N. Machiavelli: Vladař (1513)

Také u nás nebylo zavádění informačních systémů jednoduché. Nelze se domnívat, že administrativní aparát univerzity pracuje v souladu s moderními metodami, které se na univerzitě přednášejí. Aplikovat moderní způsoby řízení do vnitřního chodu vyžaduje osobní oběť, kterou je málokdo ochoten přinést. Podívejme se například na vývoj ekonomického software na ZČU.

...–1993 Škola má základní SW na systémech EC[2]. Mzdová agenda je vlastní software udržovaný na ZČU.

1993–2002 Zakoupen systém UNICOS (pouze mzdová agenda).

1994–1995 Jemně se laškuje se systémem BBM Písek, nakonec se rozcházíme pro nepřijatelné finanční požadavky[3].

1996–1999 Pokus zavést systém firmy CCA Plzeň. Po třech letech hledání chyb a požadovaných úprav CCA nabízí plný outsourcing vedení ekonomiky (včetně personálu) nebo konec spolupráce. Vybrán konec spolupráce. Klasický případ neochoty ekonomických útvarů zavést nový systém.

2000–... Vybrán systém Magion ze Vsetína. Začíná éra interaktivního přístupu k ekonomickým informacím nejen pro účetní, ale i pro osoby zodpovědné za jednotlivé útvary a katedry. Postupný převod všech ekonomických agend pod tento systém (postaven nad databázovým systémem Oracle).

Předchozí výčet není radostný. Ještě v roce 2000 si vzpomínám, jak v předloženém návrhu rozpočtu v Excelu jsme zjistili součtovou chybu. V duchu jsme se škodolibě zaradovali, že velký Bill má chybu ve svém všeobjímajícím softwaru, ale realita byla prozaičtější. Rozpočet byl spočten na klasickém papíře a výsledky byly vepsány do Excelovských tabulek včetně součtů.

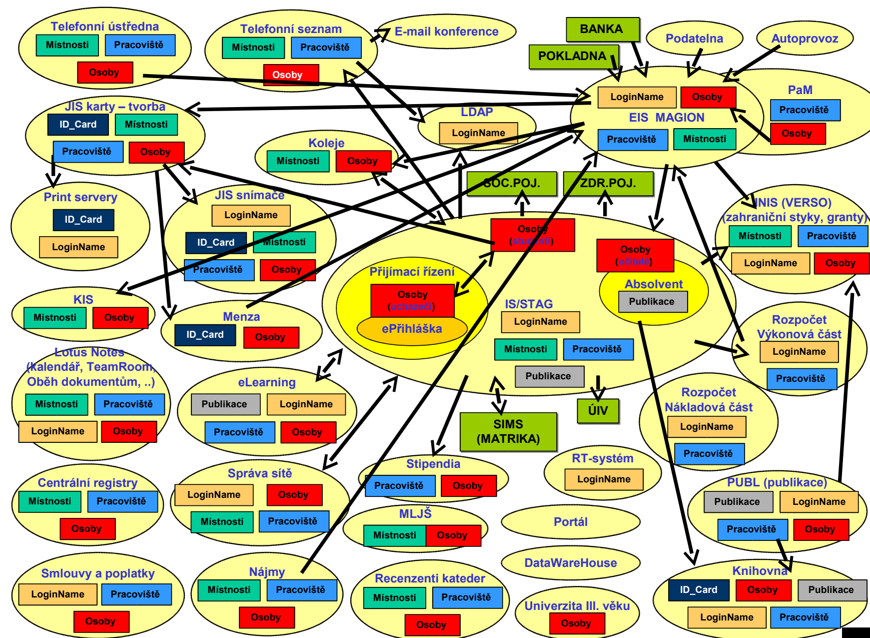
O něco radostnější je situace dnes, kdy vedení má zájem o zavádění moderních nástrojů do řízení univerzity. V roce 2003 jsme zavedli Lotus Notes jako

[2] Jedinaja sistema

[3] To byl oficiální důvod, dle mého názoru nebylo ekonomické vedení schopné ani po dvou letech jasně nadefinovat své požadavky.

nástroj pro řízení kolaborativní práce (*workgroup*), v současné době probíhá pilotní provoz elektronického oběhu dokumentů (*workflow*), v provozu je první pokus o MIS[4], těsně před uvedením do rutinního provozu je centrální registr osob.

Stále ještě přetrvává nepřiměřené množství oddělených agend, ostatně jak naznačuje schéma provázanosti [9] jednotlivých agend (obr. 1). Nehledejte zde dokonalou analýzu toku dat na univerzitě, tento obrázek je spíše pro management univerzity, zdaleka zde nejsou přesné vazby, závislosti a toky dat. Skutečnost je ještě komplikovanější.



Obr. 1 Agendy

Naším cílem je sjednotit datovou základnu, umožnit webový přístup k oprávněným informacím (prioritně pomocí portálové technologie) a umožnit získávat i komplexní informace pro vedení univerzity potřebné pro řízení univerzity (statistiky, trendy, grafické informace).

Hodně si slibujeme od zavedení centrálních registrů, které by měly zaručit nejen integritu dat, jednotnou datovou základnu, ale hlavně zprůhlednění všech datových vazeb a zavedení modelu $1 : N$ namísto stávajícího $N : N$.

[4] Management Information System

Cíle máme nelehké a otázka je, zda dosáhneme všeho, co máme v plánu. Při uvědomění, jak daleko jsme v zavádění moderního informačního systému, odpovídá realitě korekce nadpisu této kapitoly na „Hledání cesty k elektronickému úřadu“.

3.2 Studijní agenda

Přibližně v roce 1993 jsme se rozhodli a částečně byli i donuceni okolnostmi, že se o vývoj studijní agendy pokusíme vlastními silami na CIVu. První elektronický předzápis (výběr předmětu v kreditním typu studia) proběhl v roce 1994. Celý systém byl od počátku psán nad databázovým systémem ORACLE. Počátky byly na DOSovském klientu, v roce 1996 se začal používat windowsovský klient, v roce 2000 se objevil první webový klient. A že se čas nezastavuje, ukazují současné práce na portletech STAG[5].

Systém STAG je rozsáhlý systém, který není používán pouze na ZČU. V současné době jej kromě ZČU používá dalších 8 univerzit a jedna VOŠ. Systém se úspěšně zúčastnil soutěže *Eunis Elite Award* v Berlíně, kde obsadil třetí místo.

Studijní agenda je ukázka úspěšného informačního projektu. Vynikající práci při zavádění a tvorbě systému odvedl prorektor Ryjáček, který jen potvrdil známou skutečnost, že úspěšný projekt potřebuje silnou podporu managementu, jinak úspěšný nebude.

Zájemci o celou historii a podrobnosti mohou najít užitečné informace v [3] [5] [8].

3.3 Podpora výuky

Od CIV se očekává, že bude provozovat tzv. veřejné počítačové učebny, kde budou mít studenti k dispozici potřebný software, se kterým se setkávají ve výuce a který používají i jako pracovní nástroj (např. sepsání semestrální práce apod.). Jednoduché zadání, složitý úkol. Jednoduché zadání se totiž stane velmi komplexním při zjištění rozsahu požadavků, který je na funkčnost stanice ve veřejné učebně kladen.

Základní potřeby lze shrnout do následujích bodů:

- požadován OS Linux i Windows
- instalovaný systém musí být stabilní a robustní
- snadná a rychlá reinstalace
- stejné prostředí na všech stanicích ⇒ *roaming profile* pro uživatele

[5]STudijní AGenda.

- instalovaný software většinou umístěný na sdíleném souborovém systému AFS[6]

Výše uvedené požadavky jsou na CIV ZČU řešeny tzv. učebnovým obrazem pro OS Windows a systémem FAI[7] pro Linux. Všechny stanice mají duální *boot* a student si může zvolit potřebný OS. Kromě výběru OS je při počáteční volbě možné zvolit síťovou instalaci operačního systému a tím i snadné obnovení systému. Tato volba je přístupná pouze pro obsluhu učeben. Díky tomu, že většina aplikačního software je uložena na síťovém distribuovaném souborovém systému, jsou požadavky na kapacitu lokálních disků minimální a současná kapacita lokálních disků je na učebnách využita pouze z několika procent.

Centrální uložení aplikačního softwaru přináší jednoznačnou výhodu jednotnosti prostředí a centrálních aktualizací. To se projevuje zejména při potřebných datových aktualizacích, jako např. virová databáze, kterou stačí aktualizovat pouze jednou a změna se projeví na všech používaných počítačích. Důležitá vlastnost pro uživatele je i skutečnost, že profil není svázán s fyzickou stanicí, ale s uživatelským jménem, takže uživatel pracuje na kterékoliv stanici ve „svém" prostředí. Koncepce učebnových stanic se ukázala jako velmi stabilní a odolná proti agresivnímu prostředí studentských učeben.

3.4 Komunikační infrastruktura

Komunikační infrastruktura je oblast, kde se vývoj dá charakterizovat jako komplikovaná cesta od Ethernetu k Ethernetu. Honza Müller popsal ve svém příspěvku na konferenci EurOpen v Jetřichovicích vývoj síťových technologií a jejich slepých uliček. Byl to vlastně popis naší cesty. Načrtněme si zde pouze bodově použité technologie a zařízení ve spojení s časem:

1991 Tři segmenty Thin Ethernetu 10Base2, síť se používá pouze na CIV a katedře informatiky a výpočetní techniky. Síťový hardware je suplován PC stanicemi se softwarem PCroute, KA9Q.

podzim 1992 První budovy v novém vysokoškolském areálu Bory se strukturovanou kabeláží – úspěšně jsme se vyhnuli páteři na 10Base5 technologii a použili jednu z prvních páteří na FDDI technologii. Použitý hardware je již specializovaný – 3COM NetBuilder II.

1993 První propojení mezi kampusem a budovami ve městě pojítky SkyWalker a Microwave na svou dobu se slušnou rychlostí – 10Mb/s. Zato rychlost připojení na Internet [8] na konci roku 64kb/s.

[6] Andrew File System
[7] Fully Automated Installation, viz [4]
[8] V únoru 1993 zahájil činnost CESNET na rychlosti 19.2kb/s

1994 Připojen Cheb a univerzitní síť se rozrůstá o řadu budov a již se dá nazývat MAN[9]. První strukturovaná kabeláž *level 5* připravená i pro vyšší rychlosti.

1995 Získán grant na implementaci ATM[10] technologie. Rychlost připojení na Internet už na 256kb/s.

1996 První pokusy s ATM.

1997 Metropolitní propojení budov pomocí ATM s pronajatými optickými vlákny. V červnu podstatný skok v rychlosti připojení k Internetu na 34Mb/s – projekt TEN-34 CZ. Vybrané servery jsou již připojeny FastEthernet technologií.

1999 Připojena první studentská kolej. Ve spolupráci s magistrátem města Plzně začínáme budovat vlastní optickou síť založenou na temných vláknech.

2001 Připojení k Internetu povýšeno na 2.5Gb/s.

2002 ZČU přechází plně na technologii Ethernetu, páteřní technologie je Gigabit Ethernet. Slavnostně se loučíme s FDDI a ATM. Plný přechod na Cisco technologii.

2003 Začíná rozvoj a implementace technologie WiFi.

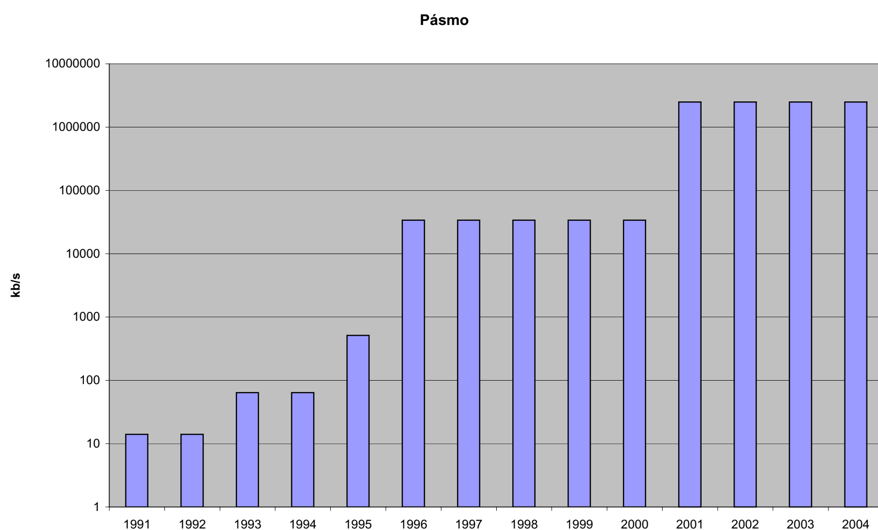
2005 Poslední spoj mikrovlným pojítkem mezi budovami nahrazen optickým vláknem. Univerzita má všechny své budovy propojeny vlastním optickým vláknem. Celkově máme propojeno 45 budov po celé Plzni a v Chebu.

Jako ilustrace vývoje poslouží sloupcový graf (obr. 2) ukazující šíři pásma. Nepřehlédněte, prosím, logaritmické měřítko na ose `Y`.

3.5 Průkopnická práce při zavádění nových technologií

Práce v univerzitním prostředí má svá specifika. Na jedné straně akademické svobody, které u nás chápeme především tak, že kromě běžných servisních a rutinních prací musí zbýt prostor na hledání nových cest v souladu s nejnovějšími trendy a co je důležité, alespoň v částečném souladu se zájmy konkrétního pracovníka. Jinými slovy, je zde prostor pro zajímavou práci, která by často (spíše většinou) nebyla akceptovatelná v komerční sféře. Výsledkem je pak výchova odborníků, kteří jsou specialisté na velmi perspektivní a zajímavé technologie již v době, kdy tyto teprve začínají pronikat do komerční sféry a tito lidé se stávají hledaným artiklem na trhu práce. Takže pojem fluktuace není pro CIV neznámým pojmem.

[9] Metropolitan Area Network
[10] Asynchronous Transfer Mode



Obr. 2 Vývoj šíře pásma

Srovnáním obsazení jednoho oddělení CIV v roce 1999 a v roce 2005 zjistíme, že průnik množiny zaměstnanců ve zmíněných rocích je pouhých 20 %. Tuto nepříjemnou skutečnost kompenzuje mládí a neopotřebovanost nových pracovníků, většinou čerstvých absolventů.

Nadpis této kapitoly trochu zapáchá samochválou, abychom ji trochu otupili, jsou v následujícím výčtu důležitých milníků vyznačeny slepé uličky *italikou*:

1993 *Zavádíme sdílení datové oblasti pro UNIX a Windows pomocí NOS PathWork firmy DEC.*

1994 *Projekt Athena univerzity Standford na správu koncových stanic [6].*

1995 *Volba Novell Netware 4.1 jako hlavní souborový systém.*

1995 Zřízení Superpočítačového centra ZČU, základ tvoří ve své době stroj v první stovce nejvýkonnějších počítačů DEC8400, 2GB RAM, 8 procesorů Alpha.

1996 Zavedení souborového distribuovaného systému AFS a autentizačního systému Kerberos [7].

1996 *Rozhodnutí o technologii ATM pro páteřní síť univerzity.*

1998 Zavedení čipové karty jako průkaz studenta. Použita karta s čipem E5560 [15].

1998 Zvolen nástroj *Request Tracker* pro uživatelskou podporu a sledování požadavků a jejich řešení [10].

1999 Začínají první hrátky s PKI.

2001 Technologie tenkého klienta pro Windows (produkt CITRIX) do rutinního provozu.

2002 Volba Lotus Notes pro podporu procesů při řízení univerzity.

2002 Použita technologie VoIP pro propojení telefonních ústředen. Celá univerzita včetně Chebu využívá počítačovou síť pro místní telefonní hovory. Výsledkem je pokles telefonních poplatků na 25 %.

2004 Zavedení jednotného přihlášení k webovým aplikacím WebAuth (univerzita Standford).

2005 Dokončován pilotní projekt provozu portálu WebSphere.

4 Závěr

Pokusil jsem se shrnout a popsat některé oblasti IT v univerzitním prostředí. Existuje řada dalších oblastí, spadající pod IT, které jsou zde buď lehce nebo nejsou vůbec zmíněny. Omlouvám se všem makáčům, kteří nenašli v příspěvku zmíněn svůj podíl na rozvoji IT, který posunul vývoj o kousek dopředu.

Zavádění nových technologií v univerzitním prostředí je cesta nejednoduchá. Velmi těžko se aplikuje metoda tvrdého manažerského rozhodnutí, běžná v byznysu[11]. Naštěstí rozvoj jde dopředu často díky získaným grantům, kde na úspěšné dořešení projektu lze navázat při zavádění technologie do rutinního provozu. Většina úspěšných aktivit byla podpořena získanými granty[12] a řešení grantu tvořilo odrazový můstek pro pokračování řešení do provozních záležitostí. Na druhou stranu, některé granty bychom měli psát *italikou.*

Literatura

[1] Balák, O.: *Zálohování dat pomocí Open Source software*, sborník XXV. konference, EurOpen, Teplá, 2004.

[2] Griessl, R., Okrouhlý, J.: *Windows NT v otevřeném výpočetním prostředí,* Sborník XV. konference, EurOpen, Nečtiny, 1999.

[11] business

[12] A tedy i grantovými penězmi.

[3] Holeček, P.: *Databázové aplikace využívající www jako rozhraní*, sborník XVII. konference, EurOpen, Kouty, 2000.

[4] Holeček, P.: *FAI na ZČU*, sborník XXI. konference, EurOpen, Znojmo, 2002.

[5] Jiroušek, P., Holeček, P., Kotouč, T: *Možnosti a zkušenosti s webovými aplikacemi na platformě Oracle*, sborník XIX. konference, EurOpen, Lísek, 2001.

[6] Kejzlar, L: *Athena*, sborník IX. konference, EurOpen, Tábor, 1995.

[7] Kejzlar, L., Sitera, J.: *No ID, no (Open) Beer aneb prokažte svou identitu*, sborník XVIII. konference, EurOpen, Dolní Malá Úpa, 2001.

[8] Kolektiv CIV: *Nedohraná partie.* CIV ZČU v Plzni, 2001.

[9] Kotouč, T.: *Prezentace pro management univerzity.* ZČU, 2005.

[10] Okrouhlý, J.: *Systém uživatelské podpory a jeho technické zabezpečení*, sborník XVI. konference, EurOpen, Velké Bílovice, 2000.

[11] Post, E.: *Opravdoví programátoři nepoužívají Pascal.* Datamation, 1983.

[12] Rudolf, V., Šmrha, P., Ledvina, J.: *The computer network of the University of West Bohemia.* ZČU, **25**, preprint, 1992.

[13] Urbanec, J.: *Když síťoví svatí nespí*, sborník XX. konference, EurOpen, Jetřichovice, 2002.

[14] Valdman, J.: *Co je to portál? Výběr portálového produktu a implementace portálu ZČU*, sborník XXIV. konference, EurOpen, Dolní Morava, 2004.

[15] Vituško, A.: *Jednotný identifikační systém na ZČU v Plzni*, sborník XXI. konference, EurOpen, Znojmo, 2002.

Agilní krabice?

Václav Pergl

E-mail: vpergl@kerio.com

Abstrakt

Agilní metodiky vývoje softwaru jsou v současné době často využívány pro tvorbu aplikací na zakázku s nejasně formulovanými a/nebo měnícími se požadavky. Náš příspěvek diskutuje některé možnosti užití agilních postupů při tvorbě krabicového software (COTS) v multiprojektovém prostředí softwarové společnosti. Svoji pozornost soustředíme zejména na možnosti a omezení při nasazení agilních principů v životním cyklu projektu tvorby krabicového software.

1 Manifest agilního vývoje softwaru

Pokud chceme hovořit o možnostech nasazení agilních principů při vývoji krabicového softwaru musíme začít od podlahy agilního přístupu – tzv. Manifestu agilního vývoje softwaru (Manifesto for Agile Software Development) z roku 2001, pod který připojilo své podpisy mnoho uznávaných vývojářů. Autoři vyjádřili přesvědčení, že existuje nová lepší cesta jak tvořit software, že oni po ní půjdou a budou na této cestě pomáhat i ostatním.

Autoři manifestu vyhlásili, že dávají přednost:

- **Individualitám a interakci** před procesy a nástroji.
- **Fungujícímu softwaru** před obsáhlou dokumentací.
- **Spolupráci se zákazníkem** před smluvním vyjednáváním.
- **Reakci na změnu** před sledováním plánu.

Přestože agilní metodiky jsou uvažovány zejména pro tvorbu aplikací na zakázku s nejasně formulovanými a/nebo měnícími se požadavky můžeme prohlásit, že i pro vývoj krabicového softwaru jsou hodnoty položek na levé straně nad hodnotou položek na pravé straně deklarace.

Agilní metody jsou spíše adaptivní nežli prediktivní. Při tradičním přístupu k tvorbě softwaru vytváříme dlouhodobé plány, které nepředpokládají měnící se požadavky. Naproti tomu agilní přístup počítá se změnami, které dříve či později nastanou. Agilní metody jsou také více orientovány na lidi než na procesy. Mnohem více spoléhají na schopnosti a znalosti navzájem dobře komunikujících členů vývojového týmu, než na tisícestránkové specifikace ISO 9000 či CMM level 5 procesů.

2 Principy vycházející z Agilního manifestu

Tvůrci manifestu formulovali následující základní principy agilních metodologií:

Naší nejvyšší prioritou je uspokojovat zákazníky včasnou a kontinuální dodávkou softwaru, přinášejícího hodnotu.

Zákazník získá největší užitnou hodnotu z dodaného softwaru, nikoliv ze stohů vývojářské dokumentace ve formě diagramů a modelů. Pro krabicový software je včasná dodávka produktu na trh ještě důležitější, než v případě vývoje zakázkového software pro konkrétního zákazníka.

Změny požadavků dokonce i v pozdních fázích vývoje jsou vítané, neboť změna může být pro zákazníka konkurenční výhodou.

Agilní metodologie očekávají změny a jsou na ně připravené. Při tvorbě krabicového softwaru jsou přicházející požadavky na změny rozsahu akceptovány až do konce fáze vývoje.

Dodávka software je realizována často (od několika týdnů do několika měsíců).

Agilní metodologie zdůrazňují velmi krátké iterace vývojového cyklu a rychlou dodávku softwaru. V případě krabicového software na fázi vývoje však navazuje další mnohdy stejně dlouhá fáze stabilizace a testování produktu u vybraného okruhu betatesterů.

Zákazníci a vývojáři spolupracují denně na projektu.

Agilní přístupy vycházejí z toho, že na začátku projektu neexistuje úplná specifikace požadavků. Zpočátku se obvykle definují klíčové požadavky, avšak očekává se, že i ty se mohou během vývoje měnit. V případě krabicového softwaru je vytvořen úvodní dokument specifikující předpokládaný rozsah projektu

(Scope/Vision dokument, Software Requirements Specification, Marketing Request Document,...). Častá zpětná vazba od požadavků zákazníků, vlastnostech konkurenčních produktů, trendech trhu musí být prováděna jak pracovníky Helpdesku tak i marketingem a obchodním oddělením společnosti.

Projekt je postavený na motivovaných jedincích, kteří mají vytvořeny dobré podmínky k práci a mají důvěru.

O úspěchu či neúspěchu projektu rozhodují konkrétní lidé, pro které je velice cenná důvěra v jejich schopnosti. Každodenní rozhodování na projektu musí provádět kompetentní a pozitivně motivovaní členové týmu.

Nejúčinější a nejefektivnější metodou přenosu informací ve vývojovém týmu je vzájemná komunikace.

Agilní přístupy vycházejí z předpokladu, že smyslem projektové dokumentace je potvrzení porozumění řešenému problému a toto porozumění je možné dosáhnout využitím přímé komunikace a nikoliv sepisováním a studováním obsáhlých dokumentů.

Fungující software je primární mírou pokroku.

Standardní součástí agilních metodik je automatické vytváření každodenních buildů, navazující generace instalačních balíčků pro podporované platformy, automatická instalace na testovací hardware a spuštění rozsáhlých testů.

Agilní metodologie předpokládají udržitelný vývoj.

Přetěžování klíčových pracovníků sice může krátkodobě vyřešit problémy projektu, ale dlouhodobě je zdrojem nízké produktivity práce. Nutnost práce přesčas je téměř vždy signálem závažných problémů.

Trvalá pozornost perfektnímu technickému řešení a kvalitnímu návrhu

Mnoho agilních přístupů zdůrazňuje kvalitu návrhu a přicházející změny v návrhu nepovažuje za chybu. Změny mohou přicházet i v době, kdy je celý kód již napsán a je nutné změnit návrh a tyto změny promítnout do zdrojového kódu.

Jednoduchost – umění maximalizovat množství práce, kterou neprovedeme – je zásadní.

V agilních postupech klademe důraz na jednoduché procesy, které se snadno mění. V případě krabicového softwaru je důležité implementovat nejdříve a rychle požadavky, které přinesou hodnotu a pro většinu uživatelů.

Nejlepší architektura, požadavky a návrhy vznikají ze samoorganizujících se týmů.

Motivované osobnosti přináší nejlepší návrhy. Je třeba je podpořit důvěrou a každodenní komunikací.

Tým se pravidelně zabývá otázkou, jak pracovat efektivněji, jak se dále rozvíjet a přizpůsobovat.

Dobrý tým ví lépe než kdo jiný o svých slabých stránkách a potenciálu ke zlepšení.

Integrace podnikových aplikací pomocí open-source nástrojů

Maxmilián Otta

E-mail: otta@civ.zcu.cz

1 Úvod

V současnosti je ze strany provozovatelů informačních systémů vyvíjen stále větší tlak na softwarové firmy, aby integrovaly jejich ať už stávající či nově vyvíjené aplikace mezi sebou, neboť informační systémy organizace přestávají být „izolovanými ostrůvky dat“. Tento příspěvek si klade za cíl prezentovat existující standardy v oblasti integrace podnikových aplikací a ukázat konkrétní řešení integrace aplikací v univerzitním prostředí, které je typicky velmi heterogenní, pomocí dostupných open-source nástrojů.

Současný stav systémové integrace, alespoň v univerzitním prostředí, je spíše živelný, založený na unikátních řešení, která mají typicky podobu „skriptových“ datových pump či originálních „hacků“. Tato integrace na úrovni dat, kdy přistupujeme přímo do báze dat aplikace, je sice levná a rychlá, ale obchází aplikační logiku a tím může snadno dojít k narušení konzistence databáze. Výsledkem této živelné integrace je prostředí s neprůhlednou topologií a vstupními body do různých vrstev aplikací.

Cílovým stavem by jistě měla být jednotná integrační architektura definující komunikační infrastrukturu a vlastnosti konektorů do jednotlivých aplikací. Nové aplikace by měly být již navrženy tak, aby snadno přilnuly k tomuto prostředí, je však zřejmé, že u většiny stávajících aplikací se nevyhneme integraci na úrovni dat díky jejich typicky dvouvrstvé architektuře – relační bázi dat s tlustým klientem. Propojení na úrovni dat je ale v tomto případě skryto uvnitř konektoru do aplikace. Integrace aplikací dále nepředstavuje jen jejich propojení a vzájemnou součinnost prostřednictvím middleware, ale i jejich napojení na autentizační infrastrukturu orgranizace. Tímto aspektem se však tento článek nezabývá.

Na tomto místě bych chtěl také učinit jednu praktickou poznámku: představa snadné integrace nově vyvíjených aplikací je do jisté míry naivní, a to především z toho důvodu, že většina tuzemských softwarových firem má svoje zaběhnuté technologie a nerada adoptuje nové ať již z nedostatku zdrojů či nedůvěře vůči

neznámému. Z toho plyne, že dosažení cílového stavu zmíněného v předchozím odstavci bude jistě dlouhodobý proces.

2 Způsoby a architektury integrace

Lze říci, že integrační platforma stojí na těchto třech pilířích: komunikační infrastruktuře (middleware), společné reprezentaci dat a konektorech do aplikací. Z pohledu komunikační infrastruktury se může jednat buď o těsně (*synchronní*) nebo volně (*asynchronní*) vázané systémy s možností „*one-to-one*“ nebo „*one-to-many*“ komunikace. Volně vázané systémy mají bezesporu výhodu větší nezávislosti a v případě výpadku nebo odstávky některé z aplikací mohou určitou dobu pracovat autonomně. Komunikace „*many-to-many*“, kterou umožňuje sběrnicová nebo hvězdicová topologie middleware, je základem pro integrační infrastrukturu umožňující směrování a transformaci zpráv, nezbytně nutnou chceme-li dosáhnout integrace na úrovni procesů. V tomto smyslu byla navržena i integrační architektura aplikací na Západočeské univerzitě v Plzni popsaná v dalších odstavcích.

2.1 Způsoby integrace

Jak již bylo naznačeno v úvodu, existují různé úrovně integrace aplikací. Jejich přehled včetně výhod a nevýhod ukazuje tabulka 1.

2.2 Architektury integrace

Co se týče architektur middleware, jsou v současnosti využívány především:

MOM – Message Oriented Middleware – umožňuje asynchronní komunikaci. K realizaci „many-to-many“ komunikace vyžaduje centrální komponentu – *Message Broker.*

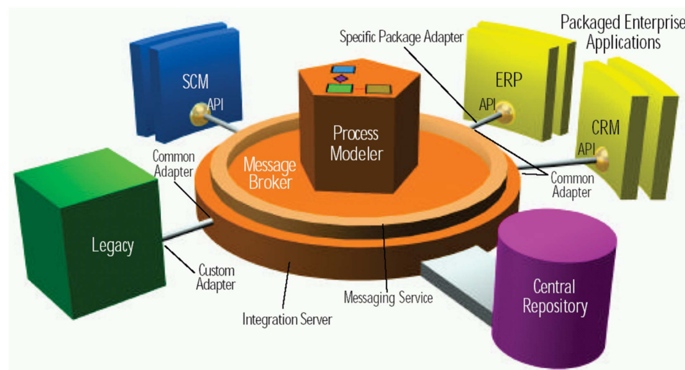
SOA – Service Oriented Architecture – na ní staví především hojně využívané *Web Services.* Hlavní myšlenkou je poskytnutí rozhraní aplikací jako služby, které lze vyhledávat v síti.

ESB – Enterprise Service Bus – je instrastruktura zajišťující dynamické směrování zpráv, jejich transformaci a zabezpečení a podporující začlenění komponent pro řízení procesů, registraci a vyhledávání aplikací (služeb).

Typická architektura umožňující integraci na úrovni procesů je tvořena třemi vrstvami: systémem pro zasílání zpráv (*MOM*), systémem pro směrování správ (*Message Broker*) a systémem modelování procesů (*Process Modeler*). Celkové schéma takovéto architektury je vidět na obrázku 1.

Tabulka 1

	Výhody	**Nevýhody**
Integrace na úrovni dat	• jednoduchá • levná	• obchází aplikační logiku • vyžaduje detailní znalost datových struktur aplikace • není vždy možná
Integrace na aplikační úrovni pomocí MOM	• používá API aplikace • asynchronní • možnost transformace a směrování dat podle jednoduchých pravidel	• bez podpory managementu pravidel
Integrace na úrovni procesů	• existují nástroje na modelování procesů • existují standardy pro popis procesů • snadná škálovatelnost	• složitost • nesnadná implementace



Obr. 1 Architektura pro integraci na úrovni procesů

3 Otevřené standardy a technologie

Chceme-li dosáhnout modulárnosti, interoperability a budoucí rozšiřitelnosti integrační platformy, je třeba se při jejím návrhu opřít o otevřené standardy. Cílem tohoto odstavce je podat stručný přehled standardů, které je možné použít na jednotlivých vrstvách integrační platformy.

3.1 Konektory

Konektory (někdy též *adaptéry*) mohou být do aplikací zapojeny na různých úrovních. Podle nich bychom měli volit i odpovídající standard, jako příklad lze uvést některé typické zástupce:

datová úroveň – *SQL – Structured Query Language*, *ODBC – Open DataBase Connectivity* nebo *JDBC – Java DataBase Connectivity* rozhraní.

aplikační úroveň – *CORBA – Common Request Broker Architecture*, *JCA – J2EE Connector Architecture; JSR-16.*

3.2 Komunikační subsystém

Při volbě *Message Oriented Middleware – MOM* bychom měli hledět na to, jaká standardní rozhraní nabízí na straně konektorů a na straně systému směrování a transformace dat. V současné době přichází v úvahu jedině tyto dva standardy: *Web Services SOAP* a *JMS*, přičemž posledně jmenovaný je v současné době již podporovaný téměř všemi *MOM* různých výrobců. *JMS* je *MOM* podporující synchronní i asynchronní zasílání zpráv systémem *point-to-point* (topologie *one-to-one*) nebo *publish/subscribe* (topologie *many-to-many*). Dále podporuje transakce, persistentní fronty zpráv, nastavení doby života zprávy a doručení změškaných zpráv příjemci při výpadku nebo odpojení.

3.3 Modelování a řízení procesů

Celou integrační infrastrukturu uvedeme do života až definicí směrovacích a transformačních pravidel, které „vykonává“ *Message Broker* a koordinuje tak interakci jednotlivých aplikací. Tato pravidla popisují jednotlivé procesy v organizaci a bylo by žádoucí, aby byl k dispozici nástroj (*Process Modeler*) pro jejich snadný popis, *Message Broker* schopný je interpretovat, aby byla uložena v snadno přenositelném formátu a aby byla nezávislá na použitém middleware. I v této oblasti vzniklo již několik standardů:

XPDL – *XML Process Definition Language.* Jedná se v podstatě o jazyk založený na popisu topologie Petriho sítě.

BPEL – *Business Process Execution Language.* Je v současnosti nejvýznamnější standard umožňující spojení synchronních i asynchronních služeb tak, aby implementovaly transakční procesní toky. Jeho základní charakteristikou je využití *Web Services* jako komponentového modelu a datový model založený na *XML.* Za zmínku stojí, že firma *Oracle* nabízí volně ke stažení

Oracle BPEL Process Manager jako plugin do vývojového prostředí *Eclipse*. *Oracle BPEL Process Manager* je vizuální nástroj pro modelování procesů, které popisuje pomocí *BPEL*.

3.4 Java Business Integration

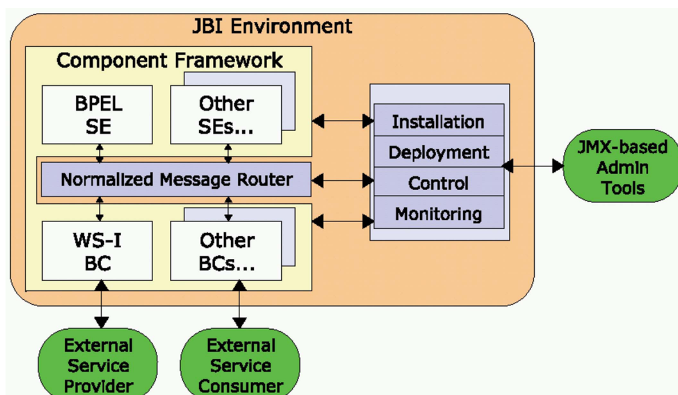
Za zmínku stojí i nově připravované standardní Java API *Java Bussines Integration – JSR-208*. To přejímá *SOA* jako komponentový model a jeho hlavní myšlenkou je umožnění realizace integračních infrastruktur formou vyměnitelných komponent. Definuje komponenty dvou typů:

Service Engine (SE) – poskytuje implementaci procesů (*Business Logic*) a transformační služby jiným komponentám, ale může být i jejich konzumentem.

Binding Component (BC) – zajišťuje konektivitu s vnějším prostředím a též umožňuje tak integraci externích služeb (aplikací) do společného prostředí.

Tyto komponenty spolu nikdy nekomunikují přímo, ale pouze přes takzvané *JBI* prostředí. Dále definuje obecný směrovač zpráv – *Normalized Message Router (NMR)* (*Message Broker*), entity podílející se na procesu výměny zprávy, formát obecné zprávy a zajištění kvality služeb. *JBI* komponenta má k dispozici API, přes které může získat kanál k NMR, zaregistrovat svoje rozhraní v *JBI* nebo vyhledávat rozhraní jiných komponent či služeb. Hrubý pohled na architekturu *JBI* ukazuje obrázek 2.

Kromě toho jsou v *JSR-208* specifikovány i standardní J2EE API související s životním cyklem komponenty včetně *JMX (Java Management Extension)*.



Obr. 2 Rámcová architektura Java Business Integration

4 Přehled open-source produktů

4.1 OpenAdaptor

OpenAdaptor – http://www.openadaptor.org je nástroj určen především pro tvorbu konektorů do aplikací, ale vystačí i na integraci aplikací. Jeho princip spočívá v tom, že uživateli poskytuje sadu standardních datových zdrojů a cílů, kterými mohou být relační báze dat, soubory (*XML i CSV*), sockety, FTP servery, e-mail, RMI, LDAP či některý ze standardních *MOM* (*JMS, Tibco Rendezvous*). Nále nabízí i tzv. *roury* umožňující transformaci dat včetně jejich šifrování. Pomocí vizuálního nástroje *Apator Framework Editor* lze snadno navrhnout konektor, jehož struktura je uložena v XML souboru, podle které pak runtime implementuje daný konektor. Celé programové vybavení je implementováno v jazyce Java a je tedy platformně nezávislé.

4.2 OpenJMS

OpenJMS – http://openjms.sourceforge.net je implementace *JMS* podporující kromě standardních vlastností i persistentní fronty, lokální transakce, autentikaci a jako transportní vrstvu *RMI*, *TCP*, *HTTP* a *SSL*. Lze ji instalovat jako komponentu do standardního J2EE kontejneru.

4.3 JBoss MQ

JBoss MQ – http://www.jboss.org je standardní komponentou aplikačního serveru *JBoss* a stejně jako *OpenJMS* nabízí persistentní fronty, automatický *server fail-over* a doručení „zmeškaných“ zpráv během výpadku.

4.4 JORAM

JORAM (Java Open Reliable Asynchronous Messaging) –
http://joram.objectweb.org je další implementace *JMS* poskytující navíc persistentní, spolehlivý a distribuovaný JNDI server, rozdělování zátěže a bridge pro připojení jiných *JMS* serverů.

4.5 MULE

MULE – http://mule.codehaus.org je *ESB* infrastruktura se směrovačem zpráv. Tento *Message Broker* podporuje různé strategie směrování zpráv, které vycházejí z integračních návrhových vzorů publikovaných v knize *Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns*. Zmíněné návrhové vzory jsou publikovány na webové stránce věnované této knize:

http://www.enterpriseintegrationpatterns.com.

4.6 WfMOpen

WfMOpen – http://wfmopen.sourceforge.net je *Workflow Engine* implementace *Workflow Management Facility Specification* od *OMG*, kterou lze nasadit do běžného J2EE serveru jako *EJB* se sadou *JMS* front. Pro popis procesů používá *XPDL*. Podobných implementací existuje více, jejich seznam lze nalézt na adrese: http://java-source.net/open-source/workflow-engines.

5 Řešení integrace aplikací na ZČU

Požadavek centrálního registru osob na Západočeské univerzitě v Plzni nevznikl pouze z nutnosti soustředit určitý druh dat do jednoho místa, ale i díky postupné integraci aplikací do intranetového portálu a především ze snahy zlepšení služeb studentům a zaměstnancům zjednodušením administrativních úkonů součinností klíčových agend. Mezi tyto agendy patří zejména:

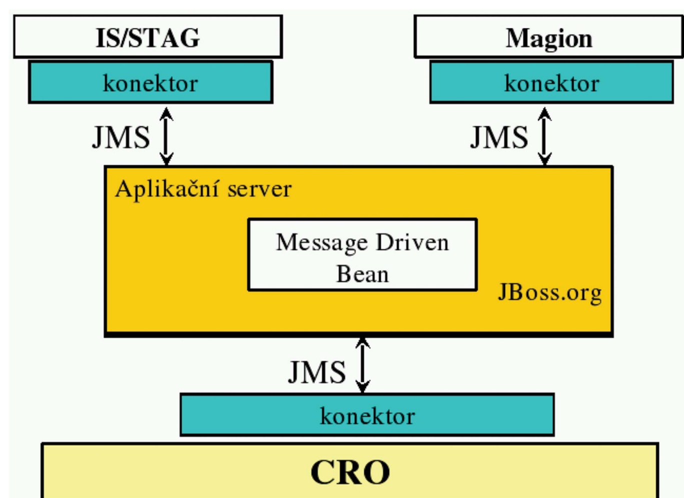
- **IS/STAG** – informační systém studijní agendy,
- **Magion** – personalistika a ekonomika,
- **Orion** – výpočetní prostředí s autentizační infrastrukturou *Kerberos* a adresářovými službami nad *OpenLDAP* a
- **JIS** – jednotný identifikační systém využívající bezkontaktních čipových karet.

5.1 Struktura centrálního registru

Jádro centrálního registru osob *CRO* tvoří relační databáze, ve které jsou uloženy vybrané informace o osobách, které jsou s univerzitou nějakým způsobem ve styku. Výhledově budou v registru i číselníky budov, místností a pracovišť. Datový model *CRO* v současné době reflektuje obecnou osobu (fyzickou i právnickou), aplikaci a takzvanou *pozici*, která je definována jako dvojice *(osoba, aplikace)*. Pozice tedy poskytují informaci o tom, v jakých agendách je daná osoba evidována. Logika registru je zabudována do aplikačního serveru, který komunikuje s okolím výměnou *XML* zpráv přes konektor *CRO*. Přes tento konektor je okolním aplikacím k dispozici aplikační rozhraní CRO – *CRO API*. Toto API poskytuje základní operace nad osobou a pozicí a vyhledávání v registru podle různých kritérií. Stejným způsobem jsou do infrastruktury zapojeny i ostatní aplikace, jejichž API v této chvíli umožňuje pouze registraci, aktualizaci a rušení osob. Jedná se především o aplikace výše jmenovaných agend, ostatní aplikace budou v tomto ohledu spíše pasivní a budou využívat *CRO* převážně

jako adresářovou službu (CRO bude zřejmě nutné pro některé stávající aplikace zpřístupnit i přes LDAP).

Komunikační subsystém využívá *JMS* jako *MOM*, přes který probíhá výměna *XML* zpráv. Jako poskytovatel *JMS* je použito *JBossMQ*, které je součástí aplikačního serveru *JBoss*. V tomto aplikačním serveru je i naistalována komponenta implementující jednoduchý *Message Router* jako *Message Driven EJB*. Konektory do aplikací a *CRO* jsou realizovány pomocí nástroje *OpenAdaptor*, jejich podrobnější popis je v odstavci 5.4. Celková architektura řešení je vidět na obrázku 3.



Obr. 3 Rámcová architektura CRO

5.2 Struktura XML zpráv

Jak již bylo řečeno dříve, aplikace komunikují s *CRO* výměnou *XML* zpráv. Každá funkce *CRO API* je volána odpovídající zprávou. Zpráva obsahuje identifikátor volané funkce, identifikátor transakce, zdroj a cíl zprávy a množinu parametrů závislou na volané funkci.

```
<?xml version="1.0" encoding="UTF-8"?>
<MESSAGE>
    <ZALOZ_OSOBU>
        <TID>29875873645</TID>
        <SOURCE>STAG</SOURCE>
        <DESTINATION>CRO</DESTINATION>
```

```
        <JMENO1>PETR</JMENO1>
        <JMENO2>PETROVIČ</JMENO2>
        <PRIJMENI1>POKUSNÝ</PRIJMENI1>
        <PRIJMENI2>POKUSOV</PRIJMENI2>
        <TITULPRJM>ING</TITULPRJM>
        <TITULZAJM>CSC</TITULZAJM>
        <RODCISLO>6112012114</RODCISLO>
        ...
        <LOGIN>POKUSNY</LOGIN>
        <PRACOVISTE>CIV</PRACOVISTE>
        <TYP_OSOBY>STUDENT</TYP_OSOBY>
        <STATNI_PRISLUSNOST>ČR</STATNI_PRISLUSNOST>
    </ZALOZ_OSOBU>
</MESSAGE>
```

5.3 Podmíněnost pozic

Má-li být například do studijní agendy zaveden nový student, musí být nejdříve evidován v jednotném identifikačním systému a mít aktivní kartu. To znamená, že pozice *STAG* je podmíněna pozicí *JIS*. Podmíněností pozic je vlastně implementován primitivní *workflow*.

5.4 Řešení konektorů

Konektory do aplikací jsou řešeny pomocí nástroje *OpenAdaptor* na datové úrovni. *OpenAdaptor* umožňuje snadnou generaci *XML* zpráv ze záznamů v databázi a dovede opět *XML* zprávy transformovat do posloupností *SQL* příkazů. Propagace událostí jako je přidání, změna nebo zrušení osoby je realizována pomocí *databázových triggerů*, které jednotlivé operace logují do zvláštní tabulky. Záznamy z této tabulky, která vlastně plní funkci interní fronty zpráv, pak vybírá *OpenAdaptor* a transformuje je do *XML* zpráv.

5.5 Globální identifikátory

Aby bylo možné identifikovat obecnou osobu v rámci celé infrastruktury a provázat tak údaje této osoby mezi všemi aplikacemi, musí jí být přiřazen globální identifikátor. Ten generuje *CRO*, a pokud aplikace založí novou osobu, obdrží zpět od *CRO* její globální identifikátor. Jinak osoba může být i jednoznačně identifikována identifikátorem aplikace, ve které vznikla a jednoznačným identifikátorem osoby v této aplikaci.

5.6 Implementace směrovače zpráv

Směrovač zpráv je v této fázi implementace *CRO* velice jednoduchý. Je to *Message Driven Java Enterprise Bean* v aplikačním serveru *JBoss*, který na základě pole `DESTINATION` ve zprávě přepošle zprávu do *JMS* fronty daného cíle.

6 Závěr

Motto: The easier it is to do, the harder it is to change. (Eng's Principle)

Integrace aplikací není ryze technologickým problémem, ale vyžaduje i rozsáhlou analýzu aplikací, datových toků a procesů. Je to dlouhodobý projekt, ve kterém je nutné postupovat po menších krocích, než zvolit všeobjímající řešení. Správná volba standardů a architektury je zásadní pro úspěch projektu neboť nad touto infrastrukturou budou v budoucnu vznikat důležité implementace podnikové logiky, které budou muset přetrvat možné technologické změny v architektuře této infrastruktury. Zkušenost je taková, že nejdéle přežívají řešení, u nichž se již od počátku počítá se změnami.

Během implementace integrační infrastruktury je zároveň velice důležité důsledně dokumentovat konektory do jednotlivých aplikací, podnikovou logiku, procesy a topologii infrastruktury. Ideálním stavem je katalog konektorů, procesů a transformací dat uložených nejlépe ve snadno přenositelném standardním formátu, čímž dosáhneme jejich znovupoužitelnosti.

Na závěr bych chtěl ještě připojit několik otázek, na něž jsme hledali odpovědi, ze kterých postupně vyplynuly požadavky na vlastnosti *middleware*:

- Jak velký počet aplikací chci propojit?
- Budou některé aplikace do centrálního registru i zapisovat?
- Které z aplikací zapisujících do registru budou primárním zdrojem konkrétních dat?
- Vyžaduje uživatel okamžité potvrzení úspěšnosti operace?
- Jaká má být maximální doba odezvy při komunikaci aplikací?
- Jak mají být směrována, ověřována a transformována data tekoucí mezi aplikacemi?
- Jaké jsou možnosti napojení middleware na jednotlivé aplikace?
- Jak budou již nyní předvídatelné změny náročné na dostupné zdroje?

Softwarové patenty – minulost, přítomnost, budoucnost

Jan Matějka
E-mail: jan@matejka.us

Takový vševládný stát (lhostejno jaké formy) roztáhne po zemi svou právní síť, a člověk bude běhat jako zajíc sem a tam, aby našel v této síti díru či větší oko, aby se dostal na místečko, kde by si oddechl a zašeptal: „Zaplať pánbů, tady jsem sám, tady si zaskotačím.“ Pozorujte proto naše zákonodárství z tohoto hlediska a dávejte pozor, abychom na místě svobody netvořili nové formy nevolnictví.

Svoboda, E.: *O vývoji v právu*, Praha 1926, s. 22

Když jsem byl požádán, zda bych nechtěl napsat svůj názor na připravovanou směrnici o softwarových patentech, tak jsem byl poděšen a potěšen zároveň. Poděšen proto, že v naprosté většině článků (včetně souvisejících diskusí), které se dosud touto problematikou zabývaly, výrazně převažovaly emoce nad věcnou podstatou problému, a mohl bych tedy oprávněně očekávat, že ať napíši v zásadě cokoliv, sklidím vždy hojnou dávku kritiky. Potěšen pak proto, že problematiku patentovatelnosti počítačových programů považuji za velmi zajímavou, navíc v odborné (nejen právnické) literatuře až nebezpečně opomíjenou. Z pohledu právní teorie (doktríny) a praxe pak navíc rozhodně nejde o problematiku zcela jednoznačnou, nehledě pak na to, že zejména s přibývajícím počtem těchto emotivních článků (a to zejména na Internetu) nejasností a mlhy spíše přibývá, než aby tomu bylo naopak. Už asi tušíte, že jsem nabídku přijal.

Právní způsoby ochrany počítačových programů (nejen autorské, ale i patentové právo)

Snad každé malé dítě ví, že jsou počítačové programy chráněny prostřednictvím autorského práva. Autorské právo lze ale považovat za celosvětově univerzální a relativně jediný fungující a existující nástroj právní ochrany počítačových programů. Nejde sice o řešení ideální či vhodné (navíc v mnoha ohledech neodpovídá

dynamice tohoto oboru), ale lze říci, že jde o řešení poměrně tradiční a dlouhodobě fungující. V každém případě byl, zhruba od poloviny osmdesátých let, tento způsob ochrany počítačových programů masivně kodifikován a transponován do naprosté většiny právních řádů, a to včetně všech evropských států, kterým byla předlohou Směrnice Rady EHS 91/250 ze dne 14. 5. 1991 o právní ochraně počítačových programů.

Ochrana autorským právem je v mnoha ohledech výhodná (poskytuje se automaticky, a téměř po celém světě, je zdarma...), avšak v některých ohledech již tak výhodná není. Předně jde o to, že nechrání (ani nemůže chránit) prostou myšlenku, ale až její tvůrčí zpracování (vyjádření). Autorskoprávní ochranu tedy lze uplatnit pouze proti šíření takovéhoto vyjádření (např. proti neoprávněnému šíření počítačových programů, tj. např. proti softwarového pirátství). Naopak autorské právo nelze použít proti tomu, kdo si „pouze vypůjčí“ myšlenku, způsob, postup, resp. metodu nějakého řešení, a v důsledku čehož pak vytvoří tuto myšlenku (...) vyjádří prostě jinak.

Srovnáme-li ochranu autorským právem s ochranou poskytovanou právem patentovým, dojdeme nepochybně k závěru, že v některých ohledech autorskoprávní ochranu výrazně přesahuje. Řízení o udělení patentu (a tedy i o poskytnutí ochrany) je sice poměrně drahé a dlouhodobé, avšak ve svém důsledku směřuje k jakémusi absolutnímu monopolu ve vztahu k patentovanému myšlence (postupu, výrobku, metodě...). Na rozdíl od práva autorského se tedy lze této ochrany domáhat i proti těm, jež k témuž dospěli sice sami a nezávisle, ale později. Nelze se tedy divit, že jakmile si velcí softwarový hráči uvědomili existující hranice autorskoprávní ochrany, pokusili se své produkty (viz níže) podřadit pod ochranu poskytovanou patentovým právem, a tam, kde nebyli úspěšní, se pokusili hranice patentové ochrany rozšířit. Patrně i z tohoto pohledu se pak nelze divit, že právě ve Spojených státech, v zemi sídla naprosté většiny světových výrobců (nejen) software, lze patentovat téměř cokoliv, a to včetně software.

Možná teď čekáte, že začnu psát o tom, jak je tento přístup (model) Spojených států právně špatný a morálně zkažený, jak znemožňuje konkurenci a soutěž mezi výrobci software, protože zavádí ochranu prosté myšlenky, postupu, nápadu, resp. příkazu, na jehož základě počítač pracuje, jak nesporně vede k monopolizaci trhu za současného vytváření celé řady kartelových dohod, které ve svém důsledku povedou k výraznému zpomalení vývoje software. Touto cestou však nepůjdu, a to hned z několika důvodů.

Předně proto, že i ve Spojených státech, kde je patentovatelnosti počítačových programů (z důvodů uvedených výše) zjevně nejrozšířenější, existuje poměrně rozsáhlá rozhodovací praxe, která, na rozdíl od praxe evropské, stanoví poměrně jasné mantinely patentovatelnosti (např. bylo několikráte judikováno, že abstraktní myšlenky a metody jsou z patentovatelnosti vyloučeny!). Dále pak proto, že právní systém Spojený států, jakkoliv se nám může zdát v mnoha ohledech na hlavu postavený, obsahuje celou řadu poměrně efektivních a vyvážených

brzd, pojistek a dalších nástrojů, které jsou způsobilé zjevně spekulativním jednáním zabránit (jde o poměrně složitý komplex různých „*catch all*" a „*fair use*" ustanovení, jehož byť i jen letmý nástin by výrazně přesáhl rozsah tohoto článku). V nespolední řadě pak proto, že Spojené státy rozhodně nejsou jedinou zemí, kde lze úspěšně patentovat počítačový program. Naopak lze říci, že až na některé dílčí odlišnosti, lze patentovat počítačový program v zásadě kdekoliv v civilizovaném světě, Českou republiku nevyjímaje. I v tom je tak trochu zakopaný pes (viz níže).

Patentovatelnost počítačových programů v Evropě (ČR nevyjímaje)

S účinností od 1. 7. 2002 přistoupila Česká republika k Úmluvě o udělování evropských patentů z roku 1973 (*European Patent Convention* dále jen EPC). Tímto přistoupením se Česká republika zavázala transponovat do svého právního řádu jak evropské patentové právo, tak i samotnou rozhodovací praxi orgánů Evropského patentového úřadu (*European Patent Office* dále jen EPO), což se také (zákonem č. 116/2000 Sb.) stalo.

Stávající Evropský přístup k patentovatelnosti softwaru je poněkud složitější než otevřeně liberální americký postoj. Definice předmětu patentovatelného vynálezu obsažená v ustanovení článku 52 odst. 2 EPC totiž z předmětu ochrany patentovým právem výslovně vylučuje mimo jiné:

- **objevy, vědecké teorie a matematické metody;**
- **plány, pravidla a způsoby vykonávání duševní činnosti, hraní her nebo vykonávání obchodní činnosti, jakož i programy počítačů;**
- **podávání informací.**

Na první pohled by se tedy mohlo zdát, že počítačové programy tedy nelze úspěšně (s ohledem na článek 52 dle EPC) patentovat. Není tomu tak. Již zmíněná dosavadní rozhodovací praxe EPO v mnoha případech došla (výkladem) k závěru, že v některých případech nelze počítačovým programům ochranu patentovým právem odpírat. EPO zde totiž vycházel z maximy, že v zásadě jakýkoliv počítačový program (software) lze snadno přeměnit přístroj, resp. zařízení (hardware), a to např. pouhým přehráním na ROM počítače, flashdisk, či v zásadě jakýkoliv fyzický nosič záznamu. V takovém případě by pak stejně bylo vždy možné, a dnes tomu tak u nás je, že se snadno přemění patentový nárok na počítačový program v nárok na účelový vázaný hardware, a jinými slovy, že se tak „obejde" ustanovení článku 52 EPC! Právě z těchto důvodu došla rozhodovací praxe EPO k závěru, že *V obecné rovině platí, že vynález, který by byl*

způsobilý k patento-právní ochraně podle konvenčních kritérii patentovatelnosti, by neměl být vyloučen z ochrany pouze z toho důvodu, že k jeho realizaci bylo užito moderních technických prostředků ve formě počítačového programu. O něco později EPO ještě dovodil, že *nárok směřující k technickému procesu prováděnému pod kontrolou počítačového programu nelze považovat za nárok směřující k počítačovému programu jako takovému.* Navzdory výše zmíněnému článku 52 EPC, hovoří stávající odhady o tom, že EPO již udělil více jak 20 000 patentů na počítačově programy!

Jinými slovy, patentovatelnost počítačových programů v Evropě (i u nás), a v době nedávné ani nikdy nebyla, zcela vyloučena. Především z tohoto pohledu je třeba nahlížet na novou směrnici o softwarových patentech, resp. přesněji směrnici o patentovatelnosti počítačových vynálezů (Directive on the patentability of computer-implemented inventions).

Směrnice o patentovatelnosti počítačových vynálezů

Aniž bych se zde chtěl zabývat poněkud trnitou legislativní cestou této směrnice, tak považuji za nezbytné zde zmínit, že v současné době se směrnice již „doklopýtala“ do fáze návrhu. Evropský parlament totiž tuto směrnici, jak ji navrhovala Evropská komise, poměrně důkladné připomínkoval a prosadil celou řadu změn a doplňku (role Evropského parlamentu zde byla a stále ještě je poměrně významná, v oblasti duševního vlastnictví má Evropský parlament velmi silnou pozici – jde o tzv. spolurozhodovací proceduru).

Základním účelem směrnice (a především to je dle komunitárního práva závazné) však kupodivu není zakotvení či vyloučení možnosti patentovat počítačové programy, ale zejména odstranění nedostatku právní jistoty pramenící ze současného nejasného právního stavu, kde na jedné straně EPC (článek 52) výslovné vylučuje počítačové programy z patentové ochrany, ale na straně druhé ji EPO, podlé téhož ustanovení, několika tisícům počítačových programů ročně přiznává!

Samotný (nový) text směrnice, a to mimo jiné v ustanovení článku 4, přiznává patentovou způsobilost všem počítačovým vynálezům (computer-implemented invention), které jsou průmyslově využitelné (resp. mohou být), nové a představují výsledek vynálezecké činnosti formou technického příspěvku ke stávajícímu stavu techniky. Takové vymezení směrnicí pak skutečně, až na některé výjimky, odpovídá rozhodovací praxi EPO. To ostatně dokládá i nově (Evropským parlamentem) přidaný článek 4a této směrnice pak obsahuje výčet toho, co předmětem patentové ochrany být již nemůže. Na první místě (článek 4a odst. 1) je zde uvedena zásada, že *Počítačový program jako takový patentovatelnost nezakládá.*

(*A computer program as such cannot constitute a patentable invention*), což ve skutečnosti znamená, že počítačový program sám o sobě, který nesplňuje výše uvedené znaky (průmyslovou využitelnost, novost...), nemůže požívat ochrany evropského patentového práva.

Jinými slovy, smysl směrnice zde není v tom, aby došlo k založení patentovatelnosti počítačových programů, ale zejména v tom, aby se odstranila dosud panující právní nejistota ve výše uvedeném smyslu. Každá směrnice (Directive) je závazná pro členské státy, kterým je určena, není však rozhodně závazná jako celek, ale pouze pokud jde o stanovený účel, resp. výsledek, kterého má být dosaženo, a tím je v našem případě právě výše zmíněné zakotvení právní jistoty. O formě a prostředcích transformace této směrnice do národních právního řádů pak budou rozhodovat orgány tohoto státu, přičemž hlavním principem zde bude, resp. by měla být zejména průhlednost a především právní jistota účastníků budoucích patentově-právních vztahů.

Závěrem

Nepřijetím této směrnice se toho tedy příliš mnoho nevyřeší, byť je samozřejmě otázkou, zda tomu opačně bude vůbec jinak. Pokud se podaří zachovat stávající text směrnice, a to alespoň tak, že zůstane její původní účel (tj. především zajištění požadavku právní jistoty), domnívám se, že může jít o věc prospěšnou. Důležitým ustanovením směrnice je výslovný zákaz jakýchkoliv přímých nároků na patentovatelnosti počítačového programu jako takového, a tedy ve svém důsledku minimalizovat riziko, že programátoři (či spíše tým patentových specialistů) budou muset rešeršovat tisíce softwarových patentů, aby zjistili, zda jimi právě vytvářený či vytvořený program některý z nich neporušuje. V tomto ohledu se domnívám, že směrnice svůj cíl poměrně plní, a že tedy nehrozí, aby programování v brzké době představovalo běh, resp. plazení se minovým polem pokrytým tisíci patenty.

Tolika můj názor na předmětnou směrnici. Stranou této úvahy však leží úvaha ještě jiná. Je vůbec správné umožnit patentovatelnost počítačových programů? Domnívám se, že nikoliv, avšak s tím už toho výše uvedená směrnice mnoho neudělá.

OpenOffice.org

Filip Molčan

E-mail: molcanf@openoffice.org

OpenOffice.org – největší Open Source projekt na světě

Kancelářský balík OpenOffice.org vyrostl za mnoho let vývoje do profesionální sady aplikací vhodných pro nasazení jak v domácím prostředí, tak ve velkých společnostech a institucích. V současné době se jedná o největší Open Source projekt na světě, který podporují přední IT společnosti. Základy OpenOffice.org jsou postaveny na jeho předchůdci – kancelářském balíku StarOffice, který se vyvíjel pro DOS, následně pro IBM OS/2 a v neposlední řadě i pro MS Windows. V roce 1999 koupil tento produkt Sun Microsystems a v roce 2000 založil projekt OpenOffice.org a položil základy největšímu Open Source projektu v dějinách.

Historie

13. 10. 2000	Sun Microsystems poskytuje zdrojové kody StarOffice projektu OpenOffice.org
1. 5. 2002	Vychází OpenOffice.org 1.0
15. 5. 2002	Vychází StarOffice 6.0
Leden 2005	Vychází OpenOffice.org 1.1.4
Červen 2005	Uvedení OpenOffice.org 2.0

Nyní je všechen vývoj kolem OpenOffice.org soustředěn do dokončované verze OpenOffice.org 2.0, která přináší mnoho nových unikátních vlastností, mimo jiné i nový formát souborů. Naplánováno je ještě vydání OpenOffice.org 1.1.5 – zde bude doplněna podpora nového formátu OpenDocument, aby byla zajištěna kompatibilita OpenOffice.org 1.x a 2.x.

Nejdůležitější údaje

- Softwarový balík se vším všudy pro začátečníky i zkušené uživatele.
- Pět nástrojů: CALC (tabulkový procesor); WRITER (textový procesor); DRAW (grafika) a IMPRESS (multimediální prezentace), BASE (databázový systém) a MATH (tvorba matematických výrazů).
- Svobodná licence znamená, že vaše kopie OpenOffice.org je zdarma k používání i distribuování. Navždy.
- Mezinárodní jazyková podpora včetně komplexních textů a jazyků se svislým psaním.
- Otevře i uloží běžně používané formáty souborů jako třeba Microsoft Office.
- Multiplaformnost: běží na Microsoft Windows 95/98/NT/Me/2000/XP, Linuxu, FreeBSD, Solaris a Mac OS X (pod X11).
- Nyní také exportuje do formátu PDF a do grafických souborů ve formátu .swf (Flash).
- Všechna vaše data jsou uložena v XML souborech – v otevřeném, do budoucna kompatibilním formátu. Tato funkce umožňuje další integraci OpenOffice.org do podnikových aplikací a IS.
- OpenOffice.org je doporučován přímo Evropskou Unií, jeho nasazováním se zabývají projekty COSPA (Consortium for Open Source in the Public Administration), eEurope a další.
- OpenOffice.org je dostupný ve více než 40 jazykových mutacích.
- Podpora LDAP umožňuje integraci OpenOffice.org do rozsáhlých systémů.
- Nový formát byl přijat skupinou OASIS Open Consortium a pravděpodobně bude přijat jako norma ISO, nový formát tak bude akceptován i vývojáří ostatních kancelářských i jiných aplikací.
- Vylepšená podpora pro import/export dokumentů MS Office 97–2003, podpora otevírání zaheslovaných dokumentů, podpora formátu WordML, SpreadsheetML, WordPerfect, Lotus 1–2–3.

OpenOffice.org – komunita

Kolem celého projektu vznikla obrovská komunita vývojářů z řad studentů, nadšenců, ale i velkých softwarových společností. Hlavní vývoj celého kancelářského balíku udržuje a koordinuje společnost Sun Microsystems ve spolupráci s dalšími světovými lídry na trhu IT a Open Source. Zde je několik fakt o komunitě kolem OpenOffice.org:

- Přes 140 000 zaregistrovaných členů komunity.
- Více než 4 000 zpráv zaslaných do diskuzních fór každý týden.
- Více než 400 jednotlivců i společností podepsalo JCA (Joint Copyright Assignment).
- Staženo 30 miliónů kopií OpenOffice.org.
- Více než 50 jednotlivých projektů.
- Lokalizace do 40 světových jazyků.

Specifika českého lokalizačního projektu

- Po anglické verzi první verze na světě, která má přeložené kompletní uživatelské rozhraní i nápovědu.
- Build systém českého lokalizačního týmu podporuje nejvíce operačních systémů vůbec.
- Build systém českého lokalizačního týmu využívají i další státy.

OpenOffice.org – vývoj

Jak již bylo uvedeno výše, sponzorem, organizátorem a koordinátorem celého projektu je společnost Sun Microsystems. Projekt je řízen prostřednictvím aplikace CollabNet, která umožňuje vývojářům společnou práci na jednotlivých projektech – přístup do CVS, správu verzí, chyb, diskuzní fóra, mailing-listy atp. Většina zdrojového kódu je psána v C++, v menší míře je potom použita Java. V současné době je OpenOffice.org nativně podporován na platformách Linux, MS Windows, Solaris (SPARC, x86), BSD a MacOS X (vyžaduje X11). Některé jednotlivé verze mají svá specifika, např. pro Linux jsou dostupná rozšíření pro lepší integraci OpenOffice.org do prostředí KDE či GNOME, pro MacOS X existuje úplně oddělený projekt NeoOffice/J, který se zaměřuje na vývoj OpenOffice.org bez nutnosti používat X11 server a lepší integraci s prostředím Aqua. Existuje také verze OpenOffice.org pro operační systém IBM OS/2 (komerční) a jsou dostupné i verze pro LinuxPPC, Linux/ARM, Linux/SPARC, IRIX, Tru64, NetBSD/SPARC, OpenVMS, BeOS či AIX.

OpenOffice.org a nasazení ve veřejné správě a velkých společnostech

Již několik let tisíce společností i institucí z celého světa nasazují kancelářský balík OpenOffice.org ve svých informačních infrastrukturách. Využívání OpenOffice.org přímo doporučují orgány Evropské Unie, která podporuje používání Open Source softwaru jak z důvodů možných značných úspor, tak z důvodů otevřenosti i lepší bezpečnosti.

Bylo provedeno mnoho výzkumů a vypracovány desítky studií zabývajících se využíváním OpenOffice.org ve veřejné správě. V současné době vzniká i v České republice projekt na podporu a rozšiřování Open Source ve veřejné správě, který podporuje přímo Ministerstvo informatiky ČR. Cílem tohoto projektu je pomoci veřejné správě a školství při nasazování OpenOffice.org a dalšího OSS. Veřejná správa tak bude pravděpodobně do budoucna přímo nucena ze strany EU do nasazování Open Source ve větší míře.

V našich podmínkách byla dlouho překážkou absence komerčních subjektů a jejich nabídky služeb, které by umožnili zájemcům o OpenOffice.org plynulý přechod, školení, integraci do jejich prostředí a technickou podporu či konzultace. V poslední době se ale této problematice v ČR věnuje několik menších společností i největší hráči v oboru IT a Open Source, pro firmy a instituce je tak přechod snadnější a bezpečnější, protože se mají v nouzi na koho obrátit.

V České republice je hlavním důvodem nasazování OpenOffice.org možnost úspory při pořizování licencí pro komerční konkurenční software. Uživatelé si však začínají uvědomovat i další výhody, které v sobě OpenOffice.org skrývá – především podporu otevřených formátů, výbornou podporu PDF, XML jako formát aplikace, velké možnosti při dalších úpravách OpenOffice.org – podpora různých programovacích jazyků – lepší bezpečnost (makroviry pro OpenOffice.org zatím stále neexistují) a podporu mnoha operačních systémů, která jim umožňuje nasazovat tento kancelářský balík i v heterogenních prostředích.

I přes řadu problémů, které jsou způsobeny zvláštností trhu v ČR a téměř slepou rozšířeností produktů společnosti Microsoft, společností přecházejích na OpenOffice.org přibývá, přičemže se často jedná i o velké organizace, pro které je úspora na licenčních poplatcích opravdu nepřehlédnutelná.

OSS Alliance a OpenOffice.org

Společnost pro výzkum a podporu Open Source (nebo také OSS Alliance), která v ČR vznikla, si klade za cíl mimo jiné rozšířit využívání OSS ve veřejné správě a školství. Jedním z nejdůležitějších projektů je podpora právě kancelářského balíku OpenOffice.org. V první fázi projektu se bude odborná skupina snažit přesně definovat nejdůlěžitější body přechodu k OpenOffice.org a úskalí především ze

strany ostatních proprietárních aplikací, které se ve veřejné správě používají. Právě množství v současné době používaného proprietárního softwaru je tou největší překážkou při nasazování OpenOffice.org ve veřejné správě a školství.

Informační zdroje o OpenOffice.org

http://cs.openoffice.org	– Oficiální stránky českého lokalizačního týmu OpenOffice.org
http://OO.o.cz	– Oficiální portál pro uživatele OpenOffice.org
http://www.openoffice.cz	– Komerční portál o OpenOffice.org
http://www.openoffice.org	– Oficiální mezinárodní stránky projektu

Do roka a do dne, aneb roční zkušenosti s implementací univerzitního portálu

Jan Valdman

E-mail: valdman@civ.zcu.cz

Abstract

This paper deals with some experience we have got during implementation of IBM WebSphere Portal at the University of West Bohemia. We realized that university portal projects in academic environment meet different requirements and have to fulfill other demands compared to enterprise portals.

In the paper, we concentrate on user–related and operation–relates issues rather on technical ones.

1 Úvodem

V tomto článku se pokusíme nastínit některá naše zjištění a zkušenosti, které jsme získali během prvního roku budování a zkušebního provozu portálu v technologii IBM WebSphere. Nechceme se však omezovat na jeden produkt konkrétního výrobce, proto se budeme věnovat spíše obecným aspektům zavádění portálu ve světle našich konkrétních zkušeností.

Portál je běh na dlouhou trať, jeho implmentace není záležitost několika měsíců. Podobně jako nelze „nainstalovat intranet“ nebo „nainstalovat web“, nelze ani „naimplmentovat portál“. Věc žádá svůj čas a vlastní tempo změn v organizaci, jak ostatně dokládá i následující citát:

> *„My department takes care of univerzity network, computing, services, web... and portal... if we ever have any...“*
>
> *IT ředitel nejmenované britské univerzity*

2 Portál není web

Jedno z prvních rozčarování nastává v situaci, kdy uživatelé od portálu očekávávají všechny vlastnosti webu a k tomu „něco navíc“. Přestože portál přináší

mnoho nového, často se jedná o kompromisy (technické i věcné) na úkor čistého webového řešení resp. ortodoxních webových zásad.[1]

Proto se stavíme za myšlenku, že „portál není web“ – tím tedy myslíme portál druhé/třetí generace, který integruje aplikace. V takovém portálu spolu portlety komunikují pomocí zasílání zpráv, což má ke klasickému pojetí hypertextu poměrně daleko. S tím souvisí problematika odkazů, které portál dynamicky generuje, a které jsou obvykle poměrně ošklivé a dlouhé. Uživatele očekávající „webové“ chování to pak zbytečně dráždí, neboť očekávají přátelská URL, která jsou stabilní v čase.

Samotné portlety jsou potom jistě více než pouhé vizuální kousky stránky (snippets), které se „cut&paste“ vloží do stránky. Portlet je potřeba navrhnout tak, aby jeho design zapadl do designu portálové stránky, která může mít různé motivy vzhledu, jež musí portlety alespoň částečně přebírat.

Portálová stránka je složena z portletů, které na stránku de facto umisťuje sám uživatel. Portálový server navíc může ještě nějaké portlety skrýt v závislosti na přístupových oprávněních konkrétního uživatele. Tím vzniká situace, že ani autor stránky prakticky neví, jak bude vypadat stránka vyrendrovaná pro uživatele. V extrémní situaci ani nevíme, zda bude portlet zobrazen v jednosloupcové nebo vícesloupcové sazbě, natož abychom se zajímali o rozlišení použitého www prohlížeče. Z toho je patrné, že o nějakém precizním výtvarnu nebo optimalizaci stránky nelze příliš mluvit. Portlet je vyrendrován na portálové stránce a prostě musí počítat s tím, že to může být na šířku 1 600 pixelů a nebo také jen na 200 px. . .

3 Stabilita portálové infrastruktury

Provozujeme IBM WebSphere Portal verze 5 v prostředí OS Linux. Náš portál využívá databázový backend Oracle a adresářové služby OpenLDAP využívající autentizaci Kerberos. Po vyřešení problémů instalace a napojení na stávající prostředí můžeme konstatovat, že portál běží stabilně. V současné době provozujeme systém několik měsíců v pilotním provozu pro cca dvě stovky tzv. pravidelných uživatelů.

Za dobu pilotního provozu jsme detekovali pouze jedenkrát, že byl portál v „podivném“ stavu a bylo potřeba jej restartovat. Portál opakovaně úspěšně přestál drobné výpadky LDAP, které se projevují tak, že se noví uživatelé nemohou přihlásit. Dvakrát došlo k výpadku databázového serveru, který ovšem portál neustál bez restartu. V jednom případě došlo k zaplnění disku databá-

[1] Například na http://www.pooh.cz/pooh/a.asp?a=2010377 se dočteme, že portál státní správy (který stál daňové poplatníky přes 20 mil. korun) porušuje většinu zásad přístupného webu, které si Ministerstvo informatiky nechalo (jistě za peníze daňových poplatníků) vypracovat. . .

zového serveru a portál (filesystém a databázi) bylo dokonce potřeba obnovit ze zálohy. Můžeme tedy konstatovat, že portál je mnohem citlivější na výpadky databáze než LDAP, ale to není vzhledem k odlišnému použití zmíněných služeb překvapivé.

4 Funkce (portlety) je potřeba pracně dodělat

IBM WebSphere Portal je produkt spadající více do kategorie APS (Application Platform Suite) než SES (Smart Enterprise Suite), což v praxi znamená, že je více infrastrukturou pro provoz a integraci aplikací než podnikový informační systém s mnoha funkcemi.

Uživatelé však chtějí funkce a ty je potřeba (v souladu s naší původní představou) doprogramovat formou portletů. Jedním z cílů pilotního provozu portálu na naší univerzitě je vlastně zjistit, jaké funkce budou uživatelé v portálu potřebovat a alespoň prototypově je naprogramovat. Takových potřeb/požadavků vzniklo poměrně hodně, protože portály jsou zaměřené na podnikovou klientelu a nikoli na akademické prostředí, čemuž odpovídá skladba dodávaných portletů.

V případě IBM WebSphere Portal jsme postrádali i některé základní funkce, jako je například sitemap, přihlašovací portlet, jednoduchý content management a podobně. V některých případech jejich absenci nerozumíme, v ostatních jsme se záměrně vyhýbali zprovozňování dalších složitých komponent/technologii (například WPCP).

Dle našeho odhadu je potřeba pro (každý) portál cca 20 až 40 specifických portletů, které je potřeba naprogramovat. Budeme-li počítat na vytvoření jednoho portletu člověkotýden práce, lze snadno odvodit čas potřebný na implementaci portálu.

V podmínkách ZČU je prioritou integrace (resp. přepsání) studijního systému IS/STAG, která nám obsadila většinu lidských zdrojů. Díky tomu se trvale potýkáme s nedostatkem programátorské kapacity. Povedlo se nám sice do projektu zapojit několik studentů, ale jejich zapracování trvá vždy několik měsíců. Nezanedbatelný čas zabere také jejich zaškolení, řízení a „učesávání“ výsledků jejich práce. Studenti se jako každý začínající programátor dopouštějí z nezkušenosti různých odchylek od požadovaného výsledku, takže vzniká potřeba dodělávek, předělávek a „evangelizace“ zdrojových kódů.

V našem konkrétním případě jsme také ztratili spoustu času přechodem z vývojového prostředí IBM WebSphere Studio Application Developer 5 na IBM Rational Appliaction Developer 6. Za pochodu jsme se museli neustále zkoumat různé záludnosti J2EE a nekompatibilitu mezi zmíněnými produkty.

Portlet je vlastně samostatná aplikace, která musí být dodělaná po všech stránkách. Portál sice programátorovi ušetří dost práce v oblasti „vnitřní infrastruktury aplikace“ (autentizace, autorizace, navigace,...), ale na druhou

stranu je každý portlet samostatnou webovou aplikací se všemi náležitostmi: nápovědou, podporou více jazyků, napojením na datové zdroje...

Náš názor je asi takový, že to co ušetříte na infrastruktuře musíte vrátit na „propracování" portletu. Celkově je výroba kvalitních (rovná se dobře přenositelných) portletů poměrně náročná a zdlouhavá.

5 Integrace apliakcí

Portál chápeme jako místo pro integraci (podnikových) aplikací. Je obecně známo, že integraci lze dělat několika základními způsoby: na datové vrstvě, na aplikační úrovni nebo na prezentační vrstvě.

V našem projektu jsme integrovali téměř výhradně na datové vrstvě. Stávající (legacy) aplikace obvykle používají relační databáze, kde standardy SQL a JDBC dávají dobrý základ pro snadné propojení aplikací. Portál v drtivém případě pouze čte z databází, takže odpadají problémy s paralelním přístupem k datům, nicméně nevýhoda duplikování aplikační logiky a případně jejího zpětného inženýringu z původní aplikace zůstává.

Integrace na aplikační úrovni je ze systémového hlediska mnohem lepší, nicméně zatím jsme nepotkali (starou) aplikaci, která by byla napsána vícevrstevně a nabízela nějaké použitelné API.

Integraci na prezentační vrstvě dělá sám portál. Portál umožňuje na jednu stránku umístit spolupracující portlety, z nichž každý využívá jinou technologii a odlišný backend systém. Z pohledu uživatele však portlety k sobě logicky patří, a uživatel vlastně vůbec netuší, že pracuje se dvěma nezávislými aplikacemi namísto jedné.

V budoucnu se chceme soustředit na využívání technologie webservices a také na větší využívání XML resp. XSLT pro přímé propojení systémů s portálem na aplikační vrstvě. Problémem však zůstává, zda stávající legacy aplikace dokáží pomocí zmíněných technologií poskytovat data. Je-li potřeba kvůli webservices nebo XML výstupu duplikovat logiku aplikace, potom je ke zvážení, zda je lepší

- využít technologii aplikace pro vytvoření výstupu, který lze snadno použít v portálu nebo
- využít Java technologii portálu a na aplikaci se napojit na datové vrstvě.

Tím se dostáváme k dalšímu aspektu implementace portálu: v portálu je potřeba mít integrovány aplikace, což mnohdy znamená doprogramování samotné aplikace. Z hlediska portálového projektu tedy vyvstává otázka, kde je hranice portálu. Je doprogramování aplikace ještě implementací portálu nebo je to běžná práce aplikačního programátora?

Dalším problémem je testování portálu z pohledu uživatele. Vzhledem k tomu, že portál integruje množství aplikací, že využívá vlastní sofistikovanou autorizační strukturu, a že (logicky) neobsahuje obdobu unixového příkazu `su`, nemá administrátor prakticky možnost ověřit, co vlastně konkrétní uživatel(é) v portálu uvidí nebo neuvidí, zda funguje správně napojení do všech backend systémů a podobně.

6 Správa obsahu

Naše zjištění: portál bez obsahu (tj. pouze s aplikacemi) je k ničemu. Potřebujete nějak začlenit doprovodné texty (jako části stránek) a vystavovat dokumenty. Tyto dvě oblasti je potřeba posuzovat odděleně a najít pro ně nějaká jednoduchá řešení. V našem případě jsme byli nuceni zavrhnout dodávanou komponentu IBM Portal Document Manager (PDM) a pořídit za ni náhradu.

V oblasti publikování obsahu (web content management) jsme zkoumali technologii WebSphere Content Publishing (WPCP), kterou jsme ale nakonec také zavrhli pro přílišnou technickou i uživatelskou složitost. V současnosti používáme vlastní jednoduché řešení (portlet) ve stylu „simple things do work" postavené na bázi HTMLarea. Tento stav nám víceméně vyhovuje resp. bude muset vyhovovat ještě nějaký čas – dokud si nevyrobíme něco lepšího nebo do přechodu na WebSphere portál verze 5.1, který je po stránce správy obsahu výrazně lépe vybaven.

Opět se prokázalo tvrzení, že dobrý portálový projekt odhalí všechny problémy v organizaci. V našem případě se potvrdilo, že žádný web content management v organizaci nemáme, a proto jsme se jej rozhodli v rámci portálového projektu raději nezavádět.

7 Provoz portálu

Z provozního pohledu stojí portálový server na vrcholu pyramidy komponent a technologií IT prostředí organizace. Výpadek některé komponenty potom může omezit nebo vyloučit provoz portálu. V lepším případě přestanou fungovat jen některé portlety, v horším se třeba nejde do portálu přihlásit nebo přidávat nový obsah.

V praxi se ukázalo, že intenzivní používání vyrovnávacích pamětí je dvojsečné. Některé problémy se totiž projevují se zpožděním až desítek minut. Vyrovnávací paměť (tzv. dynacache) nepracuje s celými portlety (stránkami), ale uchovává odděleně jejich části. V případě závady se potom může stát, že na stránkách postupně přestávají fungovat nebo mizí některé ovládací prvky tak, jak (zřejmě) expirují v cache. Z hlediska uživatele se pak portál chová velmi podivně.

Dále můžeme potvrdit radu, kterou jsme dostali na jedné IBM konferenci: nejdůležitější schopností administrátora portálu je ukazovat prstem: „to není portálový problém, to je * problém“, kde za hvězdičku doplňte síťový/databázový /LDAP/... Většinu poznámek „nejde ti portál“ lze opravdu řešit pouze ukázáním prstem na příslušného kolegu, aby odstranil problém, který je někde zcela mimo portál.

8 Závěr

Portál přestavuje po stránce provozní i uživatelské nový prvek v IT prostředí, který vyžaduje změnu v myšlení a návycích jak uživatelů, tak i vývojářů a editorů (poskytovatelů) obsahu. Myslíme si, že portál se bude podobně jako web v organizaci několik let postupně etablovat, než si najde „své místo“ a plně se projeví jeho přínos. Tento proces ale bohužel neumíme urychlit, k čemuž přispívají ještě mladé a nevyzrálé portálové produkty.

Literatura

[1] Valdman, J., Rychlík, J.: *Úloha portálu v projektu eUNIVERZITA Západočeské univerzity v Plzni.* Příspěvek na konferenci UNINFOS'04. STUBA, Bratislava 2004.

[2] Šimonek, J.: *Vývoj portletů pomocí WSAD+PT.* Příspěvek na setkání portálové skupiny Cesnet. Pardubice 2004.

[3] Valdman, J.: *Co je to portál? Výběr portálového projektu a implementace portálu ZČU.* Příspěvek na XXIV. konferenci Europen.CZ. Kralický Sněžník 2004.

[4] Otta, M.: *Open Source portály.* Příspěvek na XXIV.konferenci Europen.CZ. Kralický Sněžník 2004.

[5] Gallistl, Z.: *Vývoj aplikací pro WebSphere Portal.* Diplomová práce obhájená na Katedře informatiky a výpočetní techniky, FAV ZČU. Vedoucí Jan Valdman, oponent Jiří Šimonek. Plzeň 2004.

[6] Halas, P.: *Katedrální portál v technologii WebSphere.* Diplomová práce obhájená na Katedře informatiky a výpočetní techniky, FAV ZČU. Vedoucí Přemysl Brada, oponent Jan Valdman. Plzeň 2004.

[7] Šimonek, J.: *Portál Ostravské university.* Skripta pro distanční vzdělávání. OSU, Ostrava 2004.

[8] Valdman, J.: *Seznámení s portálem ZČU.* Univerzitní Noviny, č. 1–XI, str. 10–11. Západočeská univerzita v Plzni. Plzeň 2005.

Testování a ladění výkonnosti J2EE aplikací pro Websphere Portal

Daniel Holešínský, Robert Bohoněk, Jiří Šimonek

E-mail: daniel.holesinsky@osu.cz, robert.bohonek@osu.cz, jiri.simonek@osu.cz

1 Úvod

1.1 Stručný přehled stavu aplikací na Ostravské univerzitě

Před zavedením Websphere portálu byly jednotlivé aplikace a technologie na Ostravské univerzitě značně roztříštěné. Používaly se následující technologie: aplikační server Jboss, servletový kontejner Tomcat ve spojení s webovým serverem Apache. Každá aplikace měla svůj vlastní způsob autentizace a autorizace (LDAP, databáze). Po úspěšném nainstalování Websphere portálu se na Ostravské univerzitě zformoval vývojový tým s úkolem převést většinu aplikací pod portál. Jako hlavní výhoda použití portálu se jevila možnost využití mechanismu SSO (Single sign-on) pro jednotné přihlášení uživatelů a individuální nastavování přístupu k aplikacím. Bylo tedy nutné navrhnout mechanismus pro integraci těchto aplikací. Při navrhování mechanismu pro integraci těchto aplikací se ukázal jako problém paměťové nároky jednotlivých aplikací. Bylo tedy nutné vytvořit metodiku a postupy pro vyřešení těchto problému.

2 Testování využití zdrojů

2.1 Testování bez uživatelů

Toto testování musí předcházet testování aplikace na Websphere portálu. Jeho účelem je navržení sady testů odpovídajících dané aplikaci (jako referenci bereme 20 uživatelů – 20 vláken s různými úkoly a různou dobou spouštění. Sledují a logují se časy potřebné pro vykonání operací, dále se sleduje využití zdrojů. Výsledek – vyladění indexování, optimalizace databázových dotazů ve spolupráci se správci databázových systémů).

2.2 Testování s uživateli

Toto testování předchází ostrému nasazení aplikace. V našem případě se vyhlásí školení uživatelů pro tuto aplikaci. Snažíme se taktéž dodržet maximální počet uživatelů na 20 ve skupině. Díky předchozímu testování by již mělo být vyladěno používání zdrojů aplikace, soustředíme se tedy na pozorování uživatelů a odhadování kritických míst, které vznikají díky nepředvídatelnému chování uživatelů.

2.3 Testování webového rozhraní

Na první pohled by se mohlo zdát, že na vykreslování html stránky není co vylepšovat. Opak je pravdou (důraz je kladen na snadnou čitelnost kódu, je nutná spolupráce s našimi web designéry). Na této straně jsme se rozhodli odstranit přímý zápis řetězce html kódu do OutputStream portletu a nahradit jej jsp stránkami (důvodem byla čitelnost kódu v jsp stránce oproti zápisu html do řetězce).

Další tipy:

- Pro usnadnění vývoje webových aplikací použití frameworku Struts
- Pro dynamický výpis informací na stránce použití JSTL
- Při skládání jsp stránky z více jsp stránek použití direktivy `<%@ include file="test.jsp" %>` namísto akce `<jsp:include page="test.jsp" />` (direktiva include je akci překladače – obsah je vložen do zkompilované jsp stránky jako řetězec [JGURU00])
- Minimalizace rozsahu proměnných (scope). Programátor nebo designér implicitně určí kde se daná proměnná nachází (requestScope, sessionScope) – minimalizuje se čas potřebný pro nalezení této proměnné.

2.4 Kultura programování alias dobrý návrh nadevše

V této oblasti klademe velký důraz na používání návrhových vzorů (minimálně tyto: SessionFacade, Data Access Object, Data Transfer Object [J2EEDP02]).

Klíčový je také výběr podpůrných technologií/frameworků. V této oblasti se na OU ustálily následující techniky:

- Datová vrstva – EJB 2.0 [EJB04], Hibernate [HIB05]
- Webový framework Struts [STR05], experimentujeme s JSF [JSF05] (problémem je, že je nelze použít přímo, ale je třeba využívat IBM úpravu pro portál)

- Pro build aplikací používáme nástroje vývojového prostředí WSAD/RAD – naší snahou hlavně pro případ týmového vývoje je požívat kombinaci Maven [MAVEN05], Ant [ANT05]
- Pro vývoj aplikací používáme vlastní metodiku založenou na Unified Process [UP05]
- Pro dokumentování vývoje a tvorbu uživatelské dokumentace používáme Apache Forrest [FORREST05]
- Ve fázi testování je práce s j2ee frameworkem Spring [SPRING05]

3 Určení problematických oblastí

3.1 Analýza přístupu k aplikacím na portálu

Pro analýzu přístupu na portál používáme nástroj Tivoli Web Site Analyzer [TIVOLI05]. Tento nástroj nám umožňuje vytvořit grafický výstup na základě logovacího souboru portálu. Tento log je ve formátu NCSA Combined [NCSA] (obr. 1).

```
195.113.106.139 - Holesins [29/Apr/2005:07:28:58 +0000] "GET
/Portlet/5_0_I2N/Generování_protokolu_o_přijímacím_řízení_-_FFI?PortletPID=5_0_I2
N&PortletMode=View&PortletState=Normal HTTP/1.1" 200 -1
"http://cvportal.osu.cz/Page/6_0_20S/holesins" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.0)" "JSESSIONID=JFRfkrnmTVlXtftiNIBqEaG"
```

Obr. 1 Formát logu NCSA Combined

Site Analyzer tento soubor analyzuje a výsledky uloží do své databáze. Z těchto výsledků se poté vytvářejí jednotlivé reporty. V našem případě nás zajímá počet přístupů ke stránce portletu a k portálu (obr. 3) a přístupy jednotlivých uživatelů (obr. 2).

Na základě reportů stanovujeme priority sledování jednotlivých aplikací.

3.2 Analýza provozu garbage collectoru

Další oblastí pro analýzu provozu portálu je sledování provozu garbage collectoru. V případě, že necháme vypisovat provoz garbage collectoru do souboru, dostaneme následující výpis:

Co tento výpis znamená:

1. Požadavek na paměťový prostor 3 080 208 byte. Od posledního požadavku uplynulo 938 s.
2. Start garbage collectoru.

Portal Report

Report Date: April 21, 2005 7:14:00 AM GMT
Report Range: April 19, 2005 12:00:00 AM GMT - April 20, 2005 11:59:59 PM GMT

Portal Server Login Trend

Displays the Portal Server logins over time.

Portal Server Login by User Ranking

Displays a ranking of the Users who access your site using the Portal Server Login command.

Resource Name	Hits	% of Total
wpsadmin	32	0.04%
simonek	27	0.03%
RBohone1	15	0.02%
MRiedlo1	14	0.02%
Krkosko	8	0.01%
Horakova	8	0.01%
Solonko	8	0.01%
JBlazek1	8	0.01%
F02238	7	0.01%
F04724	6	0.01%
R03230	6	0.01%
dipl_KMS	6	0.01%
Spasova	6	0.01%
R04519	5	0.01%
Kovarova	5	0.01%
Menza	4	0.01%
D02205	4	0.01%

Obr. 2 Přehled uživatelů portálu

Portal Server Portlet Ranking

Displays a ranking of the Portal Server Portlets viewed by visitors to your site.

Resource Name	Hits	% of Total
/Portlet/5_0_2RO/Prihlaseni	9951	12.64%
/Portlet/5_0_5BV/ZCU_Static_Text_portlet_(concrete).$cloned.3_0_3GH	9946	12.64%
/Portlet/5_0_5BT/ZCU_Static_Text_portlet_(concrete)	9937	12.62%
/Portlet/5_0_4VI/ZkouskyStudenta	950	1.21%
/Portlet/5_0_5L9/StudentInfo	939	1.19%
/Portlet/5_0_4VG/ZnamkyStudenta	936	1.19%
/Portlet/5_0_4VH/RozvrhStudenta	933	1.19%
/Portlet/5_0_4VK/IS/STAG_-_Informace_o_predmetu	931	1.18%
/Portlet/5_0_2RT/NewsBoard_portlet	902	1.15%
/Portlet/5_0_4IU/Sitemap_Portlet	878	1.12%
/Portlet/5_0_2RV/StudentView_portlet	619	0.79%
/Portlet/5_0_5LA/Knos_preview	564	0.72%
/Portlet/5_0_55O/FileExplorer_portlet.$cloned.3_0_3A9	481	0.61%
/Portlet/5_0_341/Dipl_Projekt_portlet	448	0.57%
/Portlet/5_0_376/EmailSupport_portlet	367	0.47%
/Portlet/5_0_5OP/HelpDesk_portlet	318	0.40%
/Portlet/5_0_5OR/HelpDesk_Detail_portlet	315	0.40%
/Portlet/5_0_5OQ/HelpDesk_portlet_Konfigurace	315	0.40%
/Portlet/5_0_5F1/EvaResults_portlet	271	0.34%
/Portlet/5_0_52J/ZCU_StaticText_portlet.$cloned.3_0_37C	263	0.33%
/Portlet/5_0_3AA/StudentInfo	262	0.33%
/Portlet/5_0_3GH/SystÃ©m_akteditace_OU	259	0.33%
/Portlet/5_0_30T/JÃdelnÃÄ□ek	247	0.31%
/Portlet/5_0_2S2/Pohled_Magion_portlet	241	0.31%
/Portlet/5_0_536/IS/STAG_-_Rozvrh_predmetu	196	0.25%
/Portlet/5_0_534/IS/STAG_-_Informace_o_predmetu	196	0.25%

Obr. 3 Přístup k jednotlivým aplikacím (portletům)

```
<AF[56]: Allocation Failure. need 3080208 bytes, 938830 ms since last AF>      1
<AF[56]: managing allocation failure, action=1 (149212896/544704280) (18287376/28668648)>
<GC(56): GC cycle started Wed Apr  6 09:53:28 2005  2
<GC(56): freed 191342976 bytes, 62% free (358843248/573372928), in 1182 ms> 3
  <GC(56): mark: 240 ms, sweep: 36 ms, compact: 906 ms>
  <GC(56): refs: soft 0 (age >= 32), weak 1, final 1067, phantom 0> 4
  <GC(56): moved 531740 objects, 28874320 bytes, reason=1, used 5896 more bytes>
<AF[56]: completed in 1184 ms> 5

<AF[57]: Allocation Failure. need 12320784 bytes, 2075 ms since last AF>
<AF[57]: managing allocation failure, action=2 (301142920/573372928)>
<GC(57): GC cycle started Wed Apr  6 09:53:30 2005
<GC(57): freed 53484912 bytes, 61% free (354627832/573372928), in 269 ms>
  <GC(57): mark: 236 ms, sweep: 33 ms, compact: 0 ms>
  <GC(57): refs: soft 0 (age >= 32), weak 0, final 2, phantom 0>
<AF[57]: completed in 270 ms>
```

Obr. 4 Výpis garbage collectoru

3. Počet byte, které byly v rámci úklidu uvolněny. Hodnota udává, kolik byte je volných ze stávající velikosti haldy (358 843 248/573 372 928).
4. Na tomto řádku je zajímavá hodnota final 1 067 – tzn. tolik objektů čeká ve frontě pro vykonání operace finalize().

Jakým způsobem lze využít tyto informace?

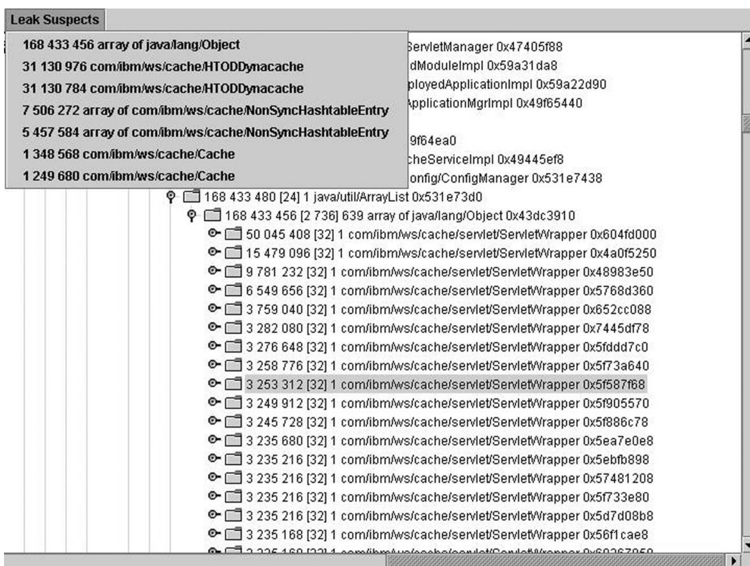
1. Napsat si program (skript) který na základě regulárních výrazů získá souhrny z tohoto logu. Jedná se nám o: celkovou dobu běhu JVM, celková doba běhu GC a velikost uvolněné paměti za tuto dobu (u Sun JVM lze sledovat i počet uklizených objektů).
2. [JVV03] udává (kapitola 2), že celkový podíl času stráveného čištěním paměti k celkové době běhu aplikace nesmí přesáhnout 15 %, v ideálním případě 5 %.
3. Zvážit použití metody finalize() u objektů [EJ01] rada č. 6.

3.3 Analýza stavu haldy

V případě, že dojde k nepopulární java.lang.OutOfMemoryException je možné nastavit JVM, aby v tomto případě provedla výpis obsahu haldy. Pro analýzu souboru, který vytváří Websphere portál lze použít nástroj pro grafické ztvárnění obsahu haldy. Nastavení portálu viz [OOM05], nástroj pro analýzu HeapAnalyzer viz [HEAP05]. Výpis haldy do souboru lze provést i příkazem `kill -3 pid_procesu` (pouze OS Linux).

Poté co HeapAnalyzer provede analýzu vstupního souboru je možné si nechat vykreslit strom objektů a nechat si vytipovat problémové oblasti pomocí volby compile leak suspects (obr. 5).

Taktéž je možno pomocí tohoto nástroje vyhledat pouze určité typy objektů. Víme-li, že webová aplikace je představována objektem com/../webapp/WebApp můžeme tyto objekty vyhledat – viz obr. 6.



Obr. 5 Zobrazení stromu objektů na haldě JVM

TotalSize/819 400 928 ▼	Size/...	Address	No.C...	Object(187 objects)
215 798 496	144	0x61198bb0	24	com/ibm/ws/webcontainer/webapp/WebApp
214 955 704	144	0x85cc1230	24	com/ibm/ws/webcontainer/webapp/WebApp
200 465 704	136	0x47117508	24	com/ibm/ws/webcontainer/webapp/WebApp
50 045 032	136	0x604e7c68	24	com/ibm/ws/webcontainer/webapp/WebApp
15 478 688	136	0x49fbd8e0	24	com/ibm/ws/webcontainer/webapp/WebApp
9 780 600	136	0x48795c30	24	com/ibm/ws/webcontainer/webapp/WebApp
6 545 928	144	0x4d115908	24	com/ibm/ws/webcontainer/webapp/WebApp
4 865 008	144	0x4e8384d0	24	com/ibm/ws/webcontainer/webapp/WebApp
3 755 512	144	0x53dcb9a8	24	com/ibm/ws/webcontainer/webapp/WebApp
3 308 336	144	0x57fb5028	24	com/ibm/ws/webcontainer/webapp/WebApp
3 278 312	144	0x4bf7e6f8	24	com/ibm/ws/webcontainer/webapp/WebApp
3 272 880	144	0x59d2e600	24	com/ibm/ws/webcontainer/webapp/WebApp
3 255 008	144	0x5acf76e0	24	com/ibm/ws/webcontainer/webapp/WebApp
3 254 520	144	0x4f44a560	24	com/ibm/ws/webcontainer/webapp/WebApp
3 249 544	144	0x57956a90	24	com/ibm/ws/webcontainer/webapp/WebApp
3 246 384	144	0x57bf0358	24	com/ibm/ws/webcontainer/webapp/WebApp
3 241 960	144	0x5a8a2a20	24	com/ibm/ws/webcontainer/webapp/WebApp

Obr. 6 Vyhledání všech objektů typu WebApp na haldě

V případě výjimky OutOfMemoryException je tedy poměrně jednoduché diagnostikovat „viníka“ problému.

4 Nastínění scénáře pro nastavení aplikací a aplikačního serveru

Před započetím jakékoliv optimalizace je třeba vzít v potaz pořekadlo „neopravujte to, co opravit nepotřebujete“ (viz úvod [JVV03], rada 37 [EJ01]).

4.1 Nastavení JVM

Je-li to možné, přejít na vyšší verzi JVM (je-li k dispozici a je-li možný přechod). Taktéž je-li to možné vyzkoušet více implementací JVM (minimálně si provést porovnání IBM/SUN viz [MAK]).

4.2 Nastavení haldy

Pro nastavení velikosti haldy se používají dva hlavní parametry –Xmx a Xms (jejich syntaxe může záležet na implementaci JVM od různých dodavatelů). S ohledem na naši hardwarovou konfiguraci bylo pro nastaveni parametru Xmx použita maximální možná velikost haldy 1 900 MB (limitace pro 32bit operační systémy viz [DIS04]). Obecné doporučení je udržovat velikost haldy pod hranicí fyzické paměti počítače (s ohledem na velikost paměti které je potřeba pro další procesy spuštěné paralelně s JVM). Minimální velikost haldy nastavená pomocí parametru Xms je nastavena na základě analýzy provozu GC na hodnotu 512 MB (viz [GC03]). Dále se může měnit algoritmus pro zvětšování haldy (bude rozvedeno během přednášky).

4.3 Nastavení aplikací

V okamžiku, kdy se aplikace testuje s uživateli je užitečné si nechat vypsat výpis obsahu haldy pro další analýzu. I u dobře navržené aplikace můžou během analýzy vyjít najevo objekty typu Collection popř. Map sloužící jako cache označené klíčovým slovem static. V takovém případě je na zvážení, zda nezapátrat po nějaké open source implementaci cache jako např. OSCache viz [CACHE]. Jestliže není nutné mít k dispozici takový nástroj je vhodné alespoň zvážit jako cache implementaci WeakHashMap.

Některé aspekty, které je třeba vzít v úvahu při optimalizaci aplikace:

- reakce operace by neměla být delší než 2 s (viděno z pozice uživatele). Jestliže některá operace má delší reakci než 10s je dobré jej informovat o postupu práce kupříkladu indikátorem průběhu činnosti (progress bar)
- uživatelé neradi (v webovém rozhraní) používají vámi definované tlačítko zpět (raději používají tlačítko zpět v html klientovi) – jestliže je na tuto akci navázán např. úklid v session, tak nemusí proběhnout
- používání vláken pro oddělení časově náročných funkcí
- částečné zobrazení dat – nečekat na kompletní dodání všech dat
- napsat si sadu testů – tyto testy spouštět po každém refaktorování popř. optimalizaci
- definování jasných cílů ladění výkonu

5 Závěr

V tomto rychlém přehledu způsobů analýzy problému, testování a ladění výkonnosti J2EE aplikací jsme se seznámili se technologiemi a postupy, které nám můžou výše uvedené procesy usnadnit. K jednotlivým oddílům tohoto přehledu se vrátíme a rozebereme během prezentace na EurOpen 2005.

Literatura

[RED03] IBM redbooks: *Portal Application Design and Development Guidelines.* http://www.redbooks.ibm.com/redpapers/pdfs/redp3829.pdf

[DIS04] TSS FORUM: *How to avoid 2 GB memory limit of JVM in Linux.* http://www.theserverside.com/discussions/thread.tss?thread_id=26347

[GC03] *Fine-tuning Java garbage collection performance.* http://www-106.ibm.com/developerworks/ibm/library/i-gctroub/

[RED05] IBM redbooks: *IBM WebSphere Portal for Multiplatforms platforms V5.1 Handbook.* http://www.ibm.com/redbooks

[JVV03] Shirazi, J.: *Java vylaďování výkonu.* Grada Publishing, a. s., 2003.

[JGURU00] JGURU: *What is the difference between* `<%@ include file="abc.jsp" %>` *and* `<jsp:include page="abc.jsp" %>`. http://www.jguru.com/faq/view.jsp?EID=13517

[J2EEDP02] *Core J2EE Patterns.* http://java.sun.com/blueprints/corej2eepatterns/Patterns

[EJB04] Sun: *Enterprise JavaBeans Technology.* http://java.sun.com/products/ejb

[HIB05] Hibernate: *Relational Persistence For Idiomatic Java.* http://www.hibernate.org

[STR05] Jakarta Struts: *Welcome to Struts.* http://struts.apache.org

[JSF05] Sun: *JavaServer Faces.* http://java.sun.com/j2ee/javaserverfaces

[SPRING05] Spring: *java/j2ee application framework.* http://www.springframework.org

[MAVEN05] *Apache maven project: Welcome to Maven.* http://maven.apache.org

[ANT05] *Apache ant: Welcome to ant.* http://ant.apache.org

[UP05] IBM: *Rational Unified Process.*
http://www-306.ibm.com/software/awdtools/rup

[FORREST05] Apache forrest project: *Welcome to Apache Forrest.*
http://forrest.apache.org/0.6/index.html

[NCSA] IBM: *Log File Formats.*
http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA_info45/en_US/HTML/guide/c-logs.html#combined

[EJ01] Bloch, J.: *Effective Java: Programming Language Guide.* Addison Wesley, 2001.

[MAK] Kuba, M.: *Speed of Java on different platforms.*
http://www.ics.muni.cz/~makub/java/speed.html

[CACHE] *OpenSymphony: OsCache.* http://www.opensymphony.com/oscache

[WEAK] *OnJava: The WeakHashMap Class.*
http://www.onjava.com/pub/a/onjava/2001/07/09/optimization.html

[OOM05] IBM: *Out of Memory errors on Windows.* Part 2b – Heap Leak,
http://www-1.ibm.com/support/docview.wss?uid=swg21140641

[HEAP05] IBM: *HeapAnalyzer.* http://www.alphaworks.ibm.com/tech/heapanalyzer

Generování zdrojového kódu – pomůcka, nebo překážka?

Oldřich Nouza

E-mail: nouza@km1.fjfi.cvut.cz, oldrich.nouza@unicorn.cz

Abstrakt

Nástroje CASE pro analýzu a návrh informačních systémů v jazyce UML jsou v současné době samozřejmostí. Mnohé z nich podporují tzv. generování zdrojového kódu z modelu, za účelem snazšího přechodu od fáze návrhu k fázi implementace. Znamená však generování kódu vždy ulehčení práce? Pokud ne, kde hledat příčinu? A jestliže se nám ji podaří nalézt, jakým způsobem ji eliminovat, nebo alespoň minimalizovat riziko jejího výskytu?

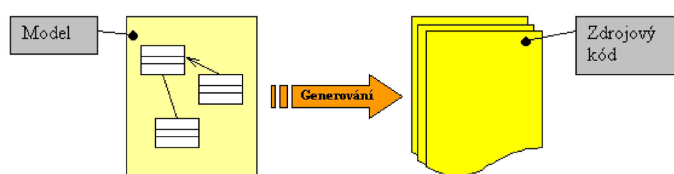
Tento příspěvek v úvodu popíše obecné rysy generování zdrojového kódu z modelů v UML. V další části pojedná podrobněji o možných problémech a nebezpečích, které jsou s generováním kódu spojeny. Na závěr nastíní několik doporučení, jak se jich v co největší míře vyvarovat.

Základní charakteristiky generování zdrojového kódu

Co je generování zdrojového kódu

Generováním zdrojového kódu lze nazvat proces, jehož vstupem je model systému a výstupem je zdrojový kód v určitém jazyce.

Modelem může být například datový model či objektově orientovaný model v jazyce UML. Typu modelu odpovídá typ jazyka vygenerovaného kódu. Z datového modelu se většinou generuje SQL skript, z objektově orientovaného modelu je generován kód v některém z objektově orientovaných jazyků (C++, Java, C#, Object Pascal, Smalltalk a další). Všechny typy modelů lze navíc transformovat do kódu v jazyce pro popis strukturovaných dokumentů, z nichž nejběžnější je v současné době XML.



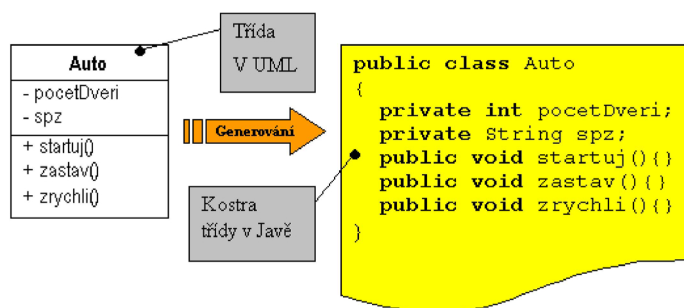
Obr. 1 Generování zdrojového kódu z modelu

Jednotka, která generování kódu provádí, se označuje jako generátor kódu. Touto jednotkou může být například člověk, samostatná aplikace či komponenta nějakého CASE nástroje (nástroje pro modelování systémů).

V následujících odstavcích bude věnována pozornost situacím, kdy vstupním modelem bude objektově orientovaný model v UML, výstupní kód bude generován automaticky (aplikací či komponentou) a v objektovém jazyce.

Význam generování zdrojového kódu

Význam automatického generování zdrojového kódu z UML spočívá především v tom, že část kódu, který by bylo nutné psát ručně, za nás vyrobí generátor, z čehož plyne výhoda úspory času. Je ovšem nutné zdůraznit, že není generován celý zdrojový kód, ale pouze jakási jeho kostra. Konkrétně se jedná např. o definice tříd a jejich atributů, hlavičky metod apod. Po vygenerování kostry zdrojového kódu je potřeba doplnit zbývající části jako těla metod a další.



Obr. 2 Příklad generování kostry třídy

Míra časové úspory roste s rozsahem projektu. U malých projektů bývá rozdíl nepatrný (v těchto případech často provádíme implementaci, aniž bychom předtím vytvářeli objektový model). Pokud jde o projekty středního rozsahu, určitou odlišnost zpravidla lze zaregistrovat. Při realizaci velkých projektů se generováním kódu může ušetřit skutečně mnoho času a tím i práce.

Toto pravidlo růstu úspory času ovšem platí pouze za předpokladu, že se při generování kódu vyhneme úskalím, která nastíníme v následující kapitole. V opačném případě téměř jistě dojde k efektu přesně opačnému.

Problémy související s generováním zdrojového kódu

Jak se říká – nic není zadarmo. Toto úsloví platí v oboru softwarového vývoje vždy a všude. Jinými slovy jakékoliv přínos je vždy vykoupen něčím, co je nutné obětovat. Za novější, lepší a výkonnější kompilátor zaplatíme zpravidla vyššími nároky na hardware, případně operační systém a jiný software. Freeware nástroje nám sice ušetří peníze, nicméně přijdeme o záruku bezchybné funkčnosti. Vizuální programování přineslo jednak usnadnění tvorby uživatelského rozhraní, avšak také menší přehlednost vytvořeného kódu, a tím i více komplikací v případě nutnosti jeho ručních úprav.

Mohli bychom uvádět další a další příklady a jejich výčet by zajisté svým rozsahem vydal za samostatný příspěvek. Vraťme se však ke generování zdrojového kódu. V předchozích kapitolách jsme nastínili jeho přednosti, nyní se zaměříme na nepříjemnosti, se kterými se při jeho využívání můžeme setkat.

Vynecháme extrémní situace, kdy proces generování nedoběhne dokonce, či jeho výsledkem bude změť znaků, ani zdaleka nepřipomínající zdrojový kód. Budeme předpokládat, že vygenerovaný kód odpovídá vstupnímu modelu. Jednotlivé případy jsme rozdělili do třech skupin podle oblastí, kterých se nějakým způsobem dotýkají:

- Oblast technologie
- Oblast organizace
- Oblast psychologie

Na každou z těchto oblastí se postupně zaměříme.

Oblast technologie

Jedním z klíčových aspektů vývoje softwaru jsou vývojové technologie. Jedná se především o programovací jazyk či jazyky, kompilátory, vývojová prostředí a další. S nimi souvisí některé překážky, které při generování kódu mohou vyvstat:

1. Vygenerovaný kód neodpovídá námi používaným technologiím. Jinými slovy, kód je generován v jazyce, jenž není podporován nástroji, které při

vývoji používáme. Může se ale jednat pouze o rozdílné verze jednoho a téhož jazyka. Potom stačí malá odlišnost a překladač již vygenerovaný kód nemusí zkompilovat.

2. V případě, že náš kompilátor jazyk generovaného kódu podporuje, je vše v pořádku. Pokud se však rozhodneme o upgrade vývojových nástrojů s podporou jazyka vyšší verze, mohou nastat problémy popsané v bodě 1.

3. V oblasti vývoje softwaru občas dojde k nahrazení stávající technologie novou, úplně odlišnou. Ta bude podporovat naprosto jiný jazyk, než se používal doposud. Komplikace popsané v bodě 1 nastanou za této situace zcela jistě v mnohem větším měřítku, než v předchozím případě.

Oblast organizace

Softwarový vývoj středně velkých a velkých projektů zpravidla předpokládá týmovou spolupráci. Každý člen týmu vystupuje zpravidla v některé z následujících rolí:

- Vedoucí projektu
- Analytik
- Vývojář
- Tester

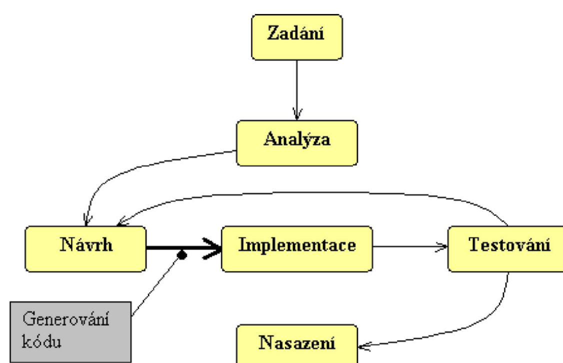
Dobré fungování vývojového týmu ovšem předpokládá jeho dobrou organizaci. Musí být dané, který člen týmu za co odpovídá, jaká jsou pravidla a v jakých termínech se musí stihnout jednotlivé dílčí práce. Pokud je při vývoji projektu používáno generování kódu, lze se setkat s těmito problémy:

1. Není zcela jasné, kdo je za generování kódu odpovědný. Je to analytik, který ručí za správnost návrhu systému, nebo vývojář, který má na starosti implementaci čili vytvoření zdrojových kódů a následné odladění?

2. Úprava vygenerovaného zdrojového kódu neodpovídá kódovacím konvencím dohodnutým v rámci týmu. Díky tomu zdrojové kódy ztrácí na jednotnosti a tím i na přehlednosti.

3. Generátor kódu plně nepodporuje používanou metodiku vývoje. Jako příklad lze uvést iterativní vývoj, kde po končeném počtu iterací postupně dospějeme k požadované podobě vytvářeného systému. V první iteraci se vytvoří prvotní návrh, ten se po dokončení implementuje a implementace se otestuje. V každé další iteraci se návrh upraví, poté se úpravy implementují a otestují. V poslední iteraci se provede poslední verze návrhu

a implementace a konečné testování. Generování kódu z prvotního návrhu bude probíhat stejně jako u neiterativního vývoje. Použití generátoru v dalších iteracích už bude komplikovanější, jelikož se budeme muset vyhnout nebezpečí přepsání dosud napsaného kódu.



Obr. 3 Generování kódu a iterativní vývoj projektu

Oblast psychologie

Psychologie má významné postavení ve všech oblastech lidského konání. Výjimkou není ani softwarový vývoj, kde úspěšné dokončení projektu je mimo jiné ovlivněno psychikou každého člena týmu. Nyní uvedeme případy, kdy psychologická stránka generování kódu může mít na vývoj negativní vliv:

1. Význam generování kódu je přeceněn. Skutečnost, že výsledkem generování kódu je pouze jeho „kostra" a že většinu stejně musí vývojář dopsat ručně, zjistí každý velice brzy, často ještě dříve, než vůbec začne generátor používat. Avšak to, že generátor je pouze výplodem člověka, člověk je tvor nedokonalý a tím pádem i generátor a potažmo vygenerovaný kód může mít nedostatky, si už každý neuvědomí. Pak se lze setkat s případy toho typu, kdy se ve vygenerované hlavičce metody `swap()` předávaly parametry hodnotou místo odkazem a vývojář strávil několik hodin nad hledáním chyby ve vlastnoručně napsaném kódu.

2. Vygenerovaný kód omezuje vývojáře. Na vygenerovaný kód se často pohlíží jako na něco, co je dané, a tudíž neměnné. Tento předsudek tak činí z generovaného kódu jakési dogma, které poněkud zužuje zorné pole kreativity vývojáře, jenž modifikaci generovaného kódu obchází pomocí jakýchsi záplat v podobě složitých konstrukcí v místech, kde jsou změny dovoleny. A to není dobré.

Řešení problémů souvisejících s generováním zdrojového kódu

Minulá kapitola zmínila některá úskalí, na která můžeme při generování zdrojového kódu narazit. Nyní se pokusíme nastínit několik způsobů, jak se s nimi vypořádat.

Oblast technologie

Způsob řešení problémů týkajících se této oblasti závisí na vlastnostech používaného generátoru. Pokud je generovaný kód v jiném jazyce či jiné verzi jazyka než používáme při implementaci, v zásadě mohou existovat dvě varianty řešení.

První z nich je, že generátor přestaneme používat a nahradíme jej jiným, který nám po stránce jazyka generovaného kódu bude vyhovovat. Mnohem schůdnější je situace v případě, že generátor kódu je uživatelsky otevřený, což znamená, že formát výstupu lze uživatelem definovat. To je umožněno například tak, že se ke generátoru připojí textový soubor, který obsahuje popis formátu výstupu pomocí metajazyka. Tento soubor je možné později modifikovat č nahradit jiným, což řeší problém s inovací či změnou vývojových technologií.

Oblast organizace

Jak již bylo řečeno výše, v projektovém týmu musí být jasné, kdo nese za co odpovědnost. To platí i pro generování kódu. Je víceméně záležitostí vedoucího projektu, zda odpovědným v tomto případě stanoví analytika či vývojáře.

Pokud jsou konvence kódování odlišné od úpravy generovaného kódu, nabízí se řešení změnit konvence. To je sice možné, avšak v případě projektu v pokročilejší fázi poněkud nepraktické. Jestliže používáme uživatelsky otevřený generátor kódu, můžeme jej našim konvencím přizpůsobit.

Pro zmíněný iterativní vývoj by se nejlépe hodil generátor, který umožňuje synchronizaci modelů návrhu s odpovídajícími zdrojovými kódy. Jinými slovy, generátor by byl schopen generovat pouze změny v modelu tak, aniž by stávající kód byl narušen. Pokud takový generátor nemáme k dispozici, pak je nutné vygenerovat kód do nových souborů a pomocí některého nástroje tyto soubory sloučit se zdrojovými kódy z minulé iterace.

Oblast psychologie

Překonat překážky týkající se psychologie bývá mnohem náročnější než přizpůsobit generátor kódu technologiím či kódovacím konvencím. Rozhodně je nutné si uvědomit, že automatické generování neznamená bezchybné generování a že se

jedná o nástroj, který nám má pomoci a nikoliv o mantinel, který bude omezovat naše tvůrčí myšlení. Je důležité, aby vedoucí projektových týmů na tyto zásady své podřízené upozorňovali a dohlíželi na jejich dodržování.

Závěr

Ačkoliv automatické generování zdrojového kódu s sebou přináší svá úskalí, rozhodně stojí za to se jím zabývat a používat jej při vývoji projektů středně velkého a velkého rozsahu. Je ovšem potřeba vybrat správný generátor, který by měl být určitě uživatelsky otevřený a případně mít i schopnost synchronizace s dosud napsaným zdrojovým kódem. Avšak i při práci s tím nejlepší generátorem kódu je potřeba mít na paměti, že o něm platí přesně to, co o ohni: je dobrý sluha, ale špatný pán.

Literatura

[1] Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modelling Language User Guide.* Addison-Wesley, 1999.

[2] Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modelling Language Reference Manual.* Addison-Wesley, 1999.

[3] Fowler, M., Scott, K.: *UML Distilled.* Addison-Wesley, 2000.

[4] Markus, V.: *A Catalog of Patterns for Program Generation.* http://www.voelter.de

[5] Sun Microsystems. http://java.sun.com

[6] Drbal, P. http://nb.vse.cz/~drbal

[7] Nouza, O.: *Principy generování zdrojového kódu z UML.* Sborník konference Objekty 2004.