VYUŽITÍ MDA PRO INTEGROVANÝ VÝVOJOVÝ NÁSTROJ QI BUILDER

Cyril Klimeš
Jaroslav Procházka

Ostravská univerzita, katedra informatiky a počítačů, 30. dubna 22, 701 03 Ostrava, ČR
E-mail: cyril.klimes@osu.cz, jaroslav.prochazka@osu.cz

Abstrakt
V článku jsou diskutována možná řešení problému odstranění klasického programování v integrovaných vývojových nástrojích sloužících pro vývoj např. ERP systémů (QI – makrojazyk, MS Navision – C/SIDE, apod.). Zásadním problémem klasického programování je, že použití chybných konstrukcí v kódu či špatná práce s pamětí mají zásadní vliv na chod výsledné aplikace. Zde jsou naznačena řešení s využitím principů MDA (Model Driven Architecture). Hlavní pozornost je věnována integrovanému vývojovému nástroji QI Builderu.

1. Úvod

Vývoj informačního systému QI (účetnictví, majetek, výroba, sklady, projekty, CRM, …) probíhá komplexně v jeho integrovaném vývojovém nástroji QI Builder. Filosofií systému QI je zkrácení a zjednodušení vývoje díky tomu, že aplikaci neprogramuji (např. v Javě, C++ nebo Delphi), ale v podstatě celou ji vytvořím analytickým modelováním. Nejdříve vytvoříme datový model aplikace, vytvoříme třídy a jejich atributy, zařadíme je do hierarchických vazeb a poté vytvoříme vazby relační-vodorovné. Dalším krokem je vytvoření množin tříd a jejich atributů, které chceme použít na formulářích, tzv. datových řezů (anglicky Data Set - DS). Jedná se v podstatě o filtry pro výběr různých tříd a atributů z databáze a to z toho důvodu, abychom nemuseli pracovat se všemi třídami datového modelu v každé aplikaci. Nad vybranými daty (tj. nad DS) tvoříme obrazovkové formuláře a tiskové výstupy. Podrobnější popis QI Builderu a metodiky tvorby byl již diskutován v [1].

Je zřejmé, že část funkčnosti je realizována již samotným datovým modelem, další část pak vhodně vytvořenými DS. Může nastat situace, kdy nám takto vytvořená funkčnost nestačí a je třeba doplnit novou funkčnost. Tvorba takové funkčnosti pomocí obecných nástrojů je mnohdy příliš komplikovaná a z tohoto důvodu se využívá v QI Builderu kód zapsaný v makrojazyku.

Makrojazyk není komplexní programovací jazyk, vymezuje pouze seznam příkazů a funkcí umožňujících doplnit analytické řešení o potřebnou funkčnost a to bez nutnosti úpravy části systému programátorem. Makrojazyk používá předdefinované třídy. Jedná se o datový typ zahrnující metody (constructor, procedure, function) a vlastnosti (property). Instance těchto tříd jsou tvořeny klasicky, pomocí konstruktorů. Definovány jsou třídy pro práci s výjimkami (EException), s datovými řezy (TAppDataSet), zpřístupňující funkce aplikačního serveru (TAppServer), zpřístupňující vlastnosti datového modelu (TDataModelling), vlastnosti formuláře (TFDyna), pro práci s poli (TField), paměťovými proměnnými, řetězci, s tiskovými sestavami, apod.

Spouštění makra probíhá na základě událostí. Je definováno několik různých událostí pro různé funkční objekty. Události záznamu datového řezu mohou být:

- po založení záznamu,
- po/před uložením záznamu,
- před výmazem,
- apod.

Události údaje DS:

- při vstupu/po opuštění údaje,
- při změně údaje,
- apod.

Události formuláře:

- před otevřením/zavřením formuláře,
- apod.

Jak již bylo zmíněno, filosofií systému QI je zkrácení a zjednodušení vývoje díky tomu, že aplikaci neprogramuji (např. v Javě, C++ nebo Delphi), ale v podstatě celou ji vytvořím analytickým modelováním v QI Builderu. Pokud je ale nutné pro vytvoření specifických funkčností použít makrojazyk, tato filosofie není dodržena. Je jasné, že pro práci s makrojazykem je třeba, aby analytik tvořící v QI Builderu, znal syntaxi, konstrukce a princip použití existujících tříd a jejich metod, formát hlaviček různých typů maker (makra údaje, DS, formu, AS, …) a spouštění maker - události. Dodržení principů a konstrukcí makrojazyka má zásadní význam na rychlost a stabilitu aplikace, k dodržení těchto zásad nás makrojazyk nijak nenutí. Mezi problémy patří neuvolnění námi otevřených DS z paměti, nevhodný způsob otevření DS (zpomalení aplikace) nebo naopak uvolnění námi neotevřených DS (možnost pádu aplikace), nepoužití transakcí může vést k nekonzistenci dat. Problém, který řešíme, je nahrazení makrojazyka resp. vložení nějakého mezikroku, který by umožnil tuto funkčnost také analyticky modelovat.

2. Možná řešení problému

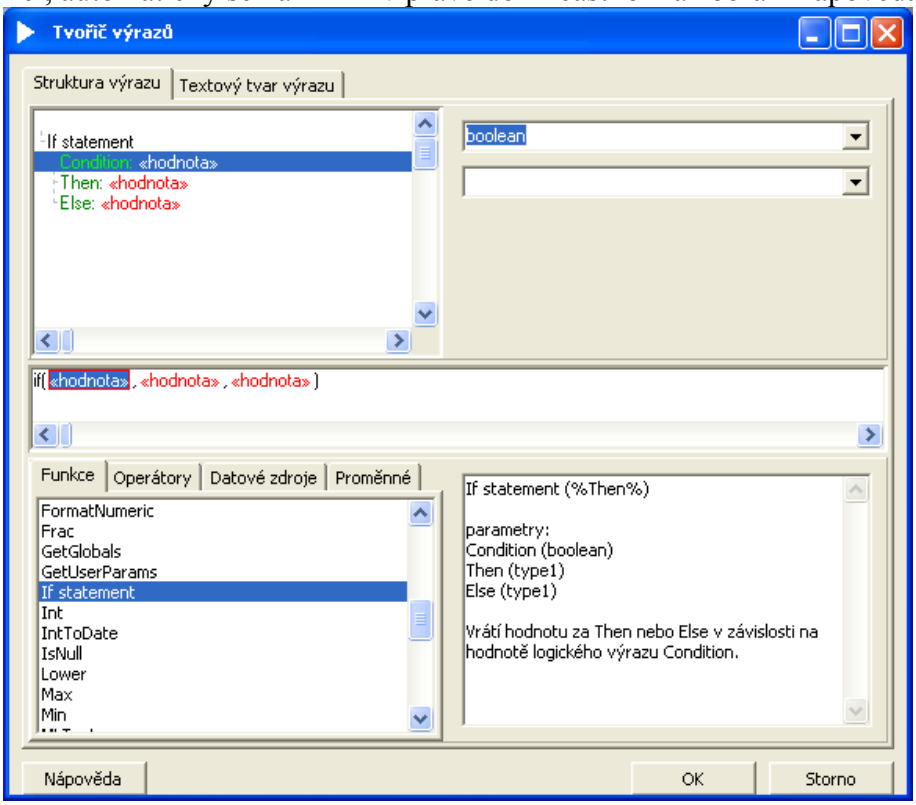
Hlavním cílem je nalézt nový způsob tvorby maker a nástroj (např. builder algoritmů, nebo grafický nástroj). S vybraným řešením souvisí ještě způsob provázání grafického nástroje s makrojazykem, tj. způsob generování maker na základě tvorby grafů či diagramů (propojení příkazů makrojazyka s grafickými objekty).

2.1 Řešení 1

Jisté řešení je v tzv. tvořiči výrazů, který je využitelný v tiskových sestavách. Výrazy (matematické, logické, …) se tvoří určitým druhem kalkulátoru. Zde je možné vložit datové pole (např. název, cena, datum, …), funkci (matematickou, logickou, statistickou, datumově-časovou, …) a také systémovou proměnou. K dispozici je několik binárních operátorů (+, -, *, /, <, >=, …) pro operace mezi těmito poli, funkcemi či proměnnými. Správnost výrazu je zkontrolována tlačítkem kontrola.

Nově navrhovaný tvořič výrazů funguje podobně, vede analytika však lépe při tvorbě výrazu, podobně jako tomu bývá u průvodců (Wizards) v aplikačních programech nebo RAD nástrojích. Okno tvořiče je rozděleno na několik částí. V jedné části je zobrazena stromová struktura výrazu. Tento strom zobrazuje jak syntaktickou strukturu, tak i způsob vyhodnocení výrazu z hlediska priority operátorů. Vyhodnocení probíhá od listu směrem ke kořeni. V prostřední části okna se nachází řádkové zobrazení výrazu, které odpovídá syntaxi daného

výrazu. Toto zobrazení je synchronizováno se stromovým. Ve spodní části se nachází seznamy pro výběr funkcí, operátorů a ostatních proměnných. Vybereme-li ze seznamu nějakou funkci, automaticky se nám k ní v pravé dolní části okna zobrazí nápověda.



Obr. 1: Tvořič výrazů

Tento uvedený tvořič ulehčuje tvorbu výrazů, je však nutné stále tvořit zápis výrazu pomocí dané syntaxe, brát v potaz priority operátorů, podmínky, apod. Jinými slovy nelze jednoduše tvořiči říct, vynásob pole *Cena bez DPH* hodnotou *DPH*. Zápis takového příkazu bude mít podobně složitou strukturu jako vidíme na obrázku 1. Tvůrce výrazů musí stále znát konstrukce jako IF – ELSE, IsNull, DateSetWhere, GetGlobals…, nemusí však znát IC a U tohoto pole, jelikož je vybere přímo z datového řezu na záložce Datové zdroje. Tento tvořič tedy zjednodušil a zpřehlednil tvorbu výrazů, ale neodstranil zápis výrazů pomocí syntaxe funkcí.

2.2 Řešení 2 – Design Patterns

Použití návrhových vzorů (anglicky Design Patterns) je řešení, kde by existovaly vzory všech typů maker (makra údaje, DS, AS, apod.), která by se určitým způsobem parametrizovala. Do základní konstrukce makra, která by byla dána vzorem a zobrazena graficky, by se uživatelsky doplňovaly pouze datové řezy, údaje, apod. Tento přístup by již umožnil odstínit nutnou znalost syntaxe všech příkazů, tříd a funkcí. Pro všechny známé problémy by tedy existovaly uložené vzory obsahující řešení. Analytik by si potřebný vzor vybral podle popisu problému, který vzor řeší. DP v podstatě reprezentují nejlepší, prověřená řešení. Ve vzoru je potřebné mít uloženo:

- motivace nebo kontext, kde se dá vzor použít,
- předpoklady pro úspěšné použití,
- popis programové struktury, kterou vzor definuje,
- seznam participantů nutných ke kompletaci řešení,
- výhody a problémy použití vzoru,

- a samozřejmě příklad použití.

2.3 Řešení 3 – Vizuální modelovací nástroj

Použití vizuálního (grafického) nástroje pro modelování funkčností systému by bylo nejvhodnějším řešením. Makra jsou pak generována z grafického zakreslení funkce, kterou chceme naprogramovat. Je zřejmé, že při použití grafického modelovacího nástroje (např. na základě UML) by musela být veškerá požadovaná funkčnost do detailu modelem popsána. Aby mohl zvolený grafický nástroj nahradit textové programování, měl by:

- být plně expresivní, výrazový, vyjadřující (síla Turingova stroje),
- musí mít více kompaktní notaci než programovací jazyk,
- musí mít efektivní, účinný „překlad“ do výkonného kódu (makrojazyka),
- musí mít jednoduchý a použitelný koncept modulů,
- podporovat možnost testování,
- nástroj musí dostatečně podporovat výše zmíněné.

Generování zdrojového kódu programovacího jazyka z vizuálního grafického modelu není nic nového. Taková řešení jsou již obsažena v RAD a CASE nástrojích, kde je možné například na základě UML diagramu tříd vygenerovat kostru zdrojového kódu. Konečnou úpravu kódu o doplnění aplikační logiky metod musí ale programátor do kódu dopsat ručně.

Podíváme-li se na řešení 2 a 3, zjistíme, že možnou cestou je využití principů MDA – Model Driven Architecture (architektura řízená modely). Snahou MDA je totiž odstínit řešitele problému od detailů technologie. V MDA nástrojích je využíváno jak bodu 3 – modelování pomocí grafické notace (konkrétně UML), tak bodu 2 – vykonávaný kód je generován také pomocí DP. Systém QI je v podstatě také založen na principech MDA. Výsledná aplikace je tvořena v QI Builderu, z velké části pomocí analytického modelování, a je odstíněna od technologického řešení (zda se jedná o databázový nebo souborový systém, v čem je implementován, apod.).

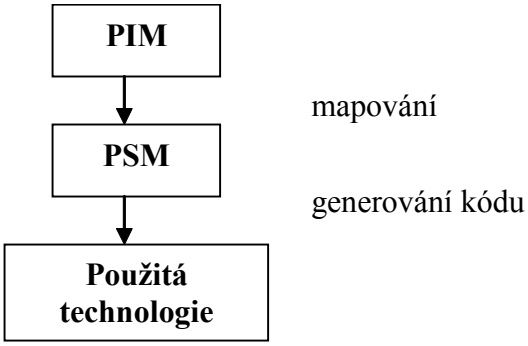
3. Principy MDA

Konsorcium OMG (Object Management Group) vede MDA jako iniciativu, která by měla vývojářům aplikací umožnit větší volnost při návrhu a portování objektově orientovaných aplikací. Je postaven na platformově nezávislém metamodelu, s nímž lze tuto migraci a implementaci provádět.

Požadavky, modely a procesy jsou nejprve zachyceny v platformově nezávislém modelu. Ten je v následujícím kroku zjemněn do platformově specifického modelu (Platform Specific Model – PSM) pro konkrétní implementační platformu – tou může být kupříkladu J2EE AS nebo CORBA server nebo také AS systému QI. Na základě takového detailního modelu jsou pak generovány proveditelné aplikace, resp. vykonavatelný kód. Důraz je přitom kladen na „generování“, s nímž při změně platformy odpovídající kód nemusí být nově implementován.

MDA je podporováno třemi klíčovými modelovacími technologiemi OMG, které jsou založené na UML: MOF (Meta-Object Facility) – slouží jako repository modelů, CWM (Common Warehouse Metamodel) – standard, který popisuje, jak reprezentovat databázové modely (schémata), OLAP a datamining modely a XMI (XML Metadata Interchange) – mapování, pomocí kterého můžeme vyjádřit UML modely v XML.

3.1 Struktura MDA



Obr. 2 Struktura MDA

PIM
Podle dnešních standardů vývoje SW specifikujeme požadavky na systém a modelujeme obchodní procesy nezávisle na konkrétní cílové platformě a tím vzniká platformě nezávislý model (Platform Independent Model – PIM). Tento základní model (the base PIM) zobrazuje strukturu a chování systému a není poznamenán technickými faktory. Technické problémy řešíme až při implementaci modelu, například použití databáze nebo aplikačního serveru od konkrétního výrobce. Díky tomu, že je PIM technologicky nezávislý, není nutné jej měnit při změně hardware či základního software. Mění se pouze, pokud se změní samotné obchodní procesy.

Modely PIM na další úrovni (aplikační modely) již obsahují některé technologické aspekty (aktivační vzory, persistenci, použití transakcí, apod.), ne však detaily specifické pro určitou platformu. Pokud tyto technologické koncepty přidáme do PIM druhé úrovně, umožníme tím přesnější mapování PIM do PSM. Začlenit tyto koncepty lze pomocí standardu OCL – Object Constrain Language, který je částí UML.

PIM, který je výsledkem prvního kroku MDA, tedy popisuje funkčnost a chování systému. UML diagramy tříd a objektů popisují strukturu, diagramy sekvencí a aktivit reprezentují chování.

Poté, co je PIM vytvořen a uložen v MOF, je provedeno mapování, jehož výstupem je PSM.

PSM
Vytvoření PSM neboli Platform Specific Model je tedy druhým krokem MDA. PSM musí být vytvořen pro každou specifickou (cílovou) platformu. V průběhu mapování jsou charakteristiky a konfigurační informace, které jsou obecně navrženy v aplikačním modelu, převedeny do specifické podoby cílové platformy. PSM je posledním krokem před automatickým generováním kódu, je tedy třeba doplnit všechny informace potřebné k vygenerování finálního kódu.

Různá rozšíření a specializace UML jsou nástroje pro vyjádření PIM a také PSM. Používají se také UML profily. Jedná se o množinu rozšíření definující UML prostředí, které je přizpůsobené jednotlivému použití, jako je modelování pro specifickou platformu.

3.2 Účel MDA

Základním účelem MDA je integrovat technologie jako CORBA, J2EE, XML či .Net, aby mohla být podporována portabilita softwaru na různé platformy. Klíčovým předpokladem k tomu je modelem řízené nasazení. Pod modelem zde rozumíme reprezentaci části funkce, struktury nebo chování systému.
Architektura MDA je podporována některými prodejci vývojových nástrojů. Podle OMG může použitím MDA dojít k redukci nákladů spojených s technologickými změnami až v řádu dvoumístných procentuálních hodnot. Přitom dojde k současnému zvýšení kvality software, neboť cílový kód je generován a proto mohou být programátorské chyby, jejichž příčinou bývá zpravidla lidský faktor, do značné míry eliminovány.

4. Závěr

Postup při náhradě programování v integrovaných nástrojích principy MDA lze realizovat následovně:
- provést výběr (nebo vytvoření) vhodného modelovacího jazyka, který by měl splňovat kriteria uvedená v navrženém řešení 3 – např. použití UML,
- z typových maker vytvořit či vygenerovat návrhové vzory (Design Patterns), na základě nichž se bude také z makro modelů generovat a vytvořit UML profil, popisující specifika a omezení daného prostředí makrojazyka,
- definovat vztahy mezi PIM a PSM, což znamená vytvořit a popsat mapovací techniky (algoritmy) mezi oběma modely,
- navrhnout a implementovat nástroje pro generování maker na základě obecných UML modelů.

Literatura:

1. Klimeš, C. - Melzer, J.: První elastický informační systém: QI. In. Sborník přednášek konference Tvorba software 2002. str. 88 – 92. TANGER, s.r.o. 2002. ISBN 80-85988-74-7
2. Klimeš C. - Melzer J.: Distribuovaný vývoj objektových databázových aplikací CASE nástrojem QI Builder. In. Sborník „Objekty 2003“, str. 283-288, Ostrava, 2003, ISBN 80-248-0274-0
3. Systém QI: www.dcconcept.cz, www.qi.cz
4. DC Concept – Developer manual.
5. DC Concept – Makrojazyk.
6. Design Patterns: Design Patterns, Elements of Reusable Object-Oriented Software Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [Gamma95]
7. MDA: www.omg.org/mda