MOŽNOSTI ROZŠÍRENIA A APLIKÁCIE JAZYKA UML PRI ANALÝZE A NÁVRHU

Ivan Polášek[1)]
Jan Ministr, Josef Fiala[2)]

[1)] Ekonomická univerzita v Bratislave, Fakulta hospodárskej informatiky, Dolnozemská cesta 1, 852 35, Bratislava, SR, ipo@gratex.com

[2)] VŠB-Technická univerzita v Ostravě, Ekonomická fakulta, Sokolská třída 33, 701 21 Ostrava, ČR, jan.ministr@vsb.cz, josef.fiala@vsb.cz

Abstrakt
Článok hlbšie analyzuje nedostatky a aplikácie diagramov jazyka UML. Popisuje využitie sekvenčného diagramu, charakterizujúceho interakcie objektov v dynamickom modeli systému, nielen pre jednoduché postupnosti ale aj v alternatívach a cykloch, v bohatých vnorených štruktúrach, čo patrí k najčastejším problémom analytikov pri modelovaní dynamického správania sa systému. Príspevok upozorňuje aj na možnosť zakresliť nevhodnú štruktúru v Activity Diagrame na reálnom príklade ako aj neschopnosť kvalitnej editácie diagramu a ponúka iné možnosti a návrhy (napríklad kombináciu a spriahnutie niekoľkých modelov cez navigátor v OO CASE systémoch)

Klúčová slová: UML,Sequence Diagram,Use Case, Activity Diagram, State Diagram, Easy Case

Úvod

Pri objektovej analýze a modelovaní systémov je dnes používaným štandardom jazyk UML, popísaný v [OMG03], ktorý je možné rozšíriť pomocou stereotypov o nové prvky a vzťahy. Ďalšou možnosťou je doplniť ponúkanú sadu diagramov o iné diagramy, prípadne doplniť existujúce diagramy čitateľným a korektným spôsobom.

1. Doplnenie sekvenčného diagramu UML pre dynamické modelovanie

Sekvenčný diagram charakterizuje interakcie objektov v dynamickom modeli systému. Vznikajú však nielen jednoduché postupnosti, ale aj v alternatívy a cykly v bohatých vnorených štruktúrach, čo patrí k najčastejším problémov analytikov pri modelovaní dynamického správania sa systému.

Napríklad už v prelomovej publikácii [GoF95] o návrhových vzoroch je sekvenčný diagram používaný popri objektových diagramoch, alebo v [BOO94] o objektovej analýze a návrhu až po najnovšie práce ([BUS01], [FOW00], [BIE00], [RUS02], [RUS03]).

Najnovšia špecifikácia jazyka v [OMG03] popisuje takéto možnosti, ale nie podrobne a dostatočne, najčastejšie len pre jednu interakciu alebo udalosť. Analytici a návrhári majú problémy najmä so zápisom postupnosti viacerých udalostí v jednom alebo vo viacerých blokoch, ktoré sa vykonávajú podmienene alebo opakovane.

Článok popisuje niekoľko možností riešenia s využitím prvkov *Activation* (aktivácia objektu - v tomto prípade napomôže znázorniť rozsah a presné umiestnenie bloku), *Message to Self* (lokálne vyvolanie udalosti na objekte, využité na vytvorenie - *vyvolanie* riadiacej štruktúry a definovanie podmienky), *Constraint* (obmedzujúce pravidlo, vhodné na uloženie podmienky pre vetvenie alebo cyklus), *Concurrent* alebo *Additional Lifeline* (alternatívna životná línia objektu, ďalej len *AL*, vhodná na znázornenie rozvetveného životného cyklu objektu pri rôznych podmienených interakciách).

Takéto využitie sekvenčného diagramu je už na pomedzí dynamického a funkčného diagramu a často je využité pre zápis volania jednotlivých metód objektov a znázornenie ich štruktúry aj v takomto pohľade. Literatúra tento typ diagramu označuje aj ako *Procedural sequence diagram*.

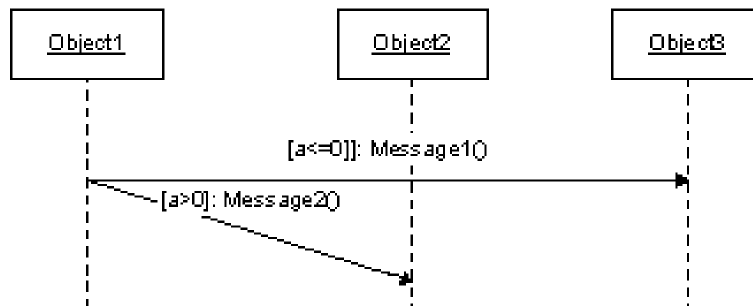
1.1 Použitie sekvenčného diagramu v projekte

Sekvenčné diagramy (*Sequence Diagrams*) rozvíjajú služby systému (*Use Cases - prípady použitia*) z Konceptuálnych diagramov (*Use Case Diagrams*) do podoby *scenárov*. Často je to prvý diagram po konceptualizácii a jeho dôležitosť je zrejmá: často je jednoduchšie charakterizovať riešenie úlohy v konkrétnom prípade použitia systému, ako vytvárať objektový model. Identifikujeme v nich tok externých udalostí, ktoré vyvolávajú odozvu systému (služby) a interných udalostí, ktoré modelujú a zabezpečujú správanie konkrétnej služby-funkcie. Udalosti opisujeme ako javy, ktoré práve nastali, najčastejšie slovnými spojeniami: *príchod hosťa*, *podanie žiadosti*, *dodanie tovaru*, *splatná faktúra* a podobne. Udalosti môžu spoluidentifikovať aj prídavné mená: *dodané*, *utriedené* a pod. Okrem udalostí spoznávame v tomto diagrame aj nové objekty, ktoré zabezpečujú príjem, spracovanie a odosielanie udalostí. Objekty označujeme aj podstatnými menami ako *fakturačné oddelenie*, *sklad*, *personálne oddelenie* a podobne. V diagrame sa pracuje nie s triedami, ale s ich inštanciami - objektmi. Možno však označiť objekt podobným menom, ako má trieda (*sklad:Sklad*), prípadne ho vynechať (*:Sklad*). Výhoda objektov je v tom, že možno vytvoriť aj viac inštancií tej istej triedy a modelovať interakcie aj medzi nimi. Ak začneme sekvenčným diagramom, kde je ďaleko jednoduchšie objavovať potrebné triedy a metódy, ktoré budú reagovať na udalosti a interakcie, potom bude následný objektový diagram pravdivejší a korektnejší. Bude obsahovať len nevyhnutné triedy a množstvo metód, ktoré by sme inak museli odhadovať.

V sekvenčnom diagrame sú v horizontálnej rovine uložené komunikujúce objekty s ťažnicami smerom nadol a vertikálna y-dimenzia predstavuje časový rozmer, kde medzi jednotlivými ťažnicami objektov sú znázornené interakcie popísanými šípkami (podmienka v hranatých zátvorkách, názov udalosti, jej prametre v guľatých zátvorkách), kde okrem jednoduchej udalosti, využívame aj synchrónnu (čaká sa na reakciu a až potom sa pokračuje ďalej) a asynchrónnu udalosť (nečaká sa na odpoveď) s polovičnou šípkou.

1.2 Bežný spôsob zápisu alternatívnych udalostí a cyklov

Obrázok č. 1 znázorňuje štandardný spôsob vetvenia udalostí. Udalosť *m1* sa vykoná, ak je splnená podmienka *a<=0*, udalosť *m2* sa vykoná ak je splnená podmienka *a>0*. Vychádzajú z jedného bodu, ale udalosť *m1* je zasielaná objektu *o3* z objektu *o1*. Udalosť *m2* zasiela objekt *o1* objektu *o2*.



Obr. č. 1 Vetvenie interakcií podľa podmienky

Opakovanie udalosti (cyklus) sa zapisuje pomocou hviezdičky pred názov udalosti. Tu vzniká problém prehľadnosti a čitateľnosti koľko udalostí zahrnúť do tohto opakovania, ako aj o type a podmienke opakovania.

1.3 Ďalšie možnosti zápisu alternatívnych udalostí

CASE systém MS Visio ponúka aj ďalší druh *obmedzenia: 2 element constraint*, v ktorom zapíšeme podmienku vykonania (zaslania) operácie/správy, s tým, že ak podmienka nie je splnená, vykoná sa asociovaná alternatíva. V tomto prípade boli použité aktivačné bloky na zviazanie niekoľkých udalostí s rovnakou podmienkou.

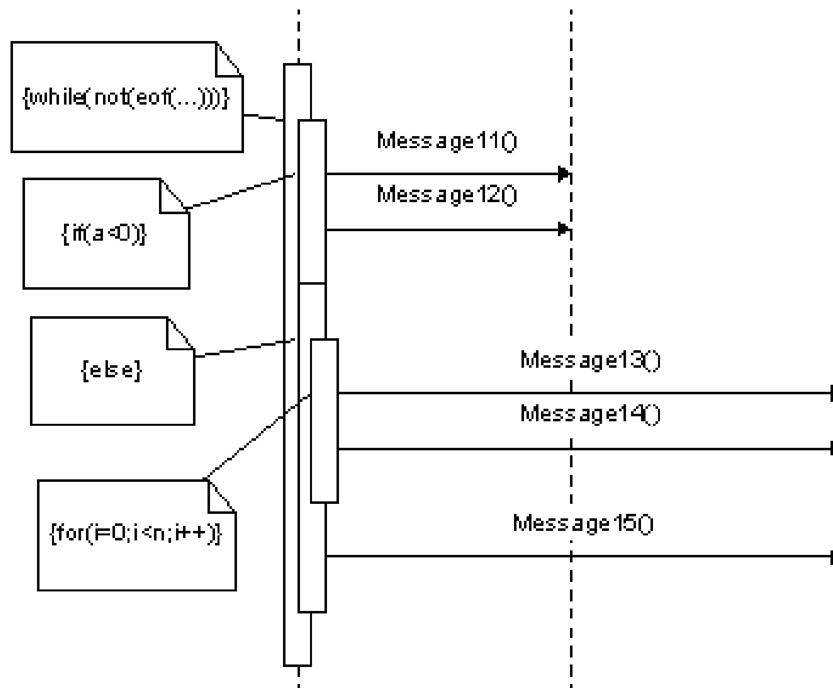
Vetvenia, ktoré obsahujú viacero udalostí v každom bloku, pričom druhá podmienka nie je doplnkom prvej nemôžu použiť *2 element constraint*. V projektoch z predošlých rokoch (popisuje aj [HAJ02]) sme riadiacu štruktúru (a jej podmienka) vetvenia a cyklu uložili ako podmienka do hranatých zátvoriek prázdnej udalosti na objekte, ktorý obsahuje telo takejto metódy, alebo inicioval takúto štruktúru. Tento spôsob sme dokonca podporili generovaním priamo do konkrétnej metódy, popísané v [HAJ02].
Ďalšou variáciou pre sprehľadnenie a zjednodušenie bolo uvedenie začiatku a konca cyklu, alebo vetvenia. Používali sme pre čo najkratší zápis: zložené zátvorky pre začiatok a koniec bloku ako v jazyku C/C++ (*{* a *}*) a nie napríklad kľúčové slová ***if* - *endif***, ***while* - *wend***, ***begin*** a ***end*** a podobne.
Tento spôsob síce fungoval a nespôsoboval problémy pri generovaní, neumožňoval však prehľadné vnáranie a zvyšoval počet interakcií na diagrame.

Použili sme preto kombináciu vnárajúcich sa blokov pomocou aktivácie a namiesto prázdnych lokálnych správ s vetvením alebo cyklom v podmienke sme využili constraint v tvare poznámkového bloku. Využitie aktivácií ako vnárajúcich sa blokov bolo jednoznačne prehľadné. Táto kombinácia (obr. č. 2) sa zatiaľ javí ako najprehľadnejšia a v tejto chvíli ju používame aj ako alternatívu pre generovanie popísané v [Pol03]. Podobný spôsob zápisu sme našli aj v [KER03], ktorý používajú pre CASE systém Together ControlCenter, čo nás utvrdzuje v akceptovateľnosti tohto spôsobu, pretože nie sme jediní, ktorí ho používajú a veríme, že v budúcnosti by sa mohol stať štandardom.

1.4. Finálny spôsob zápisu vetvenia a cyklu v sekvenčnom diagrame

Použitie spriahnutých constraintov a aktivačných blokov nás priviedlo k najprehľad-nejšiemu spôsobu kombinácie práve týchto dvoch prvkov:
1. vnáranie blokov je možné zviditeľniť vďaka umiestneniu aktivácií a subaktivácií,

2. začiatok a koniec bloku je evidentný vďaka začiatku a koncu aktivácie, nie je teda potrebné ho označovať, okrem toho ich označenie podlieha použitému programo-vaciemu jazyku v implementácii, preto nie je dostatočne všeobecné,
3. typ a podmienky bloku sú zrejmé z constraintov (poznámkového bloku), priradených k aktiváciám.

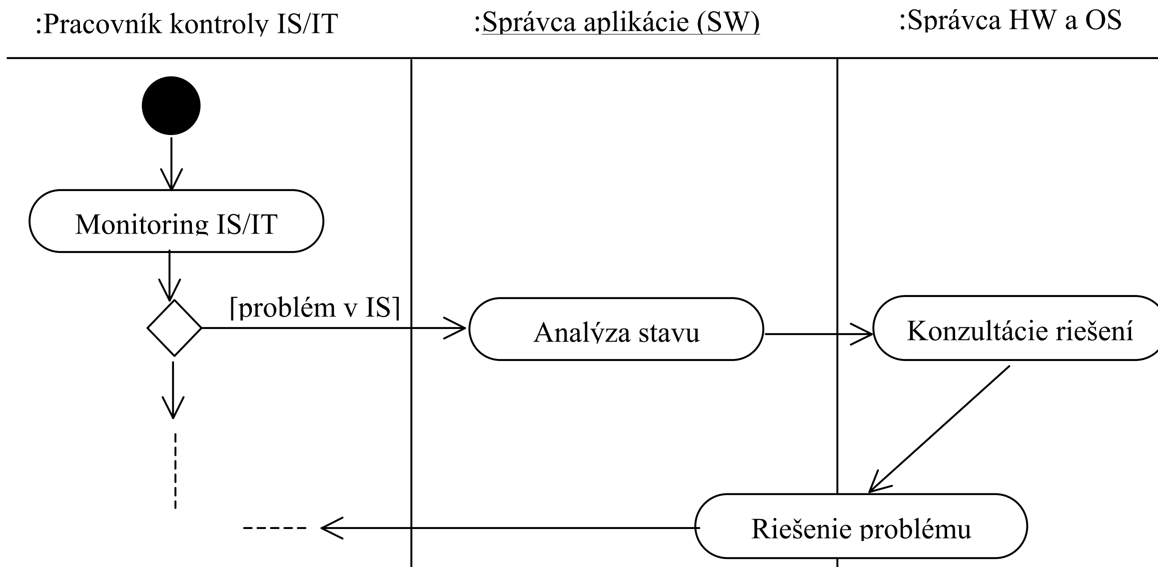


Obr. č. 2 Finálny spôsob zápisu vetvenia a cyklu v sekvenčnom diagrame

2. Nedostatky Activity Diagramu UML

Na problém needitovateľnosti Activity Diagramu, možnosti vytvárať nevhodné štruktúry sme upozornili už v [Pol97]. Diagram aktivít (Activity Diagram - *AD*) je v [OMG03] definovaný ako špeciálny prípad stavového stroja (*State Machine*), zastupovaného najmä Stavovým diagramom (*SD*, *State Diagram*) a slúži najmä na modelovanie procesov systému. Ak *SD* popisuje stavový priestor elementu (triedy alebo služby) a prechody medzi stavmi, potom *AD* definuje najmä aktivity a podmienky ich vykonania. Toto rozdelenie a príbuznosť potvrdzuje aj sémantický model z kapitoly *Semantics* v [OMG03].

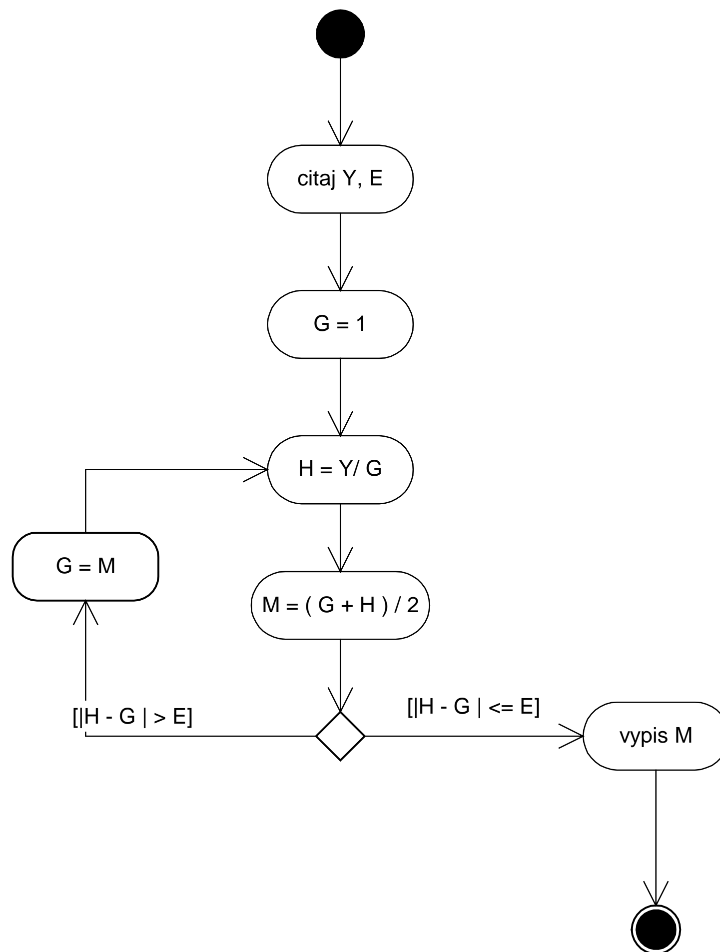
V prvých verziách UML bol podobný až zameniteľný s vývojovým diagramom, dnes sú jeho možnosti bohatšie a obsahuje implicitne aj možnosti zakresliť paralelné procesy pomocou synchronizácií typu *fork* (rozvetvenie) a *join* (spojenie). Takéto paralelizmy môžu byť aj vnorené, ako logické vetvenia. Ďalej UML dovoľuje aj rozdeliť AD podľa rolí objektov, ktoré konkrétnu aktivitu vykonávajú. Diagram je delený na polia vertikálne a v UML špecifikácii sú označované ako *swimlanes*. Príklad takto členeného diagramu je na obr. 3.



Obr. 3 Príklad rozdelenia diagramu podľa rolí objektov ([Kaj03])

2.1 Ilustračný príklad nedostatkov diagramu aktivít

Dôležitý problém absencie korektného štandardu pre zápis algoritmov, ako aj znázornenie tela metódy a jej algoritmu ilustrujeme na nasledujúcom diagrame, kde je ako príklad zapísaný Newtonov iteračný algoritmus pre druhú odmocninu.
Tento diagram aktivít sa neodlišuje takmer ničím od vývojového diagramu a dedí aj všetky nedostatky: A. nie je rozlíšené použitie rozhodovacieho bloku pre vetvenie a iteráciu alebo cyklus, B. nie je možné editovať štruktúry, prenášať celé vetvy, alebo cykly, vymazávať ich a vnárať, C. dovoľuje nevhodné štruktúry a vetvy odskokov typu *goto* ako to je viditeľné aj na príklade.
Obsahuje nevhodnú vetvu s príkazom *G=M*, ktorá sa vracia po rozhodovacom bloku, zastupujúcom spodnú podmienku iterácie späť do tela cyklu a obohacuje ho v druhom až n-tom kroku o ďalší príkaz. Pri prvom prechode cyklom sa tento príkaz nevykonáva, druhý raz už áno.



Obr. 4 Zápis iteračného algoritmu pre druhú odmocninu podľa Newtnovho iteračného algoritmu v diagrame aktivít

2.2 Implementácia ilustračného príkladu nedostatkov diagramu aktivít

Zdrojový kód v jazyku C/C++ by vyzeral nasledovne:

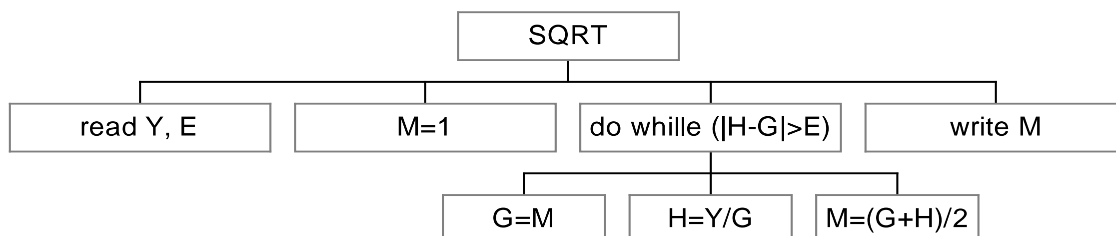
```
const double e=0.0000001;
double newton(double y)
{
        double m, g = 1, h;
start:
h = y / g;
m = (g + h) / 2;
        if(fabs(h - g) < e) goto end;
        else
        {
                g = m;
                goto start;
        }
end:
        return m;
}
```

Je zrejmé, že takáto presná implementácia síce funguje, ale vhodnejší je prepis odskokov a vetvení na menej sa vyskytujúci cyklus so spodnou podmienkou ***do – while***, bez návestí a *goto* príkazov, kedy treba zakomponovať do tela iterácie aj nevhodnú vetvu z diagramu:

```
const double e=0.0000001;
double newton2(double y)
{
	double m = 1, g, h;
	do
	{
		g = m;
		h = y / g;
		m = (g + h) / 2;
	}
	while (fabs(h - g) > e);
	return m;
}
```

2.3 Návrhy iných riešení

Pre takúto implementáciu je preto vhodnejší, grafický zápis, ktorý sa získa transformovaním kódu do hierarchickej štruktúry stromu. Ak sa zmení orientácia zľava-doprava pre hierarchické úrovne na zhora-nadol, získa sa diagram, podobný Jacksonovmu štruktúrovanému diagramu (JSD).



Obr. 5 Schéma algoritmu funkcie, druhá verzia

Okrem JSD existujú aj ďalšie možnosti, napríklad Nassi-Schneidermannove diagramy (NSD) podľa [Nas73], ktoré využíval aj systém *Easy Case* od spoločnosti *Siemens* pre generovanie jazyka C, Cobol a podobne. Táto technika je orientovaná na štruktúrované programovanie. Diagramy sa čítajú zhora-nadol a algoritmy sú zapisované do prehľadných vnárajúcich sa blokov a podblokov riadiacich štruktúr namiesto podstromov bez existencie spojníc a tým tiež zabraňuje nevhodným vetveniam typu *goto*. Je to vlastne len iné rozhranie pre ten istý princíp ako v JSD. Používa tiež tri základné štruktúry: sekvencia, selekcia, iterácia. [Pre92] označuje NSD aj ako *box diagram*, ktorý je celistvý a nedovoľuje nevhodné spojenia.

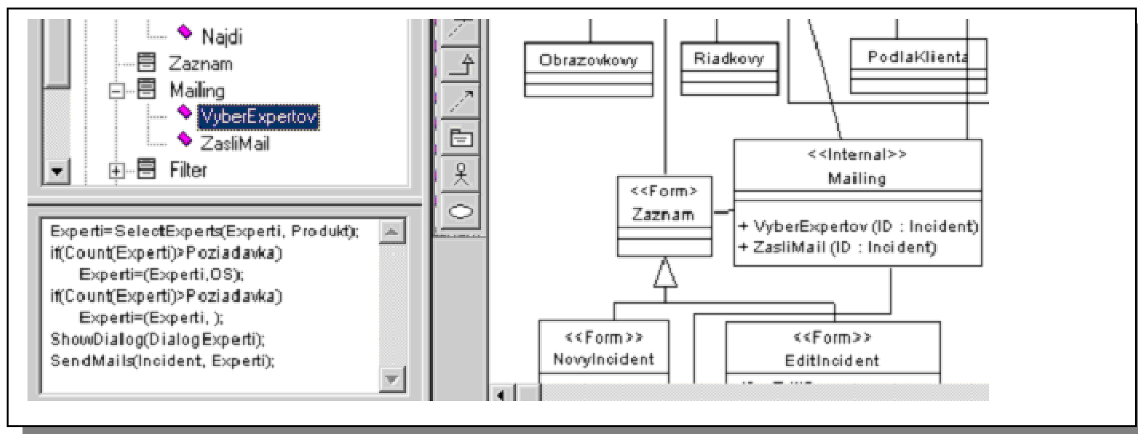
2.4 Aplikácie riešení

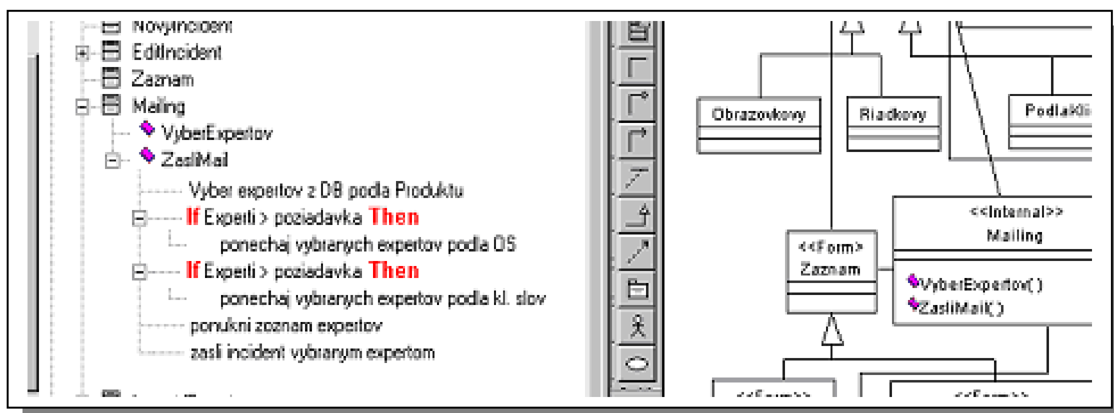
Spôsob využitia návrhu z kapitoly 2.3 bol odskúšaný na spracovaní zdrojového kódu procedúr v Transact SQL, ktorý bol ťažko čitateľný, zle udržiavateľný a jeho prezentácia zadávateľovi a uchovanie v pôvodnej forme v technickej dokumentácii neboli možné. Keďže išlo o tisícky riadkov v stovkách súborov, bolo nutné vytvoriť program na spätnú transformáciu do grafickej

podoby a generátor z týchto diagramov. Bol zvolený práve riadkovo orientovaný JSD a program bol nazvaný *SQLBeautifier*, popísaný v [Pol98]. Dovoľoval round trip engineering: napísaný kód preniesť do diagramu, meniť ho a vygenerovať pozmenený kódu naspäť.

Ďalšou možnosťou je zápis algoritmu do poznámkového bloku, vedľa metódy triedy (obr. č. 7) alebo do minieditora. Výhodou je možnosť komplexného pohľadu na algoritmy metód, nevýhodou prílišná zložitosť a nečitateľnosť väčšieho diagramu.
Štvrtá možnosť je použiť JSD a zmenou zápisu ich stromovej štruktúry z hierarchickej orientácie na riadkovú zapísať algoritmus ako podstrom v navigátore a vidieť tak telo metódy kedykoľvek po jej rozvinutí (obr. č. 8). Umožňovalo by to aj presun, vkladanie a výmaz jednotlivých vetiev metódy ale aj kopírovanie do iných metód. Používateľ by v navigátore videl jednotlivé modely a diagramy projektu, po rozvinutí ich triedy, po ďalšom rozvinutí atribúty a metódy tried. Samotné metódy by sa dali ďalej rozvinúť do úrovne jetnotlivých štruktúr algoritmu. Podľa znamienka + a – by bolo zrejmé, ktoré metódy obsahujú nadefinovaný algoritmus a aj ktoré štruktúry (cykly, podmienky) sa už dajú ďalej rozvinúť do jednotlivých podrobností. Nevýhodou by bola prílišna mohutnosť navigátora na úkor plochy diagramu a čitateľnosti, ale vďaka vhodnej stromovej štruktúre a možnosti zvinutia a rozvinutia podstromov je táto alternatíva dostatočne zaujímavá.

Obr. 6 Revidovaný JSD s funkciou insert hie s rozvinutými uzlami





Záver

V príspevku sme predostreli spôsob čo možno najprehľadnejšieho a najkorektnejšieho zápisu vetvenia a cyklu v rámci sekvenčného diagramu jazyka UML, pre ktorý sme vyvinuli aj generátor zdrojového kódu metód objektov, popísaný v [HAJ02] a v [Pol03].
Poukázali sme aj na nedostatky Activity Diagramu a spôsoby riešenia.

Literatúra:

[BIE00] Bieliková M.: Softvérové inžinierstvo, Vydavateľstvo STU, Bratislava,1.vydanie, 2000

[BOO94] Booch G.: Object-oriented analysis and design, Benjamin/Cummings, 1994

[BUS01] BUSCHMANN, Frank et al. Pattern-oriented software architecture.Wiley, 2001

[GoF95] GAMMA, E. - HELM, R. - JOHNSON, R. - VLISSIDES, J. Design Patterns. Elements of Reusable Object-Oriented Software. MA: Addison-Wesley, 1995

[HAJ02] Hajský Ľ., Mácová M.: Implementácia metód objektov, Zborník, Kvantitatívne metódy v ekonómii a podnikaní – metodológia a prax v novom tisícročí, 8. medzinárodná vedecká konferencia, 18.-20.9.2002, Bratislava, ISBN 80-225-1589-2

[FOW00] Fowler M.: UML Distilled,2nd. Edition, Addison Wesley, 2000

[Kaj03] Kajzar, D. a Polášek, I.: Projektování informačních systémů, Strukturovaný a objektový přístup, skriptá, Slezská univerzita, Filozoficko-přírodovědecká fakulta, Opava 2003

[KER03] Kern, J., Garrett C.: Effective Sequence Diagram Generation, Borland (www.borland.com/together/white_papers), 2003

[Nas73] Nassi I. and Shniederman B.: Flowcharts Techniques for Structured Programming, SIGPLAN Notices, ACM, 1973

[Pol97] Polášek I.: Návrh popisu algoritmov metód objektov v OOAaD pomocou štruktúrnych diagramov, Systémová integrace No.4, Praha, 1997, str. 63-68

[Pol98] Polášek I. a Trabalka M.: SQL Beautifier, pracovná intranet publikácia k projektu, GraTex International Bratislava,1998

[Pol03] Polášek I., Hajský Ľ.: Možnosti zápisu podmienených a opakujúcich sa interakcií v dynamickom modeli jazyka UML a automatické generovanie metód objektov, Systémová integrace No.3, Praha, 2003, str. 7-25

[OMG03] OMG: OMG Unified Modeling Language, Specification, Version 1.5, Marec 2003, www.omg.org 2003

[Pre92] Pressman S.: Software Engineering: A Practitioner's Approach, McGraw-Hill, 1992

[RUS00a] Russev S., Závodný P., Polášek I. Trenčanský I. a Ligač, M.: Tvorba informačných systémov: Metodologické aspekty, Ekonóm, Bratislava, 2000

[RUS02] Russev S., Závodný P., Grell M., Polášek I.: Metódy vývoja informačných systémov, EKONÓM, Bratislava, 2002