SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

FIIT-5221-7920

Bc. Filip Grznár

Vizualizácia dynamiky programu napísaného v jazyku C#

Diplomová práca

Vedúci práce: Ing. Peter Kapec, PhD.

máj 2013

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

FIIT-5221-7920

Bc. Filip Grznár

Vizualizácia dynamiky programu napísaného v jazyku C#

Diplomová práca

Študijný program:	Softvérové inžinierstvo
Študijný odbor:	9.2.5 Softvérové inžinierstvo
Miesto vypracovania:	Ústav aplikovanej informatiky
Vedúci práce:	Ing. Peter Kapec, PhD.

máj 2013

Zadanie: strana 1

Zadanie: strana 2

Zadanie: strana 3

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo
Autor: **Bc. Filip Grznár**
Diplomová práca: **Vizualizácia dynamiky programu napísaného v jazyku C#**
Vedúci diplomovej práce: Ing. Peter Kapec, PhD.
Máj, 2013

V rámci tejto diplomovej práce predkladáme návrh animovanej vizualizácie dynamiky objektovo - orientovaného programu v 3D priestore. Navrhovaná vizualizácia zobrazuje nielen aktuálny stav programu, ale aj časový kontext aktuálneho stavu. Okrem iného vizualizuje postupnosť vytvárania objektov a paralelné spracovanie programu. V práci tiež navrhujeme architektúru vizualizačného systému, súčasťou ktorého je komponent monitorujúci vizualizovaný program. Architektúra umožňuje získané dáta vizualizovať v reálnom čase (používateľ môže riadiť tok sledovaného programu). Navrhli sme algoritmy pre výpočet rozmiestnenia objektov a vlákien v scéne. Vrcholom práce je implementácia navrhnutého systému – programový systém SoftDynamik.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software Engineering
Author: **Bc. Filip Grznár**
Master's Thesis: **Visualization of dynamics of program written in C# language**
Supervisor: Ing. Peter Kapec, PhD.
May, 2013

Within this diploma thesis, we present the design of animated visualization of dynamics of an object – oriented program in 3D space. The designed visualization shows not only the current state of the program, but also the temporal context of current state. Among other things, it visualizes the sequence of creation objects and parallel processing of the program. We present the architecture of a visualization system, which includes a component to the monitor of the monitored program. Architecture allows visualizing the collected data in real time (the user can controls the flow of monitored program). We have designed algorithms to calculation of layout of objects and threads in a scene. The highlight of this thesis is implementation of designed system - program system SoftDynamik.

Obsah

1 Úvod ____ *12*

2 Vizualizácia softvéru ____ *14*

3 Vizualizácia softvéru ako funkcia paradigmy ____ *16*

3.1 Reálne vykonávanie programu ____ **16**

3.2 Imperatívna paradigma ____ **17**

3.2.1 Štrukturovaná paradigma ____ 18

3.2.2 Modulárna paradigma ____ 19

3.2.3 Vizualizácia pomocou diagramu aktivít ____ 20

3.3 Objektovo orientovaná paradigma ____ **21**

3.3.1 Vizualizácia pomocou sekvenčného diagramu ____ 22

3.4 Aspektovo orientovaná paradigma ____ **24**

3.5 Funkcionálna paradigma ____ **25**

3.6 Logická paradigma ____ **27**

4 Monitorovanie dynamiky programu ____ *28*

5 Komunikácia medzi monitorovaným programom a vizualizátorom ____ *29*

6 Niektoré existujúce programy vizualizujúce softvér ____ *30*

6.1 emu8086 ____ **30**

6.2 Vizualizačný nástroj VITRAIL ____ **32**

6.3 DJVis ____ **33**

6.4 Visualization of program execution in 3D environment ____ **36**

6.5 JaVis ____ **38**

6.6 Existujúce prehľady ďalších nástrojov ____ **39**

7 Návrh riešenia ____ *40*

7.1 Koncept vizualizácie ____ **40**

7.1.1 Vizualizácia stromu vytvárania objektov ____ 40

7.1.2 Vizualizácia vlákien ____ 42

7.1.3 Vizualizácia časového kontextu ____ 44

7.1.4 Dynamické obmedzenia priestoru a času vizualizácie ____ 47

7.2 Architektúra navrhnutého vizualizačného systému ____ **49**

7.3 Komponenta Monitor ____ **53**

7.3.1 Monitorovacie aspekty ____ 55

7.3.2 Reakcie na dosiahnuté videnia vtkaných aspektov ____ 56

7.4 Komponenta Trasovač ____ **58**

7.5 Menný priestor GUI ____ **61**

7.6 Menný priestor Matematika ____ **61**

7.7 Menný priestor Vizualizátor ____ **62**

7.7.1 Algoritmus rozmiestnenia objektov v scéne - ARS ____ 66

7.7.2 Algoritmus romiestnenia perifémych objekt-vlákien – ARPV ____ 71

8 Príklady aplikácií implementovaného vizualizačného sytému ____ *75*

8.1 Binárny vyhľadávací strom ____ **75**

8.2 Behaviorálny návrhový vzor Observer a viacvláknová aplikácia ___ 82

8.3 Demonštrácia riešenia problému škálovatelnosti aplikáciou navrhnutých obmedzení ___ 86

9 Záver ___ 90

Zoznam použitej literatúry ___ 93

Prílohy ___ 95

Zoznam obrázkov

Obr. č. 1 Všetky neštrukturované grafy riadenia s dvomi rozhodovacími uzlami..........18
Obr. č. 2 Plošný štruktogram..........19
Obr. č. 3 Viacvláknový sekvenčný diagram..........24
Obr. č. 4 Dátový typ bodka-dvojica..........26
Obr. č. 5 Zoznam vytvorení z bodka-dvojíc..........26
Obr. č. 6 Program emu8086 - vizualizácia pozície v kóde a hodnôt registrov..........30
Obr. č. 7 Progaram emu8086 – vizualizácia hlavnej pamäte..........31
Obr. č. 8 Program emu8086 – vizualizácia zásobníka..........31
Obr. č. 9 Metafora mesta..........32
Obr. č. 10 VITRAIL – farebná stupnica pre hrany..........32
Obr. č. 11 Vizualizačný nástroj VITRAIL..........33
Obr. č. 12 Program DJVis – Pohľad na beh programu..........34
Obr. č. 13 Program DJVis – vizualizácia tridy GraphDesktop..........35
Obr. č. 14 Program DJVis – Pohľad na triedy..........35
Obr. č. 15 Rozmiestnenie objektov okolo svojej triedy..........37
Obr. č. 16 Špirála a animácia správy..........37
Obr. č. 17 JaVis – Sekvenčný diagram vykreslený nástrojom Together..........38
Obr. č. 18 TraceCrawler..........39
Obr. č. 19 Strom vytvárania objektov pri pohľade zhora..........41
Obr. č. 20 Strom vytvárania objektov pri šikmom pohľade..........42
Obr. č. 21 Vizualizácia vlákien – pohľad zospodu..........43
Obr. č. 22 Vizualizácia vlákien – pohľad z boku..........43
Obr. č. 23 Rez obaľujúcimi sa metódami objektu – metafora letokruhov..........45
Obr. č. 24 Obaľovanie metód..........45
Obr. č. 25 Správy vytvorenia objektu a súčasného volania konštruktora..........46
Obr. č. 26 Vizualizácia synchrónnej správy..........46
Obr. č. 27 Vizualizácia asynchrónnej správy..........46
Obr. č. 28 Vysunutá hlavica objektu a symbolická vytváracia správa..........47
Obr. č. 29 ARS - Podstromy s rozdielnou priemernou hĺbkou – význam činiteľa $\overline{h}$ ...67
Obr. č. 30 ARS - Výpočet druhého argumentu funkcie max..........68
Obr. č. 31 ARS – Dcérsky podstrom P_D nesmie zasahovať do priestoru P..........69
Obr. č. 32 Vytvorenie BVS a vloženie prvej položky..........77
Obr. č. 33 Štruktúra BVS pred vložením čísla 9..........78
Obr. č. 34 BVS po vložení 4 vrcholov..........78
Obr. č. 35 BVS – vloženie čísla 9 – 0. rekurzívne vnorenie..........78
Obr. č. 36 BVS – vloženie čísla 9 – 1. rekurzívne vnorenie..........79
Obr. č. 37 BVS – vloženie čísla 9 – 2. rekurzívne vnorenie..........79
Obr. č. 38 BVS – vloženie čísla 9 – Vloženie do ľavého podstromu..........80
Obr. č. 39 BVS – Šírenie výnimky pri opakovanom vložení rovnakého kľúča..........80
Obr. č. 40 Štruktúra BVS počas výpisu..........81
Obr. č. 41 BVS – Rekurzívny výpis celého stromu..........81
Obr. č. 42 Observer – Registrácia observerov u subjektu..........83
Obr. č. 43 Kontextové menu metódy Register..........84
Obr. č. 44 Dialógové okno zobrazujúce vlastnosti metódy Register..........84
Obr. č. 45 Observer – Notifikácia observerov subjektom..........85
Obr. č. 46 Zásobníkový mód – vypnutý..........86
Obr. č. 47 Zásobnikový mód - zapnutý..........87
Obr. č. 48 Mód ignorovania premenných – vypnutý..........87

Obr. č. 49 Mód ignorovania premenných – zapnutý 88
Obr. č. 50 Mód vysúvania prázdnych objektov – vypnutý 88
Obr. č. 51 Mód vysúvania prázdnych objektov – zapnutý 89
Obr. č. 52 Mód skrývania prázdnych objektov – vypnutý 89
Obr. č. 53 Mód skrývania prázdnych objektov – zapnutý 89

Zoznam UML diagramov navrhnutého vizualizačného systému

UML diagr. č. 1 Architektúra vizualizačného systému 49
UML diagr. č. 2 Získanie a tok informácií o sledovanom programe (pri asynchrónnom spôsobe monitorovania) 52
UML diagr. č. 3 Štruktúra monitora 53
UML diagr. č. 4 Dynamika monitora a spôsob synchrónneho monitorovania 54
UML diagr. č. 5 Štruktúra trasovača 58
UML diagr. č. 6 Štruktúra trasovacích správ (návrh komunikačného protokolu) 60
UML diagr. č. 7 Stavy GUI 61
UML diagr. č. 8 Triedy menného priestoru Matematika 62
UML diagr. č. 9 Štruktúra vizualizátora 63
UML diagr. č. 10 Prvky scény a základné vzťahy medzi nimi (graf scény) 65

Zoznam tabuliek

Tab. č. 1 Riadiace signály inštrukcie mov AX, ADRESS 17
Tab. č. 2 Sledované udalosti v sledovanom programe pomocou aspektov 55

Zoznam príloh

Príloha č. 1 Obsah DVD 96
Príloha č. 2 Používateľská príručka 97
Príloha č. 3 Článok publikovaný na SCCG 2013 106

1 Úvod

Nielen programové systémy, ale aj samotné **programy vyvíjané v súčasnosti sú často rozsiahle** a pochopenie ich zdrojového kódu človekom je náročná činnosť, hlavne pokiaľ sa s daným zdrojovým kódom stretáva po prvý krát. Pochopenie neznámeho zdrojového kódu je však často potrebné. Softvér sa totiž dnes väčšinou vyvíja v tímoch, kde je vyžadované nadväzovanie na prácu kolegov. Často je potrebné meniť a dopĺňať už existujúci zdrojový kód. Pre ľudský mozog je v dôsledku miliónov rokov evolúcie prirodzenejšie vnímať informácie vhodne rozmiestnené v priestore a priestorom oddelené. Preto boli vyvinuté spôsoby ako zdrojové kódy vizualizovať. Bol definovaný grafický programovací jazyk UML. V súčasnosti sa tiež rozvíja vizualizácia softvéru ako výskumná oblasť, prichádzajúca so softvérom, ktorý samostatne získava informácie zo zdrojového kódu a vytvára jeho vizualizáciu. Prehľad týchto nástrojov prinášajú práce (Diehl, 2007) a (Teyseyre – Campo, 2009). Vizualizovať je možné štruktúru softvéru, avšak ďalším dôležitým, ale komplikovanejším krokom je vizualizovať jeho dynamiku, ktorej pochopenie je nevyhnutné k dostatočnému porozumeniu programu.

Za **cieľ našej práce** sme si zvolili vyvinúť vhodný spôsob získavania informácií o dynamike objektovo – orientovaného programu a ich vizualizáciu v 3D priestore ako animáciu, ktorá v reálnom čase reprezentuje práve vykonávaný program. Za základ našej vizualizácie sme si zvolili UML sekvenčný diagram (Booch et al., 2000), pretože ho považujeme za diagram s vysokým výpovedným potencionálom. Naviazali sme na myšlienky publikované v prácach (Mehner – Wagner, 2000) a (Mehner, 2002), ktoré na základe sledovania vykonávania programu generovali klasický 2D sekvenčný diagram. Nadviazali sme aj na myšlienky publikované v práci (Nedecký, 2010), ktorá síce nezobrazovala sekvenciu posielania správ (v jednom okamihu zobrazovala len jednu správu), išlo však o vizualizácia v 3D priestore, čo vďaka dodatočnej dimenzii umožňovalo zobraziť prehľadnejším spôsobom podstatne viac informácií ako v 2D priestore. Klasický UML sekvenčný diagram sme rozšírili o nové prvky, okrem iného sme zvýšili podporu vizualizácie viacvláknovej aplikácie, ktorá je podľa publikácie (Fleming et al., 2010) nedostatočná.

V tejto práci sa najprv zaoberáme obsahom pojmu *vizualizácia softvéru* (anglicky software visualization), ďalej skúmame vzťah medzi vizualizáciou programu a jeho paradigmou. Následne sa venujeme problematike získavania dát o dynamike programu a ich prenosu do vizualizačného programu. Práca pokračuje popisom niektorých už

existujúcich vizualizačných nástrojov a publikovaných prác. Potom uvádzame popis nami navrhnutého konceptu vizualizácie dynamiky programu a ďalej návrh a súčasne aj technické zdokumentovanie vizualizačného systému, ktorý sme vyvinuli. Následne vyvinutý systém demonštrujeme na príkladoch monitorovaných programov a tiež demonštrujeme riešenie problému škálovateľnosti. V kapitole Záver zhodnocujeme dosiahnuté výsledky, porovnávame náš spôsob vizualizácie s UML sekvenčným diagramom a náš vizualizačný systém s už existujúcim. Načrtávame tiež možnosti ďalšej práce. V prílohe uvádzame používateľskú príručku. Hlavnou a najdôležitejšou časťou práce je zdrojový kód a inštalátor vyvinuté vizualizačného systému, ktorý sme nazvali SoftDynamik a ktorý sa nachádza na priloženom elektronickom médiu.

Formálna úprava práce sa vo veciach ktoré neboli explicitne stanovené fakultou riadi akademickou príručkou (Meško – Katuščák – Findra, 2004).

Chceme vysloviť **poďakovanie** vedúcemu našej diplomovej práce, ktorým je Ing. Peter Kapec, PhD. za to že:

- Umožnil nám sa v rámci diplomovej práce zaoberať tak zaujímavou témou, keďže oblasť počítačovej grafiky považujeme za jednu z najzaujímavejších oblastí informatiky.
- Umožnil nám zvoliť si technológie a realizovať naše myšlienky bez akýchkoľvek obmedzení.
- Poskytol nám pomoc a spoluprácu pri publikovaní naších výsledkov na medzinárodnej vedeckej konferencií Spring Conference on Computer Graphics 2013 (SCCG). Zborník z tejto konferencie tiež plánuje publikovať ACM Publishing House**.**

2 Vizualizácia softvéru

Softvér je systém **inštrukcií** pre procesor a systém **dát** zodpovedajúcich týmto inštrukciám. Podľa definície IEEE z roku 1994 je softvér: *"Zbierka počítačových programov, procedúr, pravidiel a s nimi spojenou dokumentáciou a údajmi"* (Bieliková, 2000, s. 3).

*„**Vizualizácia** je proces transformujúci informáciu do vizuálnej formy, umožňujúci používateľom sledovať informáciu. Výsledné vizuálne zobrazenie umožňuje vedcom a inžinierom vizuálne **vnímať funkcie** ktoré sú skryté v dátach, ale sú potrebné pre ich výskum a analýzu.*" (Diehl, 2007, s. 1).

Diehl (2007) vyčleňuje dve hlavné disciplíny vizualizácie ako takej:

- vedecká vizualizácia: spracováva fyzické dáta,
- vizualizácia informácií: spracováva abstraktné dáta.

Vizualizáciu softvéru pokladá za vizualizáciu artefaktov vzťahujúcich sa k softvéru a k procesom jeho vývoja a zaraďuje ju do oblasti vizualizácie informácií. (Diehel, 2007).

Je možné vyčleniť tieto aspekty vizualizácie softvéru:

- **Vizualizácia statiky** softvéru – je bezčasová. Vizualizuje **štruktúru** elementov softvéru a vzťahy medzi nimi. Patrí sem vizualizácia dátových štruktúr, zdrojového kódu (z neho je možné napríklad zostaviť statický graf volaní), organizácie programu do modulov a podobne.
- **Vizualizácia dynamiky** softvéru (**behaviorálna** vizualizácia) – Obsahuje časovú dimenziu. Vizualizuje vykonávanie programu. Vykonávanie programu je časová postupnosť jeho stavov. Pod stavom programu možno rozumieť pozíciu aktuálne vykonávanej inštrukcie v inštrukčnom segmente, spolu s dátami programu. Táto postupnosť teda závisí nielen od inštrukcií programu, ale aj od vstupných dát, ktoré môžu byť pri každom spustení programu iné. Program môže získavať vstupné dáta z externého prostredia aj v priebehu výpočtu. Patrí sem napríklad vizualizácia časovej postupnosti volania funkcií programu.

Vyčlenenie štrukturálneho (bezčasového) a behaviorálneho (časového) aspektu softvéru sa zdá byť univerzálne. Tieto aspekty vyčleňuje aj modelovací jazyk UML keď diagramy, ktoré obsahuje rozdeľuje na diagramy štruktúr a diagramy správania (Fowler,

2009). Podobne sa Gammove návrhové vzory rozdeľujú na vytváracie, štrukturálne a vzory správania (Gamma et al., 2003).

Diehl (2007) medzi aspekty vizualizácie softvéru zaraďuje aj vizualizáciu evolúcie softvéru, ktorá vyzualizuje procesy jeho vývoja ako napríklad pridávanie novej funkcionality a odstraňovanie detegovaných chýb.

3 Vizualizácia softvéru ako funkcia paradigmy

"Slovo ***paradigma*** *pochádza z gréckeho slova παραδειγμα pre vzor či model. Vo filozofií tento pojem znamená určitý vzorec myslení. Teda súhrn domnienok,* ***predstáv*** *a metodických* ***pravidiel*** *pre riešenie určitého okruhu problému"* (Vaníček et al., 2007, s. 249). Tieto predstavy nemusia zodpovedať fyzikálnej realite a fyzikálnym dejom, ktoré pri riešení problému v skutočnosti prebiehajú, môžu ich viac alebo menej abstrahovať.

Návrhár a programátor používajú pri vytváraní, alebo chápaní programu spôsob myslenia vlastný paradigme, ktorú daný program používa. Preto je pre nich výhodné, aby aj vizualizácia tohto programu vychádzala z rovnakej paradigmy. To znamená, aby vizualizovala entity a vzťahy medzi nimi zodpovedajúce predstavám danej paradigmy. Preto v tejto kapitole popíšeme rôzne paradigmy používané pri tvorbe softvéru, entity a vzťahy, ktoré v sebe obsahujú a ktoré je teda vhodné vizualizovať. Poukážeme tiež na ich súvis a odlišnosť od reálnych fyzikálnych dejov, ktoré prebiehajú pri vykonávaní programu.

3.1 Reálne vykonávanie programu

Krajčovič (2000) popisuje koncept fungovania procesora architektúry SISD. Každej inštrukcií z inštrukčnej sady procesora zodpovedá príslušný mikroprogram, ktorý ju realizuje. Mikroprogramy sú zaznamenané v pamäti mikroprogramov, ktorá sa nachádza v riadiacej jednotke procesora. Pamäť mikroprogramov tak definuje inštrukčný súbor procesora. Mikroprogram sa skladá z mikroinštrukcií. Mikroinštrukcie obsahujú riadiace vektory pozostávajúce z hodnôt riadiacich signálov, ktoré vysiela riadiaca jednotka procesora prvkom operačnej časti procesora, ktoré na ich základe vykonávajú požadované výpočty a komunikujú s okolím.

Riadiaci signál je realizovaný nastavovaním hodnôt nejakej fyzikálnej veličiny (na nízku alebo vysokú hodnotu, čomu zodpovedá kódovanie 0 a 1). Má podobu napríklad elektrického prúdu (zmeny kvantových stavov elektrónov vo vodiči).

Vykonávanie programu je diskrétna časová postupnosť vektorov hodnôt riadiacich signálov (riadiacich vektorov), ktoré v konečnom dôsledku pracujú s pamäťami. Akýkoľvek program vykonávaný na danom procesore je v princípe možné **vizualizovať** vizualizovaním hodnôt jeho **riadiacich signálov** a ich vplyvu na dáta uložené v **pamätiach** (v registroch a v hlavnej pamäti).

Krajčovič (2000) uvádza obsah pamäte niektorých mikroprogramov. Pre lepšiu ilustráciu uvedieme hodnoty riadiacich signálov mikroprogramu inštrukcie pre načítanie údaju z hlavnej pamäte na adrese ADRESS do registra AX v procesore (mov AX, ADRESS):

Tab. č. 1 Riadiace signály inštrukcie mov AX, ADRESS

	Riadiaci signál a hodnota, na ktorú je nastavený	**Dôsledok**
1. mikroinštrukcia	MUX1 = 1	Príslušný multiplexor prepojí vnútornú adresnú zbernicu s registrom obsahujúcim adresu ADRESS (načítal ju tam mikroprogram na výber inštrukcie z pamäte v rámci inštrukčného cyklu).
	WR1 = 1	Zapísanie adresy ADRESS z internej na externú adresnú zbernicu.
	MEMR# = 0	Aktivuje sa signál čítania z hlavnej pamäte na riadiacej zbernici.
2. mikroinštrukcia	MEMR# = 0	Aktivuje sa signál čítania z hlavnej pamäte na riadiacej zbernici.
	RD2 = 1	Oddeľovač dátovej zbernice zapíše hodnotu z externej dátovej zbernice (ktorú nastavila hlavná pamäť) na internú dátovú zbernicu
	WRAX = 1	Register AX načíta dáta z internej dátovej zbernice.

3.2 Imperatívna paradigma

Imperatívna (príkazová) paradigma je najstaršia paradigma tvorby softvéru. Vychádza z predstavy počítača ako automatu, ktorého činnosť pozostáva zo zmien obsahu adresovateľnej pamäti obsahujúcej dáta a inštrukcie. Zmeny pamäťových miest sa v čase vykonávajú diskrétne. Po vykonaní inštrukcie sa implicitne pokračuje inštrukciou uloženou na nasledovnom pamäťovom mieste. (Vaníček et al., 2007)

Medzi imperatívne programovacie jazyky patrí jazyk symbolických inštrukcií (assembler) – poskytuje inštrukcie, ktoré **abstrahujú** nastavovanie hodnôt **riadiacich signálov**. Tiež sem patria imperatívne vyššie programovacie jazyky ako C, Pascal,... ktoré sú schopné jediným príkazom vykonať celý blok inštrukcií.

Vizualizovať dynamiku imperatívneho programu je možné pomocou **vývojového diagramu**. Je to orientovaný graf obsahujúci nasledovné typy vrcholov:

- <u>Začiatočný vrchol</u>: vrchol, z ktorého 1 hrana vychádza – vrchol, v ktorom program začína (je jediný v grafe).

- Koncový vrchol: vrchol do ktorého 1 hrana vchádza – vrchol, v ktorom program končí (je jediný v grafe).
- Vrchol, do ktorého 1 hrana vchádza a 1 z neho vychádza – vrchol reprezentuje akciu, ktorá mení obsah pamäte. Vrchol sa znázorňuje obdĺžnikom.
- Rozhodovací vrchol: vrchol, do ktorého 1 hrana vchádza a vychádzajú z neho 2 alebo viac hrán – vrchol reprezentuje akciu, ktorá mení obsah pamäte a následne dochádza na základe splnenia nejakej podmienky / podmienok k rozhodnutiu, ktorým smerom sa bude pokračovať. Vrchol sa znázorňuje kosoštvorcom.
- Zlučovací vrchol: vrchol do ktorého vchádzajú 2 alebo viac hrán a vychádza z neho 1 hrana – v tomto vrchole sa nevykonáva žiadna činnosť. Znázorňuje sa kruhom.

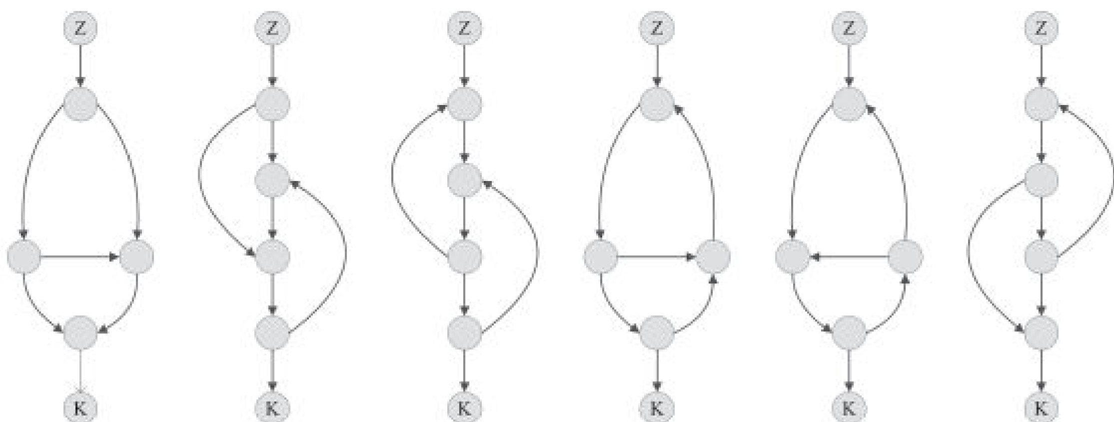
Graf je súvislý a každý jeho vrchol leží na ceste medzi začiatočným a koncovým vrcholom. (Vanička et al., 2007)

Vynechaním vrcholov, ktoré majú len 1 vstupnú a 1 výstupnú hranu vo vývojovom diagrame dostaneme **graf riadenia algoritmu**. Je to orientovaný muligraf. (Vanička et al., 2007)

3.2.1 Štrukturovaná paradigma

Čisto imperatívna paradigma umožňuje (vďaka skokovým inštukciám typu goto) vytvárať algoritmy s komplikovanými grafmi riadenia, ako je znázornené na Obr. č. 1:

Obr. č. 1 Všetky neštrukturované grafy riadenia s dvomi rozhodovacími uzlami



Zdroj: Vaniček et al. (2007. s 266)

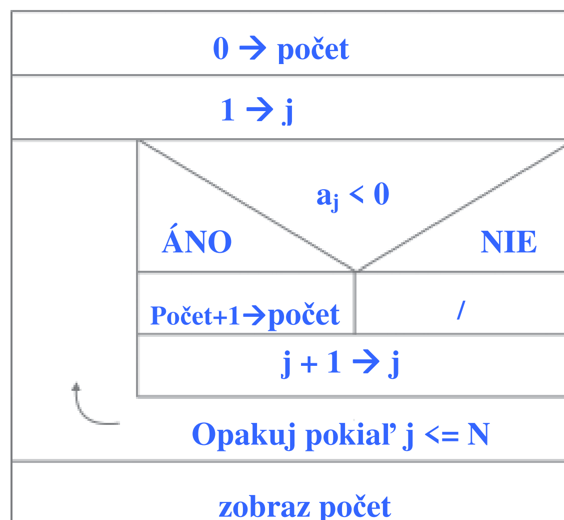
Takto vznikajú programy, ktoré sa ťažko chápu a ktoré sa ťažko udržujú. Štruktúrovaný algoritmus (D-štrukturovaný algoritmus, D podľa Dijkstra) preto program definuje ako

postupnosť nasledovných konštrukcií: **sekvencia**, **selekcia** (napríklad príkaz if), **iterácia** (napríklad príkazy while, for). (Tým, že konštrukcie selekcia a iterácia zapuzdrujú (abstrahujú) v sebe rozhodovací a zlučovací vrchol vývojového diagramu neumožňujú vykonávať ľubovoľné skoky a tým program komplikovať). BJ-algoritmus navyše umožňuje, aby mali konštrukcie selekcia a iterácia aj viac ako jeden výstup (napríklad pomocou príkazu break, exit,...).

Štrukturované riadiace konštrukcie zodpovedajú štrukturovaným dátovým typom (polia, prúdy (napríklad spájaný zoznam), záznamy(šturktúry)) (Vaníček et al., 2007)

Štrukturovaný algoritmus **vizualizujeme** pomocou **plošného šturkturogramu**. Ten neumožňuje znázorniť neštrukturovaný program.

Obr. č. 2 Plošný štruktogram



Zdroj: Vaníček et al. (2007. s 269)

Štrukturovaná paradigma je špeciálny prípad imperatívnej paradigmy. Je ju možné aplikovať aj pomocou jazyka symbolických inštrukcií.

3.2.2 Modulárna paradigma

Modulárna paradigma je len súhrn zásad ako programové moduly (relatívne samostatná časť programu, napríklad procedúra, funkcia) vytvárať a riadiť ich vzájomnú spoluprácu. Výhodou modulov je možnosť spracovávať ich samostatne a ich znovupoužiteľnosť. Súdržnosť modulu by mala byť čo najväčšia, previazanosť medzi modulmi čo najmenšia

Odovzdávanie riadenia medzi modulmi je možné **vizualizovať** pomocou **grafu volaní modulov.** Je to orientovaný graf, kde vrcholy znázorňujú programové moduly.

Ak hrana vychádza z vrchola M a vchádza do vrchola N, potom podul M volá modul N. Priama rekurzia je znázornená ako slučka v grafe, nepriama rekurzia ako cyklus v grafe. (Vaníček et al., 2007).

3.2.3 Vizualizácia pomocou diagramu aktivít

Diagram aktivít patrí medzi behaviorálne UML diagramy. Je to v podstate **vylepšený vývojový diagram**. Fowler (2009) popisuje jeho rozšírené vizualizačné možnosti. Medzi vylepšenia patrí:

- Označenie aktuálne vykonávanej akcie pomocou **tokenu**. *„Tokeny môžete zviditeľniť pomocou symbolu mince alebo žetónu posúvajúceho sa po celom diagrame“* (Fowler, 2009, s. 122).
- Pomocou rozširujúcich oblastí, je možné skrátene znázorniť iteráciu nad poliami a prúdmi - riadiacu konštrukciu typu **foreach** (štrukturovaná paradigma)
- Pomocou **oddielov** (plaveckých dráh) je možné znázorniť dekompozíciu programu na moduly (modulárna paradigma).
- Ak akcia reprezentuje celú procedúru, alebo funkciu, môžeme znázorniť vstupné **parametre** a výstupné parametre (**návratové hodnoty**) pomocou **pinov**. Pin je malý obdĺžnik na okraji akcie. Môžeme tak znázorniť nielen prenos riadenia z podprogramu na podprogram, ale aj **prenos dát** medzi nimi.
- Hlavný prínos diagramu aktivít oproti vývojovému diagramu je jeho schopnosť vizualizovať **paralelné správanie** (vlákna). Fyzikálny svet sa javí ako vysoko paralelný (napríklad sily medzi atómami vesmíru pôsobia súčasne, nie sekvenčne) čo robí problémy pri jeho simulácií v počítači. V tejto súvislosti diagram definuje ďalšie typy vrcholov:
    - Rozvetvenie (rozdelenie, anglicky fork). Je znázornené úsečkou. Má jednu vstupnú hranu a 2 alebo viac výstupných.
    - Spojenie (anglicky join). Je znázornené úsečkou. Má 2 alebo viac vstupných hrán a 1 výstupnú.
    - Ukončenie toku (anglicky flow final) vizualizuje koniec vlákna, nie však celej aktivity (celého procesu (programu)). Znázorňuje sa kružnicou, do ktorej je vpísané x.

  V diagrame programu s viacerými vláknami sa môže v jednom okamihu nachádzať viacero tokenov. Rozvetvenie vyšle do každej svojej výstupnej hrany jeden token. Spojenie čaká na to kým dostane tokeny od všetkých

vstupných hrán až potom vyšle token po výstupnej hrane. Je však možné pomocou špecifikácie spojenia (booleovský výraz, ktorý sa vyhodnocuje pri každom príchode tokenu) toto implicitné pravidlo zmeniť.

3.3 Objektovo orientovaná paradigma

Vaníčk et al. (2007) uvádza kritéria, ktoré musí systém spĺňať, aby ho bolo možné považovať za objektovo orientovaný:

- **Objekt je** logická množina údajov (atribútov objektu) a operácií nad nimi (metód). Hodnoty atribútov a kódy metód sú zapúzdrené - zvonku neprístupné. Skupina metód tvorí rozhranie, ktoré je jediným prostriedkom pre manipuláciu s hodnotami objektu (takzvaná stena z metód).
- Objekty, ktoré majú rôzne hodnoty atribútov tých istých typov a rovnaké metódy sú inštanciami rovnakej **triedy**. Množina tried v systéme je čiastočne usporiadaná reláciou dedenia. Toto je možné **vizualizovať** orientovaným grafom**.**
- Objekty medzi sebou **komunikujú** posielaním správ (volaním metód iných objektov). Správy zo sebou nesú parametre (dáta). Množina správ, ktoré môže objekt prijať sa nazýva **protokol**. Na rovnakú správu môžu rôzne objekty reagovať rôzne. Tento jav sa nazýva polymorfizmus. Po tom čo objekt dostal správu môže ako rekciu na ňu vyslať správy iným objektom.
- Medzi objektmi sú vzťahy:
  - Skladanie: Atribút objektu je tvorený iným objektom.
  - Závislosť: Zmena stavu iného objektu vyvolá zaslanie správy (volanie metódy) inému objektu.
  - Delegovanie: Objekt po prijatí správy správu odošle inému objektu, ktorý na ňu reaguje za pôvodný objekt.

Napísať program podľa objektovo orientovanej paradigmy znamená implementovať objekty a definovať väzby (vzťahy) medzi nimi. Nejedná sa tu teda o presný popis algoritmu. *„Čistý, objektovo orientovaný systém nepotrebuje hlavný program ani dekompozíciu na podprogramy. Jeho beh začína vonkajšou udalosťou z jeho rozhrania, ktorá sa interpretuje ako správa poslaná objektom, ktoré majú na starosti vstup a výstup“* (Vaníček et al., 2007, s. 257). *“Na čistý OO systém je možné sa dívať ako na asynchrónny simulačný model”* (Vaníček et al., 2007, s. 275). Možno teda povedať, že objektovo orientovaná paradigma na rozdiel od imperatívnej paradigmy

abstrahuje presnú **časovú postupnosť vykonávania programu ako celku** (to ale neznamená, že ju nie je možné interpretáciou zdrojového kódu jednoznačne určiť).

Jazyky založené na objektovo orientovanej paradigme je možné rozdeliť na:

- Čisto objektovo orientované jazyky: Neobsahujú žiadne iné dátové typy, len objekty. Chýbajú v nich niektoré procedurálne konštrukcie ako sú podprogramy a príkazy skoku. (Vaníček et al., 2007)
- Hybridné objektovo orientované jazyky: Vznikli obohatením klasických jazykov o objektovo orientované prvky. Umožňujú obchádzať zásady objektovo orientovaného programovania. Patria sem Object Pascal a C++. Vaníček et al. (2007) sem zaraďujú aj jazyky **C#**, Visual Basic a Javu.

3.3.1 Vizualizácia pomocou sekvenčného diagramu

Sekvenčný diagram patrí medzi behaviorálne UML diagramy a v rámci nich patrí do skupiny diagramov interakcií. *"Diagram interakcií (interaction diagram) popisuje, ako skupiny objektov spolupracujú v rámci nejakého správania sa. UML definuje viac foriem diagramov interakcií, z ktorých najbežnejší je sekvenčný diagram"* (Fowler, 2009, s. 67). *„Sekvenčné diagramy znázorňujú interakcie medzi čiarami života ako časovo usporiadanú postupnosť udalostí. Tieto diagramy sú najbohatšou a najpružnejšou formou diagramov interakcie"* (Arlow – Neustadt, 2007, s. 253).

"Jednou z výhod sekvenčného diagramu je, že takmer nemusím vysvetľovať jeho notáciu" (Fowler, 2009, s. 68).

Sekvenčný diagram má dve dimenzie. Prvú (horizontálnu) dimenziu tvorí zoznam účastníkov systému, druhú (vertikálnu) dimenziu tvorí čas (nemusí bežať rovnomerne). Každému účastníkovi zodpovedá jedna čiara života (anglicky lifeline).

Vychádzajúc z Fowlerovho (2009) popisu sekvenčného diagramu, tento diagram umožňuje vizualizovať:

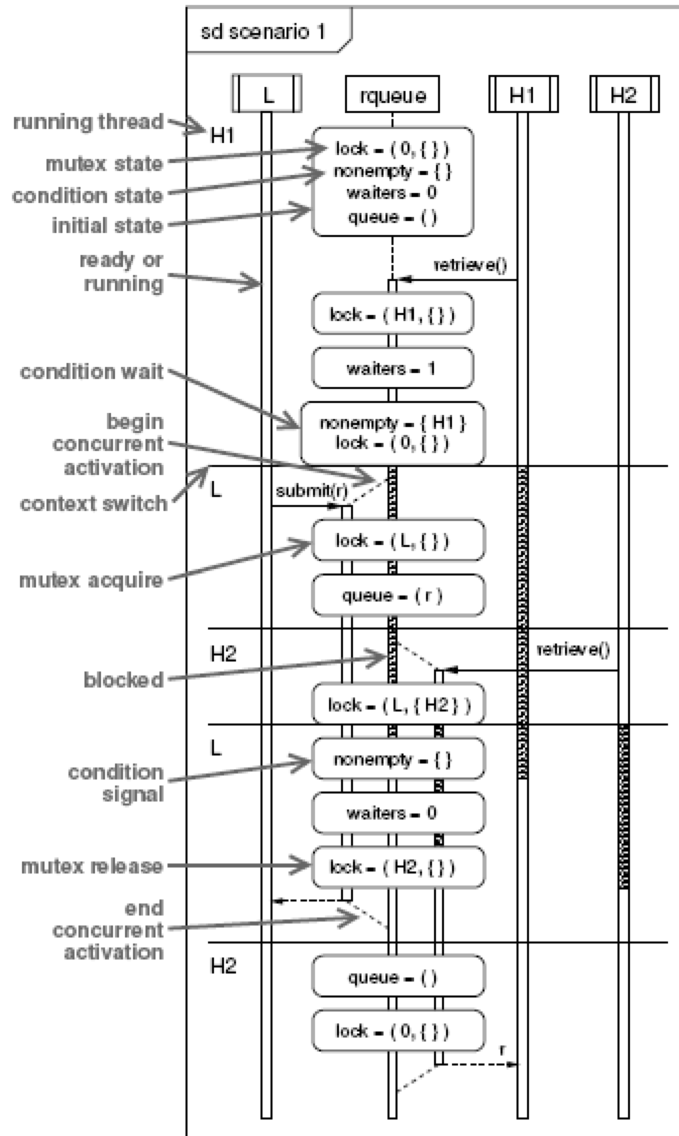
- **Objekty** nachádzajúce sa v danom časovom okamihu v programe (ako účastnikov). Objekt je znázornený obdĺžnikom, ktorý je umiestnený v čase (časom je vertikálna poloha), v ktorom bol vytvorený a z ktorého vychádza čiara života. V obdĺžniku je uvedený typ objektu (trieda).
- **Metódy** objektov (presnejšie ich vykonanie) sú znázornené ako pruh na čiare života nazývaný pruh aktivácie. Začína v čase začatia vykonávania metódy a končí v čase ukončenia metódy (to znamená v čase návratu návratovej hodnoty funkcie a jej odstránenia zo zásobníka). Pokiaľ sú na zásobníku v rovnakom čase

2 (alebo viac) metódy rovnakého objektu, tieto sú znázornené ako 2 (alebo viac) čiastočne sa prekrývajúce pruhy aktivácie. Takto je možné znázorniť aj jednotlivé iterácie rekurzie.

- **Volanie metódy** (synchrónna správa) je v zodpovedajúcom čase znázornená ako šípka s hrotom vo forme plného trojuholníka vychádzajúca z volajúcej metódy a končiaca na začiatku pruhu aktivácie volanej metódy. Nad touto šípkou je uvedený názov metódy a jej parametre.
  "Prvá správa nemá účastníka, ktorý by ju poslal, pretože správa pochádza z neurčitého zdroja. Takú správu nazývame ***nájdená správa*** *(found message)"* (Fowler, 2009, s. 69).
- **Návrat** návratovej hodnoty ako prerušovaná šípka začínajúca na konci pruhu aktivácie a končiaca na pruhu aktivácie volajúcej metódy.
- **Vlastnosti** (prístupové metódy) sa dajú vizualizovať ako metódy. Zároveň môžu reprezentovať aj im zodpovedajúce atribúty (v terminalógií jazykov C# a Java polia (anglicky field), ktoré sa priamo v sekvenčnom diagrame vizualizovať nedajú).
- **Vytvorenie nového objektu**: sa vizualizuje ako správa vychádzajúca od metódy, ktorá daný objekt vytvorila a končiaca na okraji obdĺžnika, ktorým vizualizujeme vytváraný objekt. *„Ak začne účastník hneď po vytvorení niečo vykonávať umiestnime začiatok aktivácie hneď pod obdĺžnik účastníka"* (Fowler, 2009, s. 70). Takto je možné vizualizovať **konštruktor**.
- **Zrušenie objektu**: sa vizualizuje ako krížik (v tvare písmena X) na konci čiary života v čase, ktorý zodpovedá zrušeniu objektu. Ak zrušenie objektu vyvolala nejaká metóda, znázorní sa zaslanie správy z tejto metódy šípkou končiacou v strede krížika.
- **Vytvorenie nového vlákna** (zaslanie asynchrónnej správy): Asynchrónna správa sa vizualizuje šípkou s jednoduchým hrotom. Že ide o asynchrónne volanie je možné zdôrazniť aj pomocou stereotypu.
- Vetvenie (podmienku), cyklus, vyskočenie z cyklu (break), súbežnosť, kritickú sekciu (synchronizáciu vlákien),... je možné znázorniť pomocou **interakčných rámcov**, ktoré podrobnejšie opisuje Fowler (2009) na stranách 71 – 73 a Arlow – Neustadt (2007) na stranách 260 – 267.

„Hoci sú UML sekvenčné diagramy široko používané pre reprezentáciu interakcií medzi objektmi, nazdávame sa, že ich notácia je neadekvátne pre vyjadrenie niektorých úskalí interakcií medzi vláknami." (Fleming at al., 2010, s. 34). Preto Fleming at al. navrhli rozšírenie notácie sekvenčného diagramu pre viacvláknové aplikácie. Toto rozšírenie je demonštrované na Obr. č. 3.

Obr. č. 3 Viacvláknový sekvenčný diagram



Zdroj: Fleming et al (2010, s. 35)

3.4 Aspektovo orientovaná paradigma

Aspektovo orientovaná paradigma nadväzuje na objektovo orientovanú paradigmu. Usiluje sa oddeľovať od seba (modularizovať – kde modulom je napríklad trieda) jednotlivé záležitosti (aspekty problému) a vyhnúť sa tak zapleteniu kódu (to je keď jeden modul obsahuje viaceré aspekty problému) a roztrúseniu kódu (to je keď

jeden aspekt problému je riešený vo viacerých moduloch a tým sa kód stáva neprehľadným). *"Tento prístup nadväzuje na objektovo-orientovanú paradigmu, t. j. podobne ako pri zmene z procedurálneho na objektovo – orientované programovanie nedá sa hovoriť o zlome paradigmy, ako ho definoval Thomas Kuhn"* (Vranič, 2008, s. 167).

Asymetrický prístup (najčastejšie používaný) vyčleňuje z tried pretínajúce sa záležitosti (záležitosti ktoré spôsobovali zapletenie, alebo roztrúsenie kódu) do modulov, ktoré sa nazývajú **aspekty** (sú ekvivalenty tried). V programe sa tak nachádzajú triedy a aspekty. Tieto aspekty obsahujú **videnia**, čo sú ekvivalenty metód. Videnia majú definované takzvané **bodové prierezy**, čo sú množiny bodov spájania. **Bod spájania** je miesto, kde sa má príslušné videnie vykonať (jeho kód tam vloží prekladač pred alebo po preklade). Toto miesto sa napríklad nachádza v triede, z ktorej bol daný aspekt vyčlenený. To, ktoré miesta v kóde sú exponovanými bodmi spájania, závisí od konkrétneho jazyka (napríklad exponovaným bodom spájania jazyka AspectJ je vstup do metódy, nie je cyklus ani vetvenie) Videnia majú (napríklad cez svoje parametre) prístup ku kontextu bodov spájania, takže vo svojom tele dokážu pracovať s týmto kontextom.

Ďalším prvkom aspektovo orientovanej paradigmy sú **medzitypové deklarácie**. Pomocou nich je možné pridávať členov existujúcim triedam bez zmeny ich zdrojového kódu.

Dá sa povedať, že aspektovo orientovaná paradigma **abstrahuje** skutočnú štruktúru programu.

Predmetom **vizualizácie** aspektovo orientovaného programu teda sú triedy, aspekty a ich videnia a to,, na ktoré konkrétne bodové prierezy sú jednotlivé videnia vtkané. Tiež ktoré prvky boli do tried pridané pomocou medzitypových deklarácií.

3.5 Funkcionálna paradigma

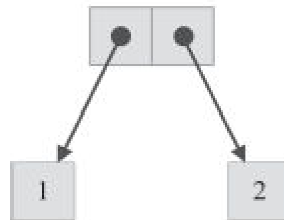
*"Funkcionálne programovanie je založené na myšlienke lambda – kalkulu, že každý algoritmus sa dá zapísať ako **reťazec funkcií** a ich redukciami a konverziami dostaneme výsledok"* (Vaníček et al., 2007, s. 259). Lambda kalkul (λ-kalkul) je matematický aparát, ktorý zaviedli Church a Keen v roku 1936 a ide v podstate o veľmi úsporný, ale univerzálny programovací jazyk. Jediné s čím pracuje je **funkcia jednej premennej** (funkciou sú napríklad aj prirodzené čísla a boolovské hodnoty). Parametrom tejto funkcie môže byť ďalšia funkcia. Dôležitú úlohu zohráva rekurzia.

Fukcia je anonymne definovaná pomocou takzvaného lambda-výrazu. Funkcie sú bezstavové - ich volanie nemá žiadny iný efekt, len ten, že vrátia výsledok. (Vaníček et al., 2007) *"Sila zápisu funkcionálneho programovania je daná predovšetkým dokonalou možnosťou dekompozície na elementárne funkcie"* (Vaníček et al., 2007, s 253).

Vhodným spôsobom **vizualizácie** by mohlo byť znázornenie reťazca funkcií pomocou stromu a znázornenie postupného vyhodnocovania jednotlivých funkcií prechodom tohto stromu v čase.

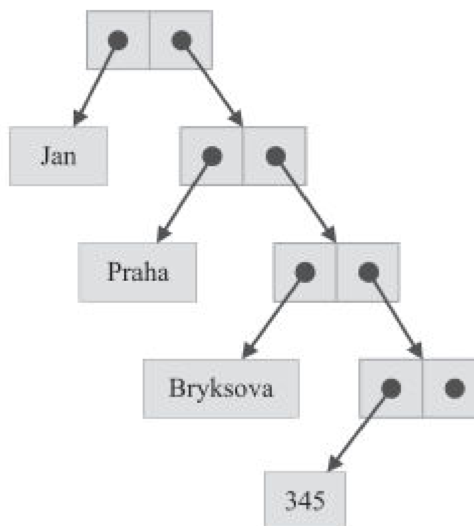
Impelementácie funkcionálnych programovacích jazykov však obsahujú aj nefunkcionále prvky, ako sú klasické jednoduché dátové typy, systémové zdroje a funkcie s vedľajšími efektmi. Najjednoduchší zložený dátový typ je dvojica nazývaná **bodka-dvojica**. Prvkom bodka-dvojice môže byť znova bodka-dvojica. Takto je možné vytvárať ľubovoľne zložité dátové štruktúry (analogicky tomu, že z jednoduchých funkcií je možné zostaviť celý algoritmus). (Vaníček et al., 2007).

Obr. č. 4 Dátový typ bodka-dvojica



Zdroj: Vaníček et al. (2007. s 256)

Obr. č. 5 Zoznam vytvorení z bodka-dvojíc



Zdroj: Vaníček et al. (2007. s 257)

Prvky funkcionálnej paradigmy (napríklad, že parametrom funkcie môže byť iná funkcia (Pascal), rekurzia,...) sa používajú aj v jazykoch založených na iných paradigmách (Vaníček et al., 2007).

3.6 Logická paradigma

Napísať program v logickej paradigme znamená vymenovať (zapísať do databázy) známe znalosti (**fakty** a **pravidlá**) o výseku sveta, ktorý je predmetom skúmaní, pomocou formulí predikátovej logiky. Používateľ potom zadá ako vstup otázku, na ktorú dostane odpoveď áno / nie, alebo dostane zoznam hodnôt premenných, pre ktoré je odpoveď áno. Samotné odvodenie odpovede zo znalostí nie je úlohou programátora. (Vaníček et al., 2007) *"V "čistých" logických programovacích jazykoch je zotrený rozdiel medzi programom a dátami"* (Vaníček et al., 2007, s. 282). To teda znamená, že logická paradigma **abstrahuje postup riešenia problému** v čase (abstrahuje celú dynamiku programu).

Predmetom **vizualizácie** programu založeného na logickej paradigme sú teda fakty, pravidlá a zadaná otázka. Vizualizovať by sa dal aj postup odvodenia, ten však nie je popísaný samotným programom.

Avšak platí že: *"Prakticky používané logické programovacie jazyky spravidla umožňujú programovať vstupy dát, výstupy dát i vykonávať niektoré zásahy do automatického odvodzovania dôsledkov prostriedkami prevzatými z imparatívnych programovacích jazykov"* (Vaníček et al., 2007, s. 283).

4 Monitorovanie dynamiky programu

Existuje viacero spôsobov ako sledovať čo sa deje v programe, ktorého vykonávanie chceme vizualizovať (Nedecký, 2010, In: Diehl 2007):

- Pomocou takzvaných **zaujímavých udalostí**: Do kódu monitorovaného programu vložíme na miesta, ktorých dosiahnutie chceme sledovať kód, ktorý nám odošle správu (vytvorí udalosť) o tom, že program dosiahol daný bod. Súčasťou odoslanej správy môžu byť aj parametre (napríklad hodnoty parametrov volanej metódy, ktorej volanie vložením zaujímavej udalosti na začiatok jej tela). Modifikácia kódu monitorovaného programu je veľkou nevýhodou tohto prístupu. Okrem manuálneho vkladania zdrojového kódu (alebo makra) zaujímavej udalosti do zdrojového kódu programu je možné využiť **prostriedky aspektovo orientovaného programovania**, ktoré rieši práve oddelenie jednotlivých záležitostí na úrovni zdrojového kódu. Je tiež možné nemodifikovať zdrojový kód, ale až skompilovaný binárny súbor.
- **Deklaratívne**: Kód monitorovaného programu nie je nutné modifikovať. Toto sa dá dosiahnuť napríklad takzvaným stavovým mapovaním - démon sleduje zmeny stavu vykonávaného programu a notifikuje vizualizátor.
- Použitím **interpretera,** ktorý priamo interpretuje zdrojový kód programu. Takto má úplné a do najmenších podrobností detailné informácie o vykonávaní programu, keďže ho sám vykonáva. Zároveň ho tiež môže vizualizovať, vtedy ho nazávame vizuálny interpreter. Tento spôsob sa tiež označuje ako sémantikou riadený spôsob.
- Použitím **špeciálnych dátových typov** pri programovaní programu, ktorý budeme chcieť vizualizovať. Tieto špeciálne dátové typy majú v sebe zabudovanú podporu pre monitorovanie (vizualizáciu). Ak napríklad použijeme dátový typ, ktorým je trieda, táto môže mať konštuktor, deštruktor, telá metód a prístupových metód (vlastností) upravené tak, aby správy o ich vyvolaní odosielali (vizualizovali).

5 Komunikácia medzi monitorovaným programom a vizualizátorom

Pokiaľ vizualizačný program (vizualizátor) neinterpretuje priamo monitorovaný program, ale tento je vykonávaný v samostatnom procese, je potrebné prenášať informácie o udalostiach čo nastali v procese monitorovaného programu do procesu vizualizačného programu.

*„Existujú štyri základné spôsoby integrácie systémov, ktoré sa niekedy nazývajú aj integračné štýly“ (*Šešera et al., 2011, s. 279, In: Hohpe et al 2004):

- Zdieľanie dát v databáze
- Prenos dát pomocou súborov
- Priama komunikácia posielaním správ medzi systémami.
- Priama komunikácia volaním procedúry v jednom systéme druhým systémom, ktorá sa nazýva Vzdialené volanie procedúry (anglicky Remote procedure call, RPC). Objektovo – orientovaná verzia sa nazýva **Vzdialené volanie metódy** (anglicky Remote Method Invocation, RMI). Detaily komunikácie medzi systémami (sieťovú komunikáciu, keďže systémy (rovnako ako aj pri ostatných integračných štýloch) vo všeobecnosti môžu byť distribuované v počítačovej sieti) v skutočnosti zabezpečuje midlvér (medzi najznámejšie patria CORBA, DCOM, **.NET Remoting**, Java RMI). Obe komunikujúce systémy musia používať rovnaký midlvér (rôzne midlvéry nie sú medzi sebou kompatibilné).

Softvér vizualizujúce softvéry často využívajú prenos dát pomocou diskového súboru so **záznamom** vytvoreného počas monitorovania programu. Tento súbor je následne v inom čase vizualizovaný vizualizátorom. Diehl (2007) tento spôsob označuje ako “postmortem” (posmrtný).

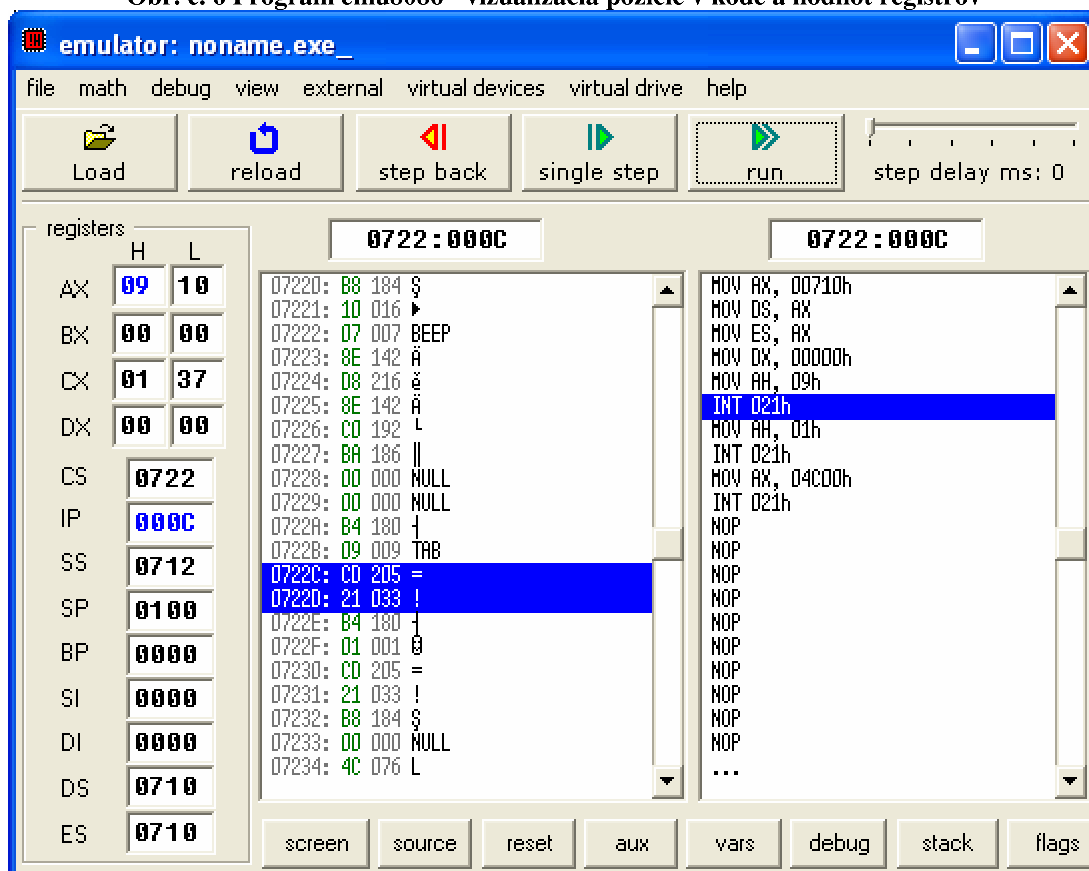
Veľkou výhodou **priamej komunikácie medzi procesmi** monitorovaného programu a vizualizačného programu technikou RMI je, že takto je možné vizualizovať čo sa práve (v reálnom čase) v programe odohráva. Používateľ môže do vizualizovaného programu priamo zasahovať (napríklad ovládať jeho tok riadenia, ale prípadne aj hodnoty dátových položiek) a rozhodovať sa na základe jeho reakcií. .NET Framework už od verzie 1.0 za týmto účelom poskytuje technológiu nazvanú .NET Remoting.

6 Niektoré existujúce programy vizualizujúce softvér

6.1 emu8086

Program emu8086 je vizuálny interpreter zdrojových kódov programov napísaných v jazyku symbolických inštrukcií (v asembléry) pre procesor Intel 8086, alebo kompatibilný, ktorý emuluje. Zo zdrojového kódu dokáže tiež zostaviť spustiteľný program.

Obr. č. 6 Program emu8086 - vizualizácia pozície v kóde a hodnôt registrov

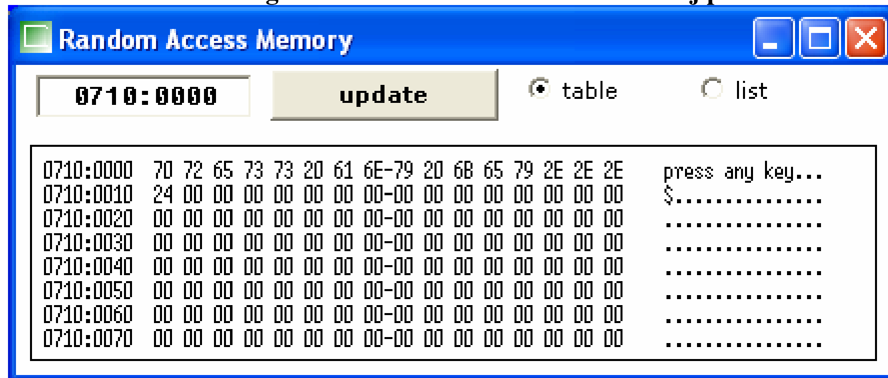


Obsahuje editor zdrojového kódu, ktorý následne interpretuje. Používateľ môže program krokovať o krok **dopredu**, o krok **dozadu**, alebo ho spustiť až do konca s prestávkami medzi každým krokom. **Trvanie prestávok** je možné nastaviť. Každý krok spôsobuje zmenu stavu programu. Pre každý stav sa **vo viacerých oknách** vizualizujú:

- Pozícia nasledujúcej inštrukcie v zdrojovom kóde
- Pozícia nasledujúcej inštrukcie v kódovom segmente

- Hodnoty univerzálnych registrov (AX - DX), indexových (SI, DI), zásobníkových (SP, BP), hodnota registera IP (instruction pointer – register obsahujúci adresu nasledovnej inštrukcie), hodnoty segmentových registrov. Pokiaľ v predchádzajúcom kroku **došlo k zmene** hodnoty niektorého z týchto registrov, je jeho hodnota zvýraznená modrou farbou.
- Obsah príznakového registra (anglicky flag)
- Hodnoty registrov aritmeticko – logickej jednotky (ALU)
- Hodnoty registrov jednotky pre výpočty v pohyblivej rádovej čiarke (FPU)
- Obsah hlavnej pamäte ako postupnosť bajtov a postupnosť znakov. Takto je možné zobraziť obsah dátového segmentu programu:

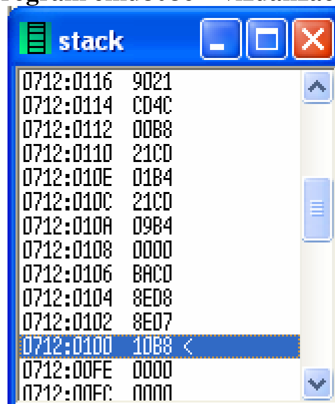
Obr. č. 7 Progaram emu8086 – vizualizácia hlavnej pamäte



Tiež je možné znázorniť zoznam pomenovaných premenných.

- Zásobník:

Obr. č. 8 Program emu8086 – vizualizácia zásobníka



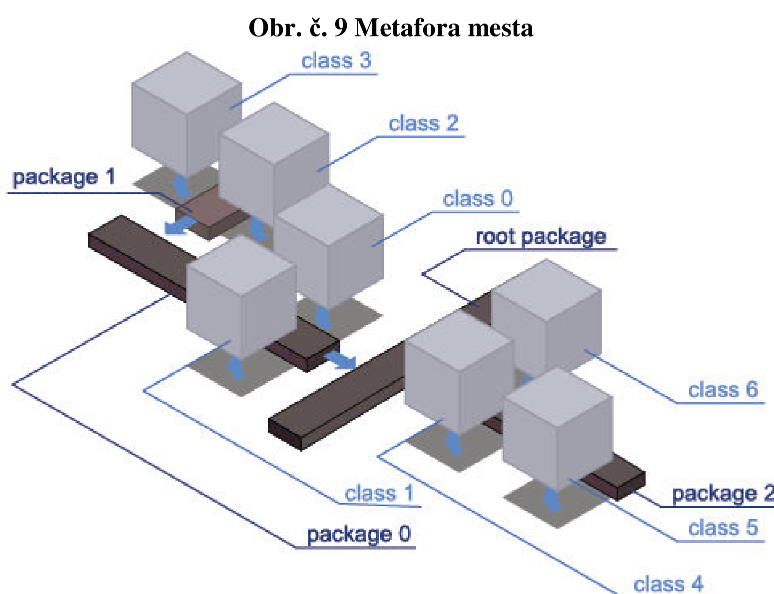
- V prípade potreby aj konzolové používateľské rozhranie.

Hodnoty registrov a pamäťových buniek je možné medzi jednotlivými krokmi nastaviť aj **interaktívnou zmenou zobrazovanej hodnoty**. Je možné nastaviť inštrukciu, na ktorej má vykonávanie programu zastaviť (anglicky **break point**).

6.2 Vizualizačný nástroj VITRAIL

Vizualizačný nástroj VITRAIL (Caserta – Zendra – Bodenes, 2011) používa pri zobrazení štruktúry softvéru v 3D priestore **metaforu mesta**. Autori tvrdia, že použitím tejto metafory je možné rýchlejšie rozpoznávať globálne softvérové štruktúry a lepšie a rýchlejšie pochopiť aj komplexným situáciám. Predíde sa tak najproblematickejšiemu aspektu 3D vizualizáci, ktorým je dezorientácia používateľa. Metafora využíva analógiu medzi organizáciou mesta do mestských štvrtí, ulíc a budov, s organizáciou objektovo – orientovaného programu na časti ako sú na balíky a triedy.

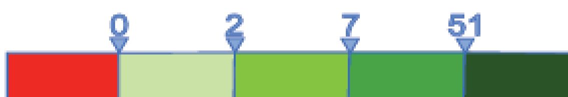
Obr. č. 9 Metafora mesta



Zdroj: Caserta – Zendra – Bodenes (2011)

Vzťahy medzi jednotlivými elementmi softvéru sú znázornené hranami (spojnicami) medzi nimi. Farba hrany vyjadruje kvantitu vzťahu, ktorá je klasifikovaná do 5 množín (viď Obr. č. 10). Vzťahom môže byť napríklad vzájomné volanie metód medzi triedami.

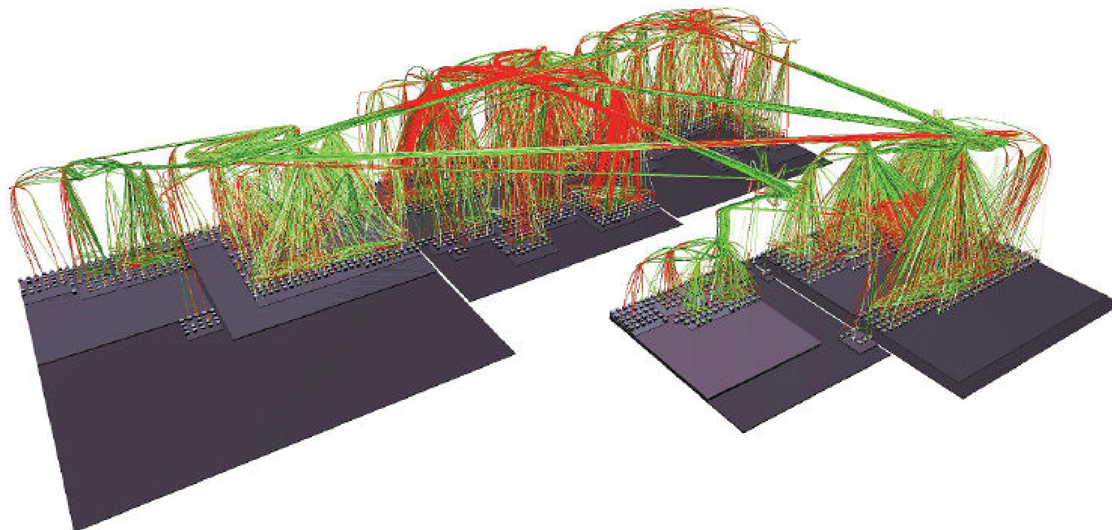
Obr. č. 10 VITRAIL – farebná stupnica pre hrany



Zdroj: Caserta – Zendra – Bodenes (2011)

Vizualizačnému nástroju VITRAIL je venovaná webová stránka: http://www.loria.fr/~casertap/visualizer.html.

Obr. č. 11 Vizualizačný nástroj VITRAIL



Zdroj: Caserta – Zendra – Bodenes (2011)

6.3 DJVis

DJVis je vizualizačný nástroj určený pre vizualizáciu rozsiahlych programov napísaných v jazyku Java. Vykonávanie programu monitoruje prostredníctvom technológie JPDA (Java Platform Debug Architecture), ktorá umožňuje program sledovať **bez nutnosti meniť jeho zdrojový kód**. (Smith – Munro, 2002)

DJVis poskytuje štandardné "debugovacie" rozhranie, ktoré umožňuje napríklad skok na nasledujúci riadok, spustenie programu a nastaviť "break point" (Smith – Munro, 2002).

K dispozícií ponúka viac vizualizačných pohľadov, ktoré sa zameriavajú na **rôzne aspekty** získaných informácií. Hlavné pohľady sú:

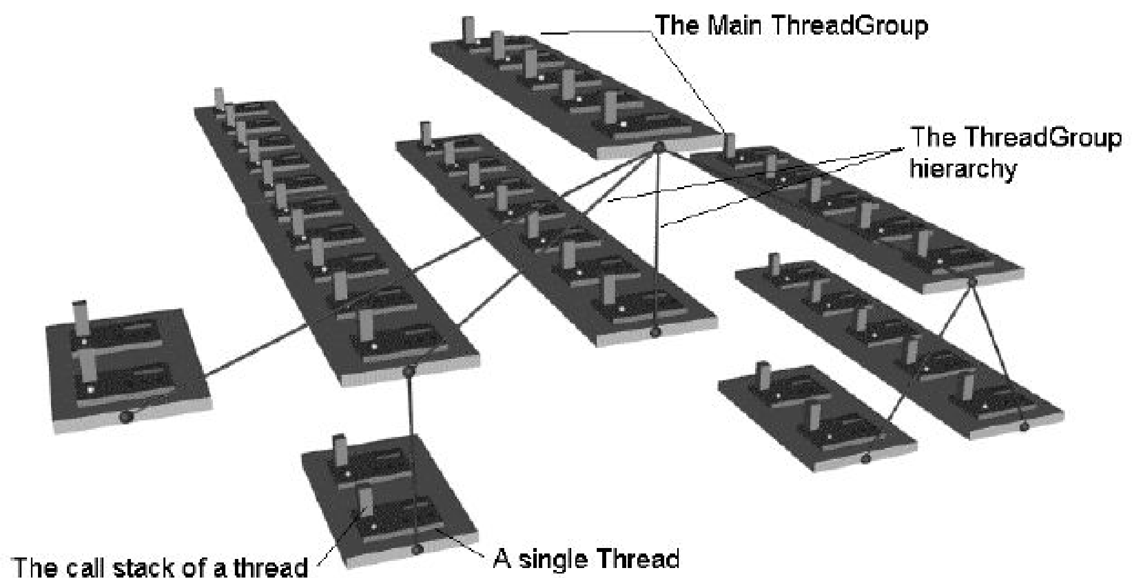
- Pohľad na beh programu (Runtime View)
- Pohľad na triedy (Class View)
- Dotazovací pohľad (Query View)

Ďalej poskytuje ďalšie pomocné pohľady, ktoré sú prepojené s hlavnými pohľadmi. (Smith – Munro, 2002)

Pohľad na beh programu sa zameriava na vizualizáciu udalostí na úrovni vlákien. Informuje o **volaniach** a argumentoch volaní. Zobrazuje skupiny vlákien a pomocou

stromového pohľadu hierarchickú štruktúru medzi nimi. V rámci každej skupiny vlákien, zobrazuje jednotlivé **vlákna** a informácie o nich. Napríklad blokovanie vlákna iným vláknom. Pre každé vlákno zobrazuje **zásobník volaní** (anglicky call stack) vo forme kvádra. Výška tohto kvádra indikuje počet metód na zásobníku. Zásobník je možné priblížiť a uvidieť tak detaily o jednotlivých metódach. Pohľad na beh programu je možné prispôsobiť tak aby bolo možné vidieť volania jednotlivých tried a tak ľahko určiť, ktoré vlákna vykonávajú kód používateľa a ktoré len kód aplikačného rozhrania (API) Javy. (Smith – Munro, 2002)

Obr. č. 12 Program DJVis – Pohľad na beh programu



Zdroj: Smith - Munro (2002)

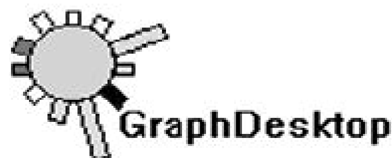
<u>Pohľad na triedy</u> zobrazuje **triedy** nachádzajúce sa v programe a **vzťahy** medzi nimi vo forme grafu, ktorý sa skladá z uzlov a hrán:

- Uzly znázorňujú triedy. Ich číselné ohodnotenie a ofarbenie umožňuje reprezentovať informácie o **počte inštancií** danej triedy v programe, ktoré boli vytvorené buď od spustenia programu, alebo od určitého okamihu. Tiež je možné zobraziť **poradové číslo načítania** triedy.

  Trieda ja vizualizovaná ako kruh, okolo ktorého sú rozmiestnené stĺpce reprezentujúce jednotlivé metódy. Takto je viditeľný **počet metód triedy**. Dĺžka stĺpca a jeho farba môže reprezentovať **dĺžku zdrojového kódu, zložitosť,**

počet volaní a prístupové práva metódy. Na Obr. č. 13 dĺžka stĺpcov znázorňuje dĺžku metód a ich farba znázorňuje počet volaní príslušných metód:

Obr. č. 13 Program DJVis – vizualizácia tridy GraphDesktop

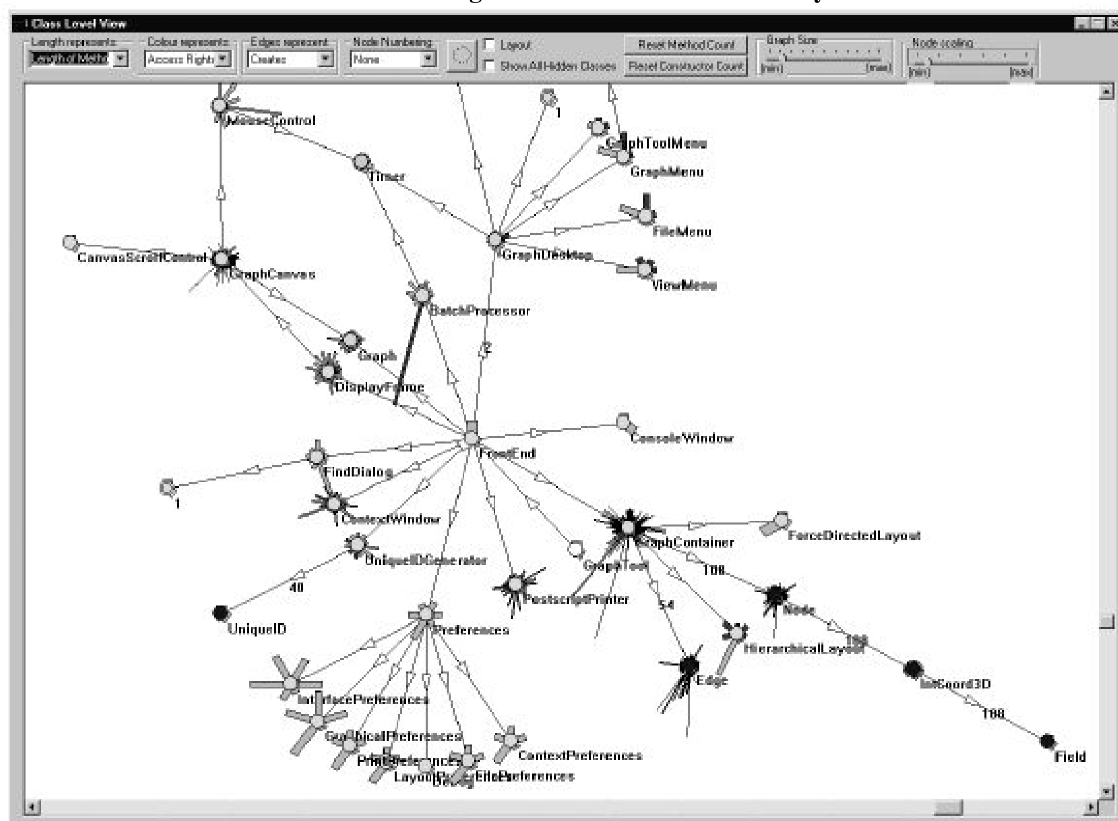


Zdroj: Smith - Munro (2002)

- Hrany zobrazujú vzťahy medzi triedami ako je napríklad ich vzájomné referencovanie sa a vzťahy týkajúce sa **vytvárania** tried. (Smith – Munro, 2002)

Pohľad je možné prispôsobiť požiadavkám používateľa tak aby vizualizoval ním požadované typy informácií.

Obr. č. 14 Program DJVis – Pohľad na triedy



Zdroj: Smith - Munro (2002)

Dotazovací pohľad umožňuje používateľovi premiestniť jednotlivé elementy z iných pohľadov do dotazov a ľahko tak vytvárať dotazy za cieľom získať o jednotlivých elementoch viac informácií. Používateľ sa tak môže zamerať napríklad na dve konkrétne vlákna, alebo len určité triedy. (Smith – Munro, 2002)

6.4 Visualization of program execution in 3D environment

Tento systém dokáže vizualizovať vykonávanie ľubovoľného programu napísaného v jazyku Java. Monitorovanie vykonávania vizualizovaného programu nie je súčasťou vizualizačného programu. Namiesto toho je najprv nutné **manuálne** pomocou kompilátoru jazyka AspectJ **vtkať** spolu so systémom **dodané aspekty**, čím vznikne nový .jar súbor a tento je následne potrebné spustiť. So spusteným programom vykonáme tie operácie ktoré budeme chcieť následne vizualizovať a program ukončíme. Vtkané aspekty zabezpečili zaznamenanie prevedených operácií do súboru vo formáte .xml. Tento XML súbor je vstupom pre vizualizátor. (Nedecký, 2010)

Program vizualizuje tieto udalosti: volanie metódy, vrátenie metódy, vytvorenie inštancie triedy a volanie konštruktora, ukončenie vykonávania konštruktora, nastavenie poľa a čítanie poľa (Nedecký, 2010).

Program vizualizuje entity, ktoré sa zúčastnili zaznamenaného vykonávania programu. **Triedu** alebo **objekt** zobrazuje pomocou špirály. Na vrchole špirály sa nachádza guľový vrchol, ktorý je žltej farby v prípade triedy, tyrkysovej v prípade objektu. Špirála pokračuje postupnosťou menších oranžových gúľ, ktoré predstavujú **metódy** objektu, alebo **statické metódy** triedy. Nasleduje postupnosť zelených kociek, ktoré reprezentujú **polia** objektu, alebo **statické polia** triedy. Pri dostatočnom vzdialení sa od špirály sa bude zobrazovať len jej vrchol, nie členy. Výhodou použitia špirály, je že jednotlivé položky sa neprikrývajú a špirála rozdeľuje priestor na dva podpriesotry: vnútorný podpriestor, v ktorom je vizualizovaná vnútorná komunikácia medzi položkami a vonkajší podpriestor, v ktorom je vizualizovaná komunikácia medzi položkami rôznych špirál. (Nedecký, 2010)

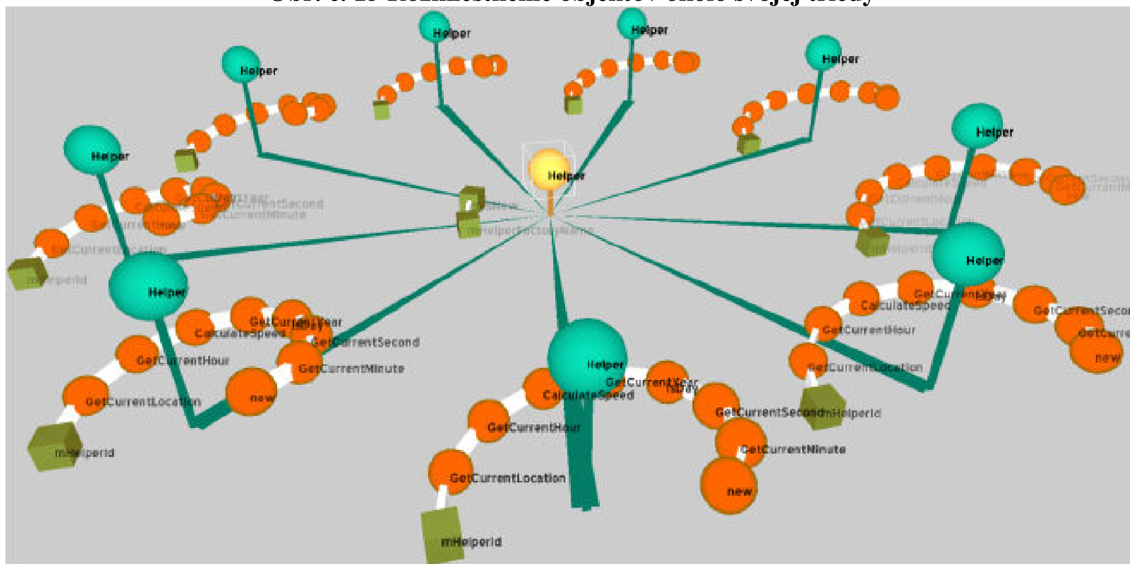
Po spustení programu sa na scéne rozmiestnia všetky triedy, ktoré budú zobrazované. Sú rozmiestnené podľa toho, **ako často medzi sebou komunikovali**. Tie, ktoré komunikovali častejšie sú umiestnené bližšie pri sebe. Následne sa v okolí týchto tried počas behu programu vykresľujú ich jednotlivé inštancie, podľa toho, **kedy boli vytvorené.** (Nedecký, 2010).

Správy sú vizualizované plynulou animáciou prechodu entity reprezentujúcej správu (na Obr. č. 16 je reprezentovaná modrou guľôčkou) po dopredu vykreslenej spojnici medzi odosielateľom a prijímateľom (Nedecký, 2010).

Záznam je možné prehrávať smerom dopredu, dozadu, dopredu len jednu udalosť, dozadu len jednu udalosť a **pretočiť** ho na nejakú konkrétnu udalosť.

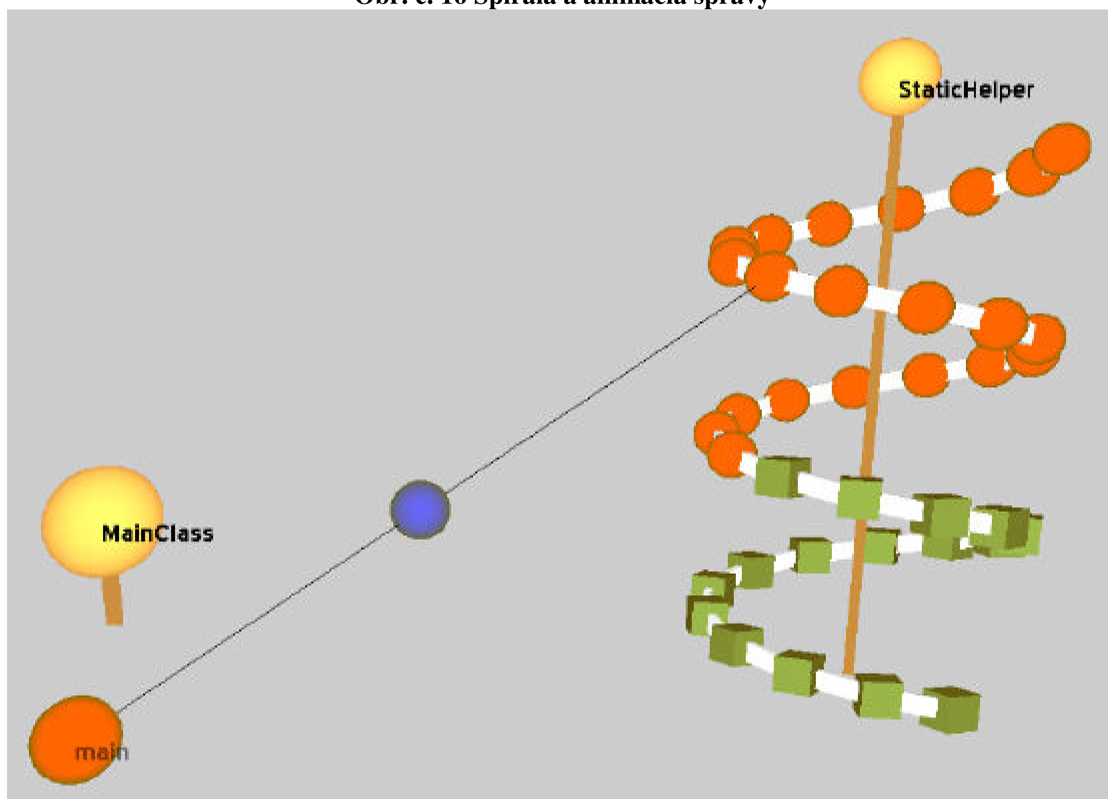
Program tiež zobrazuje **zoznam** udalostí v systéme, zoznam dotknutých tried a štruktúry a podrobný popis vybratého elementu (Nedecký, 2010).

Obr. č. 15 Rozmiestnenie objektov okolo svojej triedy



Zdroj: Nedecký (2010. s 48)

Obr. č. 16 Špirála a animácia správy



Zdroj: Nedecký (2010. s 49)

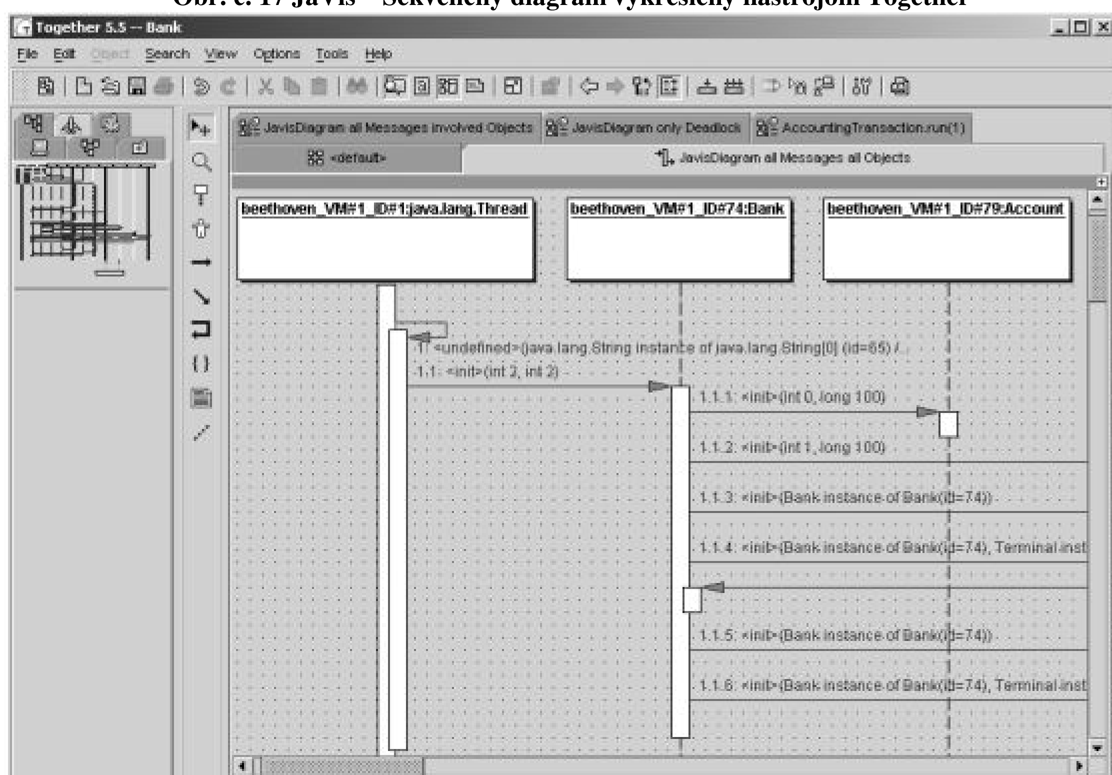
6.5 JaVis

Nástroj JaVis (Mehner, 2002) generuje klasické UML 2D diagramy z bežiacich programov napísaných v jazyku Java. Dôraz kladie na vizualizáciu viacvláknových programov. Skladá z dvoch komponentov:

- Trasovací komponent
- Vizualizačný komponent

Trasovací komponent získava trasovacie informácie o vykonávaní programu napísaného v jayku Java a detekuje uviaznutia (anglicky deadlock). Využíva Java Debug Interface (JDI) implementované behovým prostredím platformy Java - Java Virtual Machine (JVM). JDI umožňuje získavať debugovacie a trasovacie informácie počas behu programu napísaného v jazyku Java bez potreby modifikovať jeho zdrojový kód.

Obr. č. 17 JaVis – Sekvenčný diagram vykreslený nástrojom Together



Zdroj: Mehner (2002)

Vizualizačný komponent po skončení vykonávania sledovaného programu (**post-mortem**) najprv analyzuje získané trasovacie dáta a následne z nich vygeneruje UML diagramy. Trasovacie dáta neobsahujú informáciu o volanej metóde, nie však o volateľovi tejto metódy. Za účelom získania volateľa je nutné simulovať zásobník volaní (anglicky call stack) pre každé vlákno. JaVis **diagramy nevykresľuje priamo**,

ale prostredníctvom UML CASE nástroja Together (viď Obr. č. 17), ktorý ich umožňuje definovať cez svoje otvorené aplikačné rozhranie (API). Samotný nástroj Together dokáže síce získavať informácie o behu programu, obsahuje však len slabú podporu pre vizualizovanie viacvláknových programov, preto za účelom tejto podpory bol vyvinutý nástroj JaVis.

Na Obr. č. 17 vidíme v ľavej časti **celkový pohľad** na vygenerovaný sekvenčný diagram a vpravo vybratú veľmi malú časť z neho zobrazujúcu inicializačnú fázu s mnohými volaniami konštruktorov.

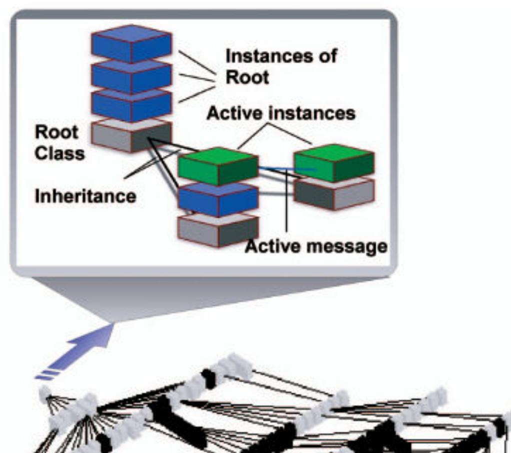
Zobrazené objekty v sekvenčnom diagrame sú identifikované úplnou cestou (obsahujúcou názvy balíkov) k ich triede a ID odvodeným mimo iného aj od vnútorného objektového ID daného objektu, ktoré mu pridelil JVM. Za účelmi vizualizácie synchronizácie používa JaVis v diagramoch stereotypy <<acquire>> a <<lock>>, ktoré sú však nástrojom Together vykreslené formou jednoduchého textového popisu, nie ako nové formálne stereotypy.

6.6 Existujúce prehľady ďalších nástrojov

Rozsiahlejší prehľad nástrojov vizualizujúcich softvér rozdelených do kategórií podľa typu vizualizácie (vizualizácia štruktúry, dynamiky a evolúcie softvéru) podáva Diehl (Diehl, 2007). Na konci publikácie tiež (strany 163 – 165) uvádza zoznam nástrojov, ktoré nepopísal podrobnejšie.

Práca (Teyseyre – Campo, 2009) tiež obsahuje na strane 97 prehľadovú tabuľku existujúcich 3D vizualizačných nástrojov.

Obr. č. 18 TraceCrawler



Zdroj: Teyseyre – Campo (2009, s. 98)

Nedecký vo svojej práci (Nedecký, 2010) podrobnejšie popisuje nástroje: SAM, JOOPA2, InspectJ.

7 Návrh riešenia

V tejto kapitole najprv popisujeme nami navrhnutý **spôsob vizualizácie** dynamiky objektovo – orientovaného programu, (ktoré prvky programu vizualizovať, ako ich reprezenovať a kedy ich zobrazovať) a následne návrh architektúry a jednotlivých komponentov (vrátane návrhu ich objektových štruktúr a niektorých algoritmov) nami navrhnutého a implementovaného **vizualizačného systému**, ktorý sme nazvali SoftDynamik.

7.1 Koncept vizualizácie

V tejto časti popisujeme základné myšlienky predkladaného spôsobu vizualizácie dynamiky objektovo – orientovaného programu, pričom ako referenčný programovací jazyk používame jazyk C#.

Predkladaná vizualizácia je animácia bežiaca v 3 - rozmernom priestore a 1 - rozmernom čase, stojaca na 4 základných pilieroch:

1. Vizualizácia stromu vytárania objektov – pohľad na scénu zhora
2. Vizualizácia vlákien – pohľad na tú istú scénu zospodu (prípadne zboku).
3. Vizualizácia časového kontextu – pohľad na tú istú scénu zboku.
4. Dynamické obmedzenia priestoru a času vizualizácie.

Používateľ sa môže v scéne voľne pohybovať a ľubovoľne ju natáčať.

7.1.1 Vizualizácia stromu vytvárania objektov

Čiastočnú postupnosť vytvárania objektov zobrazujeme orientovaným acyklickým grafom (stromom) zobrazujúcim objekty nachádzajúce sa v programe a ich rodičovské vzťahy. Dcérsky objekt a rodičovský objekt sú spojené orientovanou hranou. Táto hrana je orientovaná od rodičovského objektu k dcérskemu objektu. Koreňom stromu je prvý objekt programu (ktorý priamo alebo cez viaceré generácia objektov vytvoril všetky objekty v programe). Takýto objekt v tejto práci nazývame *jadrový objekt*. (V programoch jazyka C# je to vždy "statický objekt" Program.). Pri pohľade zhora je dcérsky objekt nejakého rodičovského objektu zobrazený naľavo od svojho staršieho brata. Najpravejší nasledovník bol tak vytvorení najskôr, najľavejší najneskôr.

K zobrazeniu tohto grafu využívame horizontálne priestorové dimenzie (x - ová a y - ová) (**pohľad zhora**) Takto je definované **rozmiestnenie objektov** v scéne. Existuje viac možnosti ako tieto objekty rozmiestniť, napríklad zgrupovať objekty

rovnakej triedy do jednej oblasti priestoru, alebo objekty, ktoré spolu komunikujú najviac dať k sebe bližšie. Tento spôsob rozmiestnenia sme zvolili z nasledujúcich dôvodov:

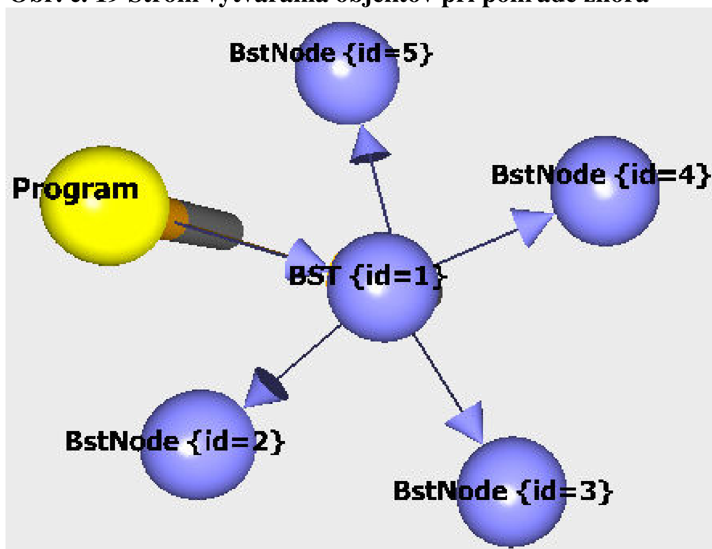
- Poskytuje prídavnú informácie o rodičovských vzťahoch medzi objektmi.
- Rodičovské objekty vlastnia referencie na dcérske objekty, preto je pravdepodobné, že najviac komunikácie bude práve medzi rodičovským objektom a dcérskym objektom (konkrétne, že rodič bude volať metódy dcér).
- Pokiaľ dôjde k volaniu metódy, alebo nastaveniu poľa (členskej premennej) iným než rodičovským objektom, indikuje to použitie zložitejšej programátorskej techniky, napríklad návrhového vzoru.

Za účelom tohto horizontálneho rozmiestnenia objektov v scéne sme navrhli a implementovali **algoritmus**, ktorý popisujeme v kapitole 7.7.1.

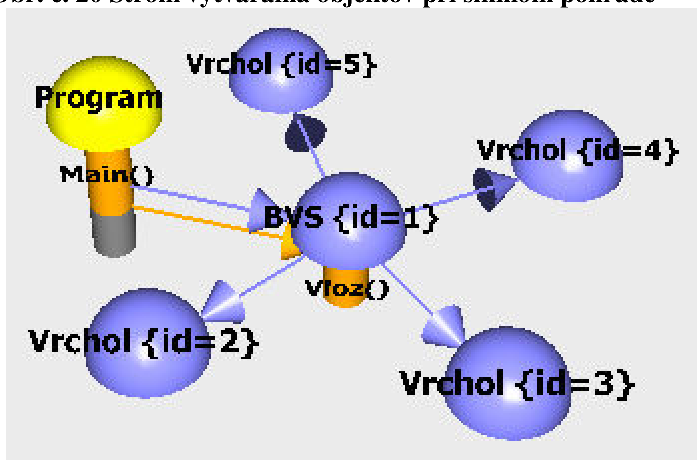
V jazyku C# (rovnako aj v jazyku C++, Visual Basic a Java), sa vyskytujú takzvané **statické prvky tried** (metódy a polia), ktoré nepatria žiadnej jej inštancií (žiadnemu objektu), ale sú nimi zdieľané. Túto podmnožinu triedy, ak je neprázdna, vizualizujeme ako samostatný objekt, keďže má viaceré znaky objektu (napríklad má vlastný konštruktor, obsahuje metódy a polia, jej vytvorenie zapríčiňuje iný objekt). Pre účely tejto vizualizácie nazvime túto podmnožinu *statickým objektom*. Týmto navrhovaná vizualizácia zostáva nezávislá od tohto prvku jazyka.

Objekty vizualizujeme ako **polgule** (presnejšie povedané tieto polgule reprezentujú objekt ako celok, sú to hlavice objektov). Statické objekty odlišujeme od inštančných objektov farbou pologule. Obvod rezu polgule vymedzuje ***priestor vnútornej komunikácie objektu***.

Obr. č. 19 Strom vytvárania objektov pri pohľade zhora



Obr. č. 20 Strom vytvárania objektov pri šikmom pohľade



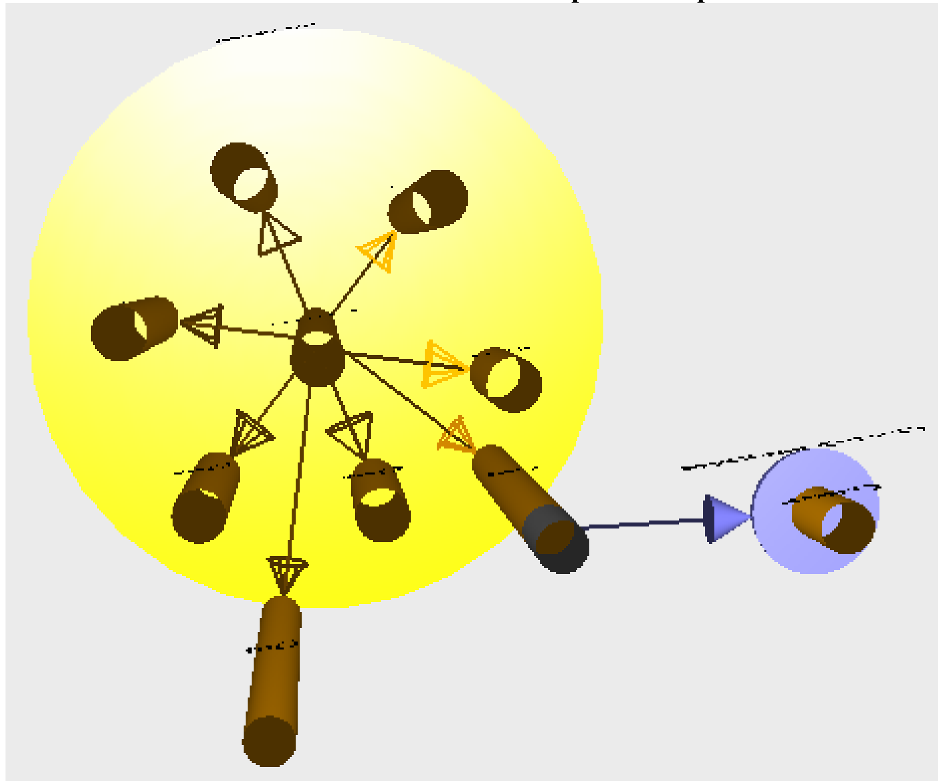
7.1.2 Vizualizácia vlákien

Pre účely tejto vizualizácie zaveďme pojem *objekt-vlákno*, ktorým budeme označovať časť vlákna (anglicky thread), ktorá je vykonávaná nad určitým objektom. Objekt-vlákno je postupnosť vykonávania metód a pristupovania k členským premenným (v jazyku C# nazvaných polia) v rámci konkrétneho vlákna a objektu. Objekt-vlákno vlákna, v ktorom bol objekt vytvorený nazvime *jadrové objekt-vlákno* objektu.

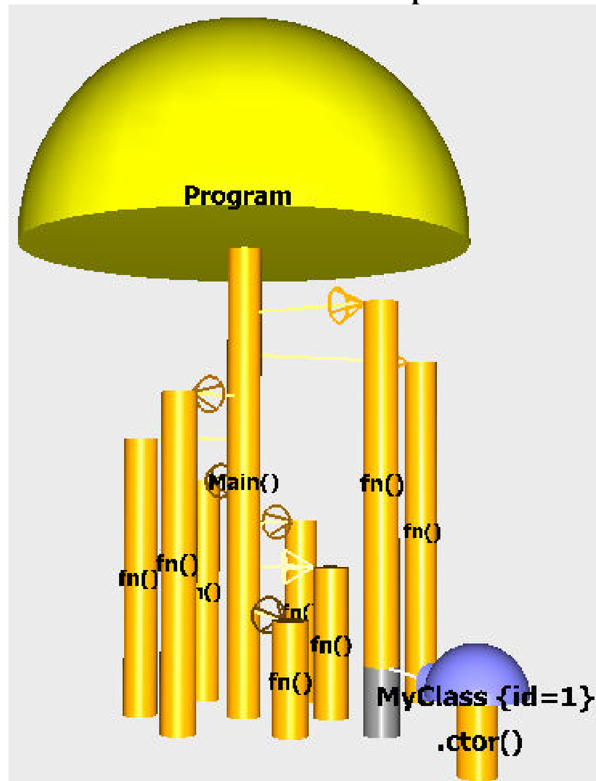
Objekt-vlákno je vizualizované v priestore vnútornej komunikácie príslušného objektu. Jednotlivé objekt-vlákna sú v rámci objektu rozmiestnené horizontálne (v dimenziách x, y). Pri návrhu ich rozmiestnenia sme využili **metaforu atómu**. Jadrové objekt-vlákno sa nachádza v strede priestoru vnútornej komunikácie objektu (je umiestnené do stredu objektu). Ostatné objekt-vlákna (preriférne objekt-vlákna) sú rozmiestnené na presne definovaných dovolených pozíciach v okolí jadrového objekt-vlákna. Tieto dovolené pozície sú usporiadané do niekoľkých sústredných kružníc (so stredom v strede objektu), ktoré nazývame vrstvy. V každej vrstve je maximálne 6 objekt-vlákien, čo umožňuje aby boli vzájomné vzdialenosti medzi objekt-vláknami rovnakej vrstvy a tiež ich vzdialenosti k jadrovému objekt-vláknu rovnaké (keďže rovnostranný trojuholník má uhly o veľkosti 60° a takýchto uhlov je v plnom kruhu práve 6). Po tom čo sú všetky voľné pozície na vrstve zaplnené pridá sa vyššia vrstva. Za účelom výpočtu horizontálnej polohy objekt-vlákien sme navrhli a implementovali algoritmus, ktorý popisujeme v kapitole 7.7.2.

Vzájomné vzťahy medzi jednotlivými objekt-vlákami sú dobre viditeľné **pri pohľade zospodu**.

Obr. č. 21 Vizualizácia vlákien – pohľad zospodu



Obr. č. 22 Vizualizácia vlákien – pohľad z boku



7.1.3 Vizualizácia časového kontextu

Vizualizácie dynamiky programu je možné rozdeliť do dvoch skupín:

- Vizualizácie bez časového kontextu (príkladom je nástroj *Visualization of program execution in 3D environment* popísaný v kapitole 6.4). Zobrazujú štruktúru programu a len jednu práve posielanú správu.
- Vizualizácia s časovým kontextom. **Výhodou** tohto typu je, že umožňuje v jednom časovom okamihu vidieť viac kľúčových detailov programu. Napríklad:
    - kedy a kto vytvoril objekt, nad ktorým beží práve bežiaca metóda
    - celý reťazec volaní od jadrového objektu, ktorá spôsobila vyvolanie aktuálne bežiacej metódy.
    - že ide o rekurziu
    - celú cestu šírenia výnimky
    - časti programu podobajúce sa návrhovým vzorom
    - čo bude nasledovať v budúcnosti (napríklad, že bude pokračovať vykonávanie metód, ktoré ešte neskončili).

K vizualizácií časového kontextu sme využili vertikálnu priestorová dimenzia (z – ovú dimenziu), ktorá predstavuje čas. Časový kontext je viditeľný **pri pohľade z boku.**

V smere proti smeru súradnice z sú postupne umiestňované a naťahované **metódy** (presnejšie ich vykonávania), ktoré sú reprezentované valcami. Metódy sú v horizontálnych dimenziách umiestnené v zodpovedajúcom objekt-vlákne, ktorému prislúchajú.

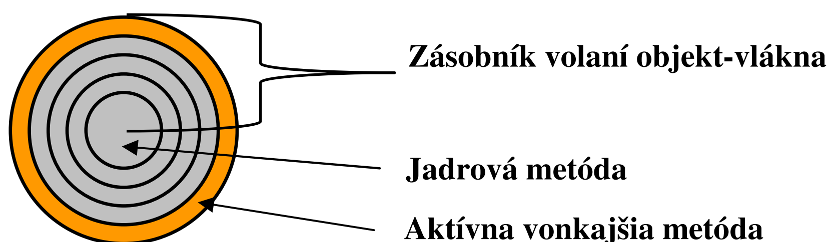
Tá časť metódy, ktorá bola v zodpovedajúcom čase aktívna (bola na vrchole zásobníka volaní daného vlákna) je zobrazená farebne. **Neaktívne časti metódy** (zodpovedajúce času keď metóda ešte nebola ukončená – čakala na ukončenie metódy, ktorú zavolala) sú zobrazené predvolene šedou farbou (farby môže nastavovať používateľ).

Metóda, ktorá sa začala vykonávať súčasne s vytvorením objektu (spravidla **konštruktor**) sa dotýka roviny rezu polgule reprezentujúcej daný objekt.

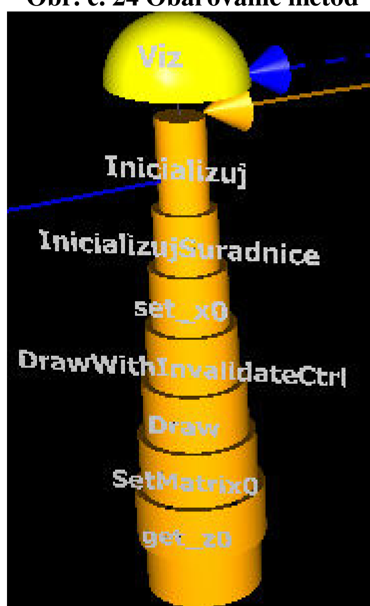
V jednom časovom okamihu môže byť nad jedným objektom a v rámci jedného vlákna (v rovnakom objekt-vlákne) spustených viac metód. Prvú spustenú metódu budeme nazývať *jadrová metóda*. Následne spúšťané metódy obaľujú túto metódu, podobne ako letokruhy stromu obaľujú staršie letokruhy (**metafora letokruhov**).

Aktívna môže byť (nemusí) vždy len *vonkajšia metóda*. Rez týmito metódami zobrazuje zásobník volaní daného objekt-vlákna.

Obr. č. 23 Rez obaľujúcimi sa metódami objektu – metafora letokruhov



Obr. č. 24 Obaľovanie metód

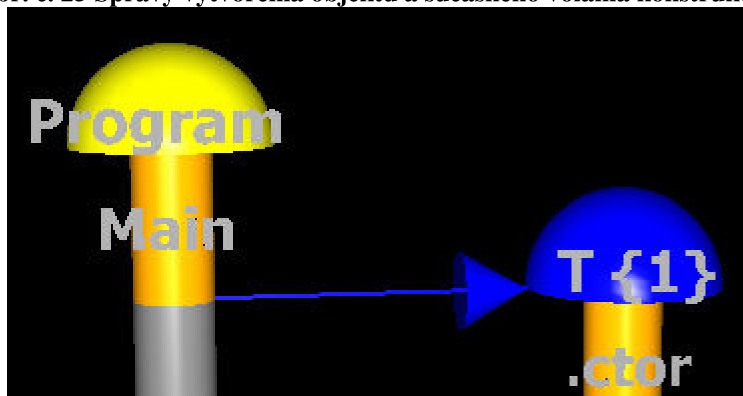


Prístup k premenným (čítanie, zápis) predstavuje jediný časový okamih. Kvôli lepšej viditeľnosti je však vizualizácia daného poľa (členskej premennej) realizovaná zobrazením kvádra, umiestneného v dimenziách x, y na rovnakej pozícií ako jemu prislúchajúci objekt. Kváder prípadne obaľuje metódy tohto objektu.

Komunikáciu (**posielanie správ**) medzi prvkami programu reprezentujeme na príslušnej z-ovej súradnici zobrazením šípky. Smer šípky vyjadruje smer odovzdávania riadenia a dát. Farba šípky určuje typ správy. Rozlišujeme nasledovné typy správ a ich predvolené farby:

- Vytvorenie objektu – modrá farba
- Vytvorenie objektu a súčasné volanie metódy (konštruktora) – modrá farba:

Obr. č. 25 Správy vytvorenia objektu a súčasného volania konštruktora



- Volanie metódy – oranžová farba
- Čítanie poľa – zelená farba. Šípka smeruje od poľa k metóde.
- Zápis do poľa – zelená farba. Šípka smeruje od metódy k poľu.
- Vrátenie návratovej hodnoty (vrátane typu void) – šedá farba
- Vyhodenie výnimky – červená farba.

Farby môže používateľ zmeniť.

Synchrónne a asynchrónne správy sú odlíšené tvarom hrotu. Synchrónne správy majú hrot zobrazený ako plný povrch kužeľa. Asynchrónne len ako naznačenie povrchu kužeľa, čím sa vyjadruje, že dochádza len k čiastočnému odovzdaniu riadenia.

Obr. č. 26 Vizualizácia synchrónnej správy

Obr. č. 27 Vizualizácia asynchrónnej správy

Nepriame správy sú znázornené prerušovanou čiarou. *Nepriama správa* je použitá ak odosielateľ neposlal správu priamo adresátovi, ale urobil tak cez ďalších (neidentifikovaných) účastníkov (môže ísť napríklad o metódu definovanú mimo zdrojový kód monitorovaného programu), respektíve pri vrátení návratovej hodnoty,

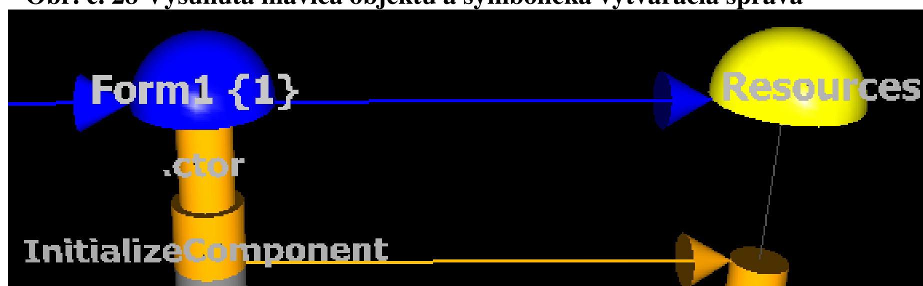
alebo vyhodení výnimky ju nevrátil adresátovi, ale neidentifikovanému účastníkovi predtým volanému adresátom.
Šípky správ zasielaných v rámci toho istého objektu a vlákna (objekt-vlákna) nie sú vykresľované.

7.1.4 Dynamické obmedzenia priestoru a času vizualizácie

Zobrazovať v jednom okamihu celú históriu dynamiky programu by mohlo skončiť veľmi veľkou rozsiahlosťou a neprehľadnosťou celej vizualizácie. Preto bolo nevyhnutné poskytnúť používateľovi možnosť zvoliť si automatický vizualizačný mód, v ktorom sa budú zobrazovať len určité časti histórie dynamiky – tie, ktoré pravdepodobne obsahujú najdôležitejšie informácie. Preto súčasťou predkladanej vizualizácie sú navzájom nezávislé dynamicky voliteľné možnosti nasledovných obmedzení:

- **Zásobníkový mód**:
  Po zapnutí tohto módu sa všetky následne zavolané metódy po skončení zo scény odstránia. Polia sú odstraňované hneď potom ako boli znázornené. Ak bol tento mód zapnutý počas celej vizualizácie, tak každý krok (stav) vizualizácie **vizualizoval len obsahy zásobníkov volaní** vlákien programu. Po vypnutí tohto módu sa už metódy zo scény odstraňovať nebudú.
  Toto obmedzenie obmedzuje z – ovú dimenziu.
  V prípade, že bola zo scény odstránená metóda, ktorá vytvorila nový objekt (odoslala správu vytvárajúcu objekt) tak sa **hlavica vytvoreného objektu vysunie** z úrovne zodpovedajúcej okamihu vytvorenia objektu na úroveň hlavice objektu, ktorý obsahoval odstránenú metódu (teda na úroveň hlavice rodičovského objektu) a vykreslí sa symbolická správa vytvorenia objektu ako šípka vychádzajúca z hlavice rodičovského objektu a končiaca na hlavici vysúvaného objektu:

Obr. č. 28 Vysunutá hlavica objektu a symbolická vytváracia správa

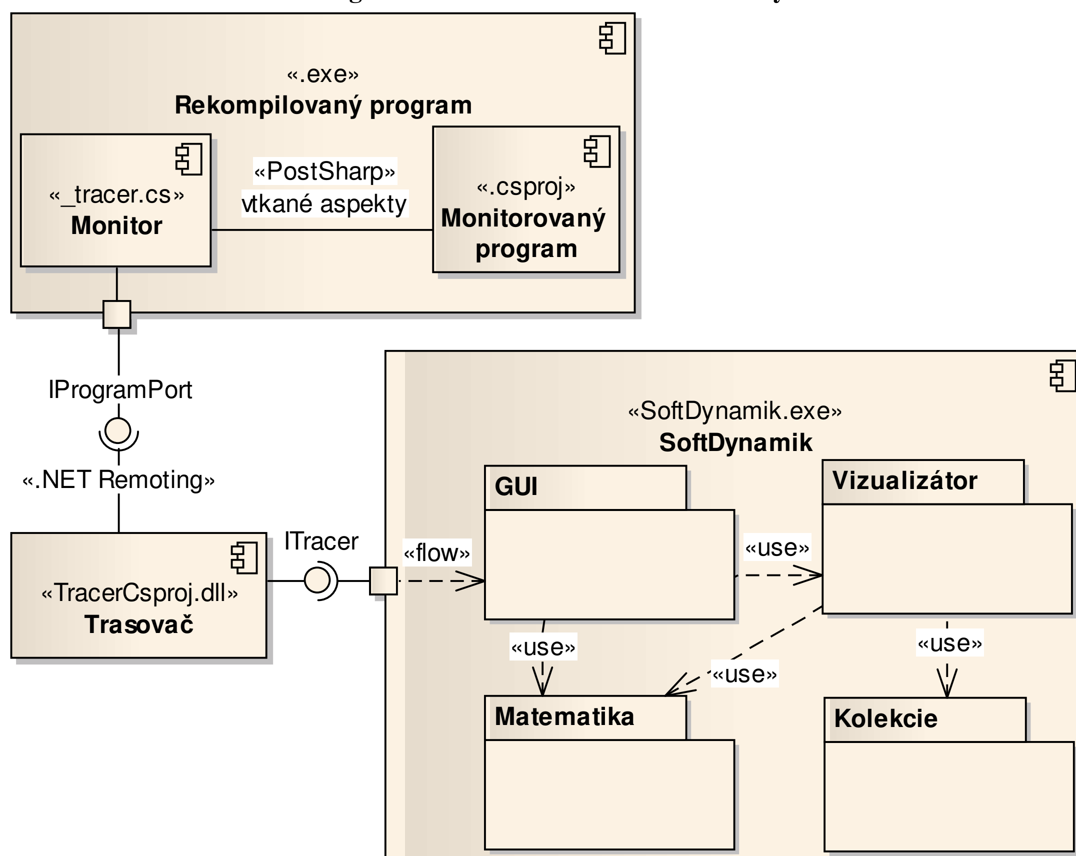


- **Mód skrývania prázdnych objektov:**
  Po zapnutí tohto módu bude každý objekt ihneď po tom čo bol vizuálne vyprázdnený (v scéne je znázornená len hlavica toto objektu) skrytý. Výnimkou je prípad ak je objekt rodičom iného viditeľného objektu (kvôli zachovaniu stromu vytvárania objektov). Toto obmedzenie je obmedzením x – ovej a y – ovej dimenzie.
- **Mód vysúvania prázdnych objektov** na hladinu hlavice rodičovského objektu:
  Metóda, ktorá by vytvárala veľké množstvo objektov by bola reprezentovaná veľmi dlhým valcom a scéna by bola neprehľadná. Preto je možné zapnúť mód, ktorý hlavicu objektu hneď potom ako bude objekt vizuálne vyprázdnený (bude viditeľná len jeho hlavica) vysunie na úroveň hlavice rodičovského objektu a vytváraciu správu tohto objektu znázorní len symbolicky.
  Toto obmedzenie obmedzuje z – ovú dimenziu.
- **Mód ignorovania premenných:**
  Vizualizovanie prístupu k členským premenným tried (ktoré sú v terminalógií jazyka C# nazývané polia) môže celú vizualizáciu nadmerne spomaľovať, preto ho je možné vypnúť.
  Toto obmedzenie obmedzuje z – ovú dimenziu a čas animácie.
- **Preskakovací mód:**
  Mód umožňuje preskakovať ďalšie volania metód a ďalší prístup k poliam. Obmedzuje z – ovú dimenziu a čas animácie.

Názorné **príklady uplatnenia** týchto obmedzení sú uvedené v kapitole 8.3.

7.2 Architektúra navrhnutého vizualizačného systému

UML diagr. č. 1 Architektúra vizualizačného systému



Návrh architektúry vizualizačného systému rieši tieto **základné problémy**:

- Spôsob získavania informácií k vizualizácií – spôsob monitorovania dynamiky sledovaného programu.
- Spôsob prenosu získaných informácií do vizualizačného programu. Dôraz sme kládli na rozšíriteľnosť.
- Spôsob samotnej vizualizácie vo vizualizačnom programu a v prípade zvolenia synchrónneho spôsobu monitorovania aj riadenie (krokovanie) sledovaného programu používateľom vizualizačného programu.

Štruktúra navrhnutej **architektúry** je znázornená v UML diagr. č. 1 prostredníctvom diagramu komponentov.

Navrhovaný vizualizačný systém je vykonávaný v **2 samostatných procesoch** operačného systému:

- Proces sledovaného programu:

  Tento proces je zavedený zo spustiteľného súboru nanovo skompilovaného (rekompilovaného) sledovaného programu, ktorý bol doplnený

o monitorovaciu logiku (komponenta Monitor) a do ktorého boli vtkané monitorovacie aspekty prostredníctvom .NET rámca PostSharp počas rekompilácie.

- Proces vizualizačného programu:

  Proces je zavedený zo spustiteľného súboru používateľského rozhrania vizualizačného systému, ktorému zodpovedá komponenta SoftDynamik.

  Ku komponente SoftDynamik je možné dynamicky pripájať trasovače (dynamicky pripájané knižnice), ktoré musia poskytovať rozhranie ITacer. Komponenta Trasovač má za úlohu komponentu SoftDynamik dynamicky poskytovať informácie, ktoré sa budú vizualizovať. Takto je zabezpečená nezávislosť vizualizácie od spôsobu monitorovania sledovaného programu, od spôsobu komunikácie s ním a aj od jazyka, v ktorom bol sledovaný program implementovaný. Trasovač tak plní úlohu adaptéru. My sme v rámci tejto práce implementovali trasovač pre priame (v reálnom čase, "online") monitorovanie programu napísaného v jazyku C#. Nami implementovaný komponent zabezpečuje automatickú rekompiláciu a spustenie sledovaného programu.

Spomínané procesy spolu komunikujú pomocou technológie .NET Remoting, ktorá implementuje RMI (Remote Method Invocation – vzdialené volanie metód) slúžiace k tvorbe distribuovaných systémov.

Rozhrania jednotlivých **komponent** sú definované v samostatnom zostavení (anglicky assembly) SoftDynamik.SI.dll, ktoré referencujú všetky komponenty. Z pohľadu architektúry je najvýznamnejšie rozhranie ITracer implementované adaptérom Trasovač, ktoré bližšie popisujeme v kapitole 7.4.

Komponenta SoftDynamik obsahuje nasledovné menné priestory (anglicky namespace):

- Menný priestor GUI:

  Zabezpečuje interakciu s používateľom a v rámci nej riadi krokovanie sledovaného programu pričom získava informácie o jeho dynamike. Tieto ďalej odovzdáva mennému priestoru Vizualizátor na spracovanie.

- Menný priestor Vizualizátor:

  Obsahuje vizualizačnú logiku a to najmä objektový model scény a algoritmus rozmiestnenia objektov v scéne.

- Menný priestor Matematika:

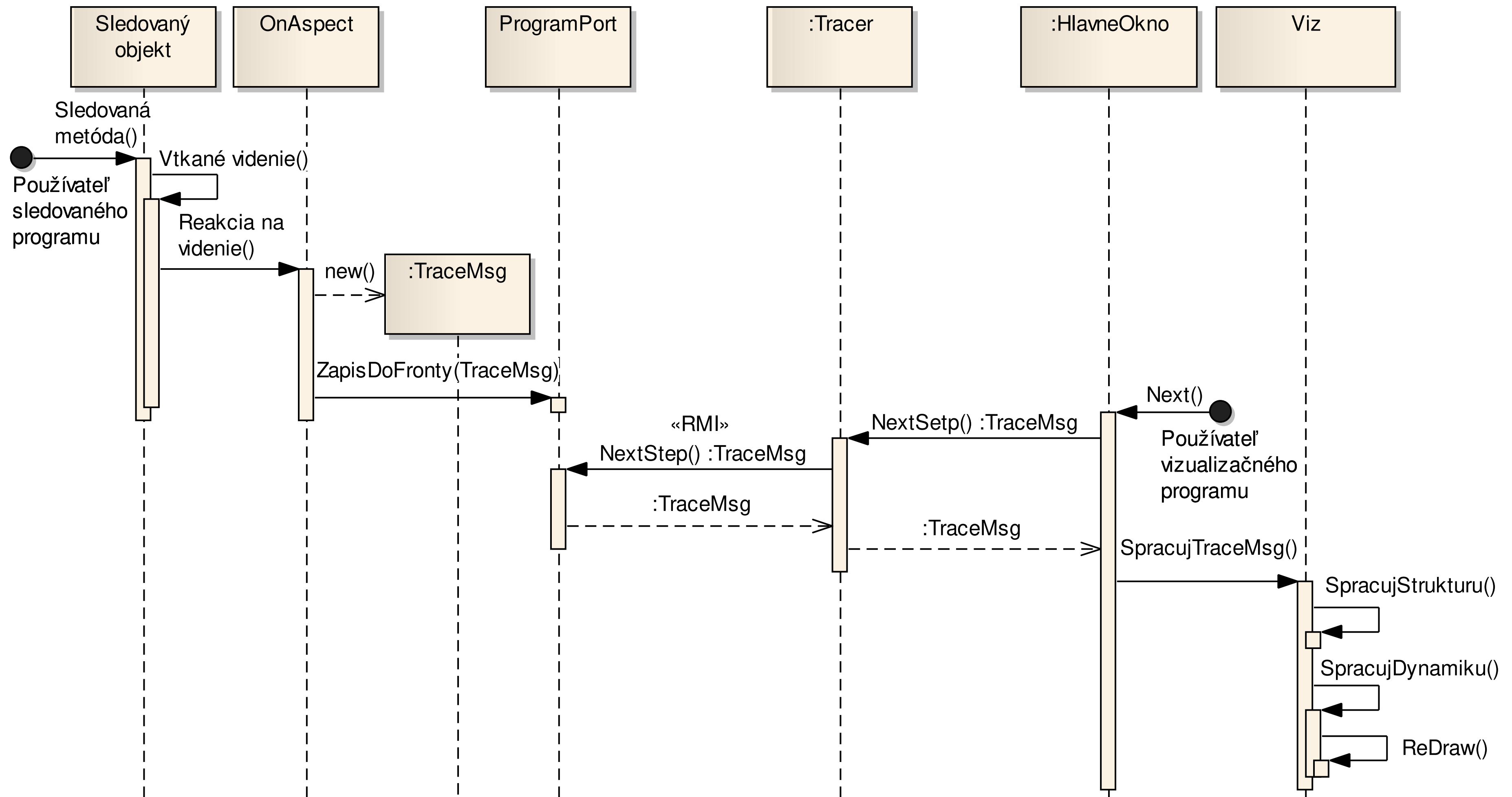
Obsahuje nami implementované všeobecné triedy podporujúce matematické výpočty (výpočty s uhlami, vektormi a súradnicovými sústavami).

- Menný priestor Kolekcie:

  Obsahuje nami implementované generické kolekcie: AVL strom a Prioritný front využívané v nami navrhnutých algoritmoch.

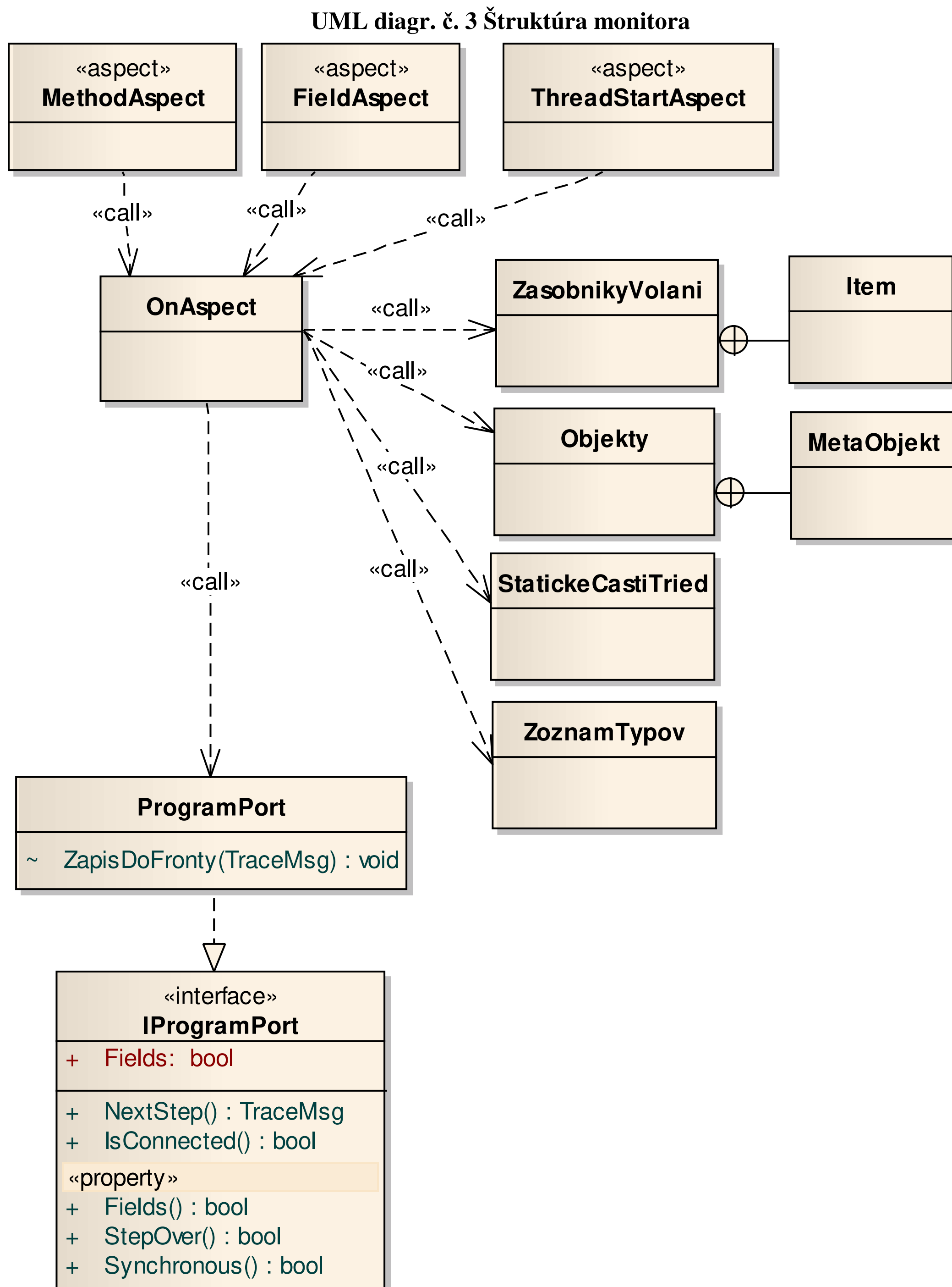
Dynamiku architektúry znázorňuje UML diagr. č. 2 prostredníctvom sekvenčného diagramu. Zo sekvenčného diagramu vidíme, že systém má dvoch používateľov – používateľa monitorovaného programu a používateľa vizualizačného programu. Vidíme celú reakciu systému od okamihu vyvolania niektorej zo sledovaných metód používateľom sledovaného programu v sledovanom programe (napríklad prostredníctvom ovládacích prvkov používateľského rozhrania) až po prekreslenie scény vo vizualizačnom programe. *Sledovaný objekt* je objektom sledovaného programu. Triedy *OnAspect* a *ProgramPort* sú triedy komponenty Monitor. Trieda *Tracer* je triedou komponenty Trasovač a triedy *HlavneOkno* a *Viz* sú triedami komponenty SoftDynamik. Získané informácie o dynamike sledovaného programu sú prenášané objektom triedy *TraceMsg* (viď UML diagr. č. 6), ktorý tiež môže prenášať niektoré informácie o štruktúre toho programu.

UML diagr. č. 2 Získanie a tok informácií o sledovanom programe (pri asynchrónnom spôsobe monitorovania)



7.3 Komponenta Monitor

UML diagr. č. 3 Štruktúra monitora

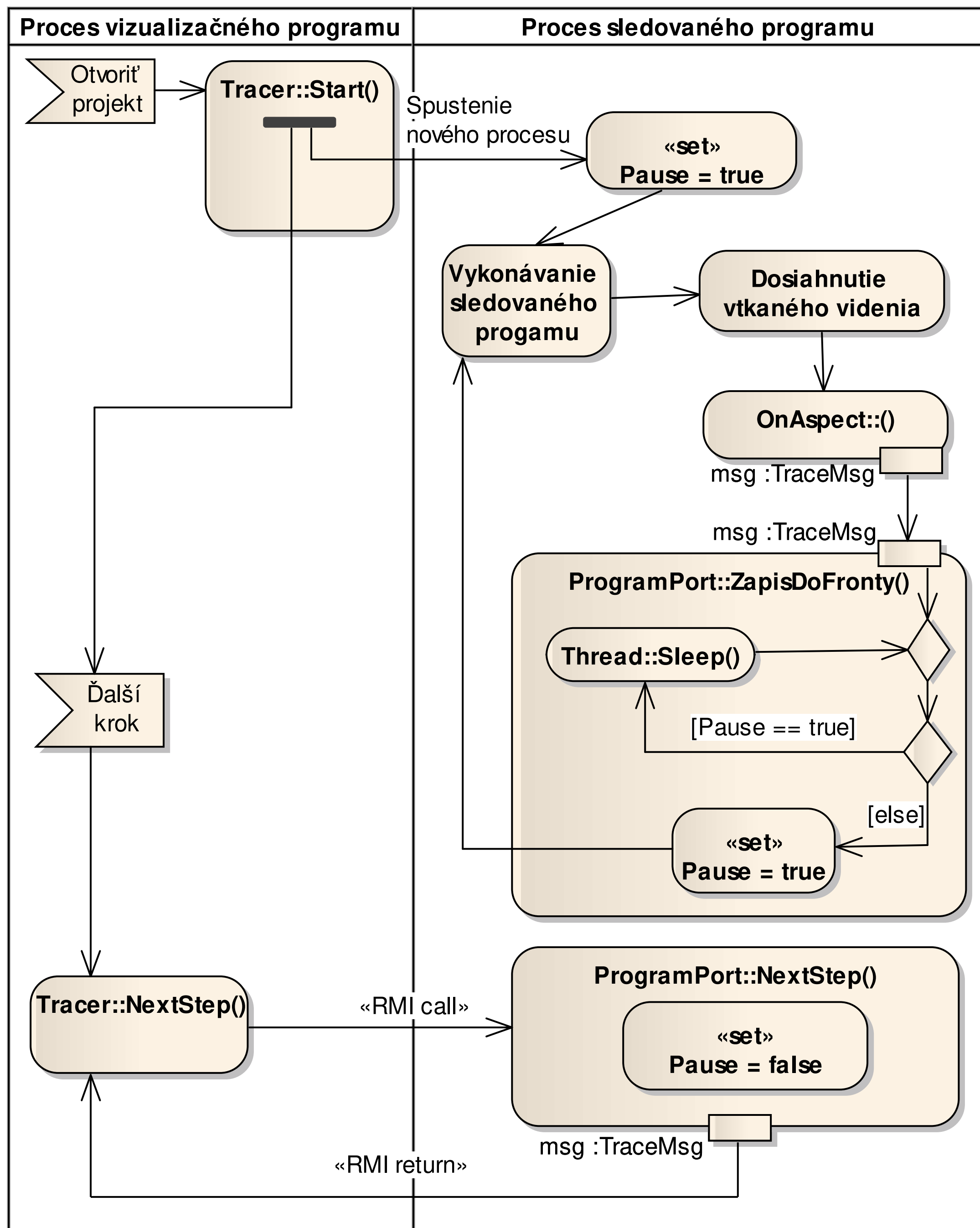


Komponent monitor predstavuje univerzálny a ucelený zdrojový kód (množinu tried) pridaný k zdrojovému kódu sledovaného programu a spolu s ním skompilovaný. Tento zdrojový kód obsahuje nasledovné časti:

- **Monitorovacie aspekty,** ktorých videnia sú počas kompilácie vtkané do zdrojového kódu sledovaného programu.

- **Reakcie na dosiahnuté videnia vtkaných aspektov** (ťažiskom je trieda OnAspect) nezávislé od spôsobu ich vtkania (nezávislé od konkrétnej verzie .NET rámca PostSharp)
- **Komunikácia s trasovačom** (ťažiskom je trieda ProgramPort), zabezpečujúca odpovede na dotazy trasovača ohľadne získaných informácií o dynamike. V prípade sychrónneho spôsobu monitorovania zabezpečuje táto časť reakcie na krokovacie požiadavky trasovača.

UML diagr. č. 4 Dynamika monitora a spôsob synchrónneho monitorovania



Monitorovanie dynamiky programu môže byť vzhľadom k vizualizácií vykonávané dvomi spôsobmi (ktoré si volí používateľ):

- **synchrónny spôsob** monitorovania:

  Tok riadenia sledovaného programu je ovládaný vizualizačným programom (cez trasovač) tak, že vizualizácia zobrazuje to, čo sa v sledovanom programe práve deje. Za týmto účelom je potrebné spomaliť rýchlosť vykonávania jeho opakovaním prerušovaním (krokovaním).

  Spôsob akým ako sa to dosahuje znázorňuje UML diagr. č. 4 (diagram aktivít), ktorý zároveň celkovú dynamiku kompontenty Monitor.
- **asynchrónny spôsob** monitorovania:

  Údaje o dynamike programu získané monitorom sú vo väčšom množstve uchovávané v jeho pamäťovom priestore (vo fronte triedy *ProgramPort*) v časovom poradí a dodatočne (počas vykonávania nasledujúcich akcií v sledovanom programe) sú prečítané trasovačom a vizualizované. Tento spôsob umožňuje používateľovi sledovaného programu plynule pracovať s používateľským rozhraním.

7.3.1 Monitorovacie aspekty

Tab. č. 2 uvádza, ktoré udalosti v sledovanom programe sú sledované a tiež, ktoré konkrétne typy aspektov a ich videní boli na toto sledovanie použité, v závislosti od použitej verzie .NET rámca PostSharp:

Tab. č. 2 Sledované udalosti v sledovanom programe pomocou aspektov

Sledovaná udalosť	**Aspekt::Videnie()**	
	PostShrap 1.5	PostSharp 2.x
Začatie vykonávania metódy	OnMethodBoundaryAspect::OnEntry()	
Úspešné ukončenie metódy	OnMethodBoundaryAspect::OnSuccess()	
Ukončenie metódy vyhodením výnimky	OnMethodBoundaryAspect::OnException()	
Čítanie poľa	OnFieldAccessAspect:: OnGetValue()	LocationInterceptionAspect:: OnGetValue()
Nastavenie poľa	OnFieldAccessAspect:: OnSetValue()	LocationInterceptionAspect:: OnSetValue()
Volanie metódy Thread::Start() – spustenie nového vlákna	OnMethodInvocationAspect:: OnInvocation()	MethodInterceptionAspect:: OnInvoke()

PostSharp 1.5 je kompatibilný s .NET Framework 2.0, PostShrap 2.x s novším.

Aspekty sú aplikované na celé zostavenie (anglicky assembly) s výnimkou komponentu Monitora (čo by spôsobilo nekonečné cyklické vtkávanie aspektov na seba) pomocou nasledovných príkazov umiestnených v zdrojovom kóde komponentu Monitor mimo menné priestory:

```
//Hromadné aplikovanie aspektov
[assembly: Aspekty.MethodAspect(AttributeTargetTypes = "*",
         AttributePriority = 1)]
[assembly: Aspekty.MethodAspect(AttributeTargetTypes = "Aspekty.*",
         AttributeExclude = true, AttributePriority = 2)]

#if PostSharp_1 //PostSharp 1.5
[assembly: Aspekty.FieldAspect(AttributeTargetTypes = "*",
         AttributePriority = 1)]
#endif
#if PostSharp_2 //PostSharp 2.x
[assembly: Aspekty.FieldAspect(AttributeTargetTypes = "*",
         AttributeTargetMemberAttributes =
         PostSharp.Extensibility.MulticastAttributes.NonAbstract,
         AttributePriority = 1)]
#endif
[assembly: Aspekty.FieldAspect(AttributeTargetTypes = "Aspekty.*",
         AttributeExclude = true, AttributePriority = 2)]

[assembly: Aspekty.ThreadStartAspect(AttributeTargetAssemblies =
         "mscorlib", AttributeTargetTypes = "System.Threading.*",
         AttributeTargetMembers = "Start", AttributePriority = 1)]
```

Videnia volajú metódy, ktoré reagujú na vyskytnutú udalosť nezávisle od použitej verzie PostSharp.

7.3.2 Reakcie na dosiahnuté videnia vtkaných aspektov

Komponent Monitor si za účelom reakcií na sledované udalosti (dosiahnuté videnia) vytvára a udržuje nasledovné **zoznamy**:

- Zoznam objektov
- Zoznam statických objektov (statických častí tried)
- Zoznam typov
- Zoznam zásobníkov volaní pre každé vlákno

Po dosiahnutí videnia vtkaného na začiatok metódy**,** sa v reakcií na toto videnie určí:

- Či ide o statickú metódu (ak je referencia na objekt, nad ktorým je vykonávaná daná metóda, získaná cez kontext bodu spájania je null) alebo je to metóda inštančná.
- V prípade inštančnej metódy sa určí či objekt, nad ktorým je vykonávaná už je zaznamenaný v zozname objektov. Ak nie, indikuje to **vytvorenie nového objektu**.
- V prípade statickej metódy sa určí či typ, ktorý metódu deklaruje (získaný cez kontext bodu spájania) je už zaznamenaný v zozname typov. Ak nie, indikuje to vytvorenie *statického objektu*.

Kontext bodu spájania neposkytuje informácie o druhom účastníkovi posielanej správy (v zmysle objektovo - orientovanej paradigmy) – napríklad o tom, ktorá metóda, ktorého objektu zavolala danú metódu. Za účelom **zistenia druhého účastníka** vytvárame (paralelne so sledovaným programom) vlastný zásobník volaní a to pre každé vlákno zvlášť. Napríklad volajúcu metódu (a jej objekt) potom v čase reakcie na dosiahnuté videnie vtkané na začiatok metódy určíme ako vrchol zásobníka volaní aktuálneho vlákna.

V prípade ,že metóda (označme ju metóda A) v zdrojovom kóde sledovaného programu volá metódu, ktorej zdrojový kód nie je súčasťou zdrojového kódu sledovaného programu (označme ju metóda B) a táto ďalej volá metódu v zdrojovom kóde sledovaného programu (označme ju metóda C), tak metóda B sa nedostane do nášho zásobníka volaní (pretože sa do nej nevtkal žiadny monitorovací aspekt a tak sme neboli informovaní o začatí jej vykonávania). V okamihu zachytenia udalosti volania metódy C, bude na vrchole nášho zásobníka volaní metóda A, ktorá bude označená ako volajúca (odosielajúci účastník správy). Táto správa (toto volanie) bude označená ako nepriama správa. To, že ide o nepriame volanie identifikujeme pomocou .NET triedy StackFrame, ktorá nám umožňuje zistiť názvy metód na zásobníku volaní sledovaného programu, nie však objekty týchto metód (teda neumožňuje identifikovať presnú inštanciu danej metódy).

Ak je pri zisťovaní druhého účastníka zásobník volaní daného vlákna prázdny, znamená to, že práve došlo k spusteniu nového vlákna. Volajúcu metódu a objekt, nad ktorým bola vykonávaná určíme z predchádzajúceho zachytenia udalosti volania metódy Thread::Start(), ktorá vytvára nové vlákno.

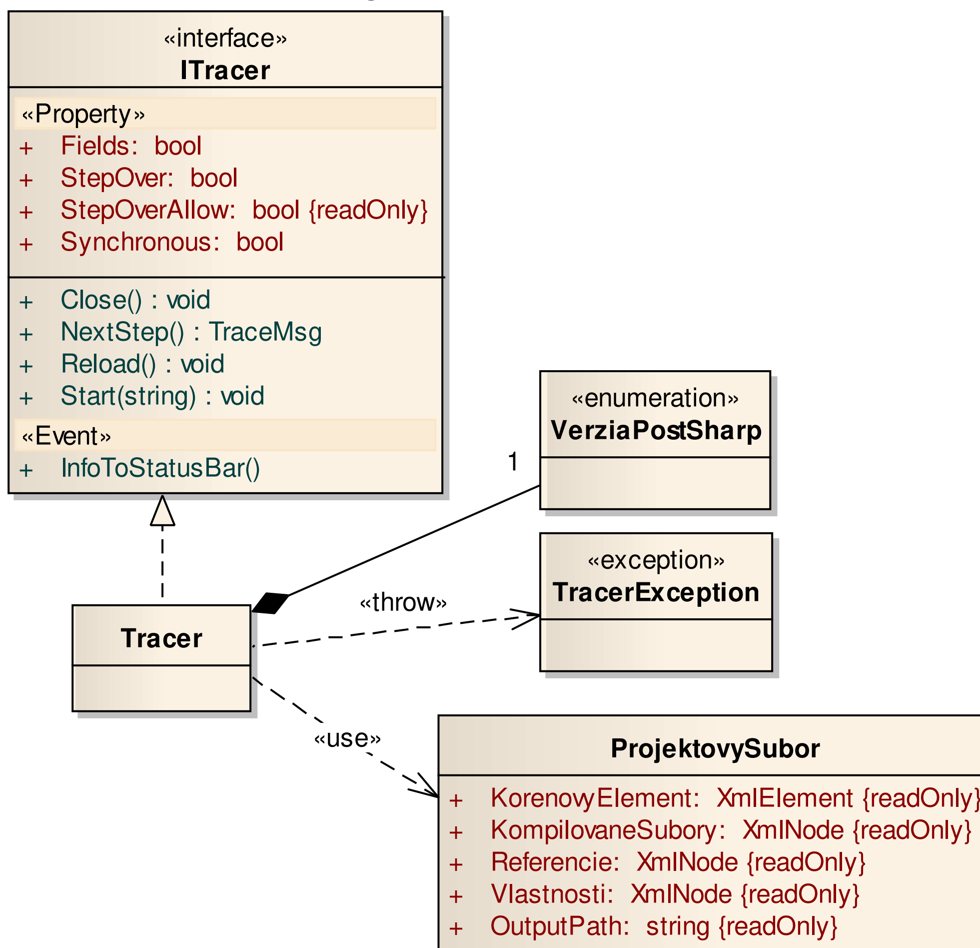
Metódy komponentu Monitor reagujúce na dosiahnuté videnia (sledované udalosti sledovaného programu) nakoniec vytvárajú správy o aktivitách v sledovanom programe

a tieto zapisujú do **fronty správ** (v komunikačnej časti komponenty) odkiaľ ich číta trasovač (prostredníctvom komunikačnej časti Monitora).

7.4 Komponenta Trasovač

Komponenta Trasovač je adaptér umožňujúci adaptovať rôzne spôsoby monitorovania sledovaných programov napísaných v rôznych jazykoch.

UML diagr. č. 5 Štruktúra trasovača



Každý trasovač (napríklad trasovač vytvorený používateľom) musí byť .NET zostavenie (assembly), obsahujúce verejnú (public) triedu Tracer v mennom priestore (namespace) Tracer, ktorá implementuje rozhranie ITracer definované v UML diagr. č. 5. Bližší popis tohto rozhrania obsahuje Príloha č. 2 Používateľská príručka.

Trasovač je potrebné zaregistrovať vo vizualizačnom programe (*SoftDynamik.exe*), čo je možné urobiť dynamicky cez dialógové okno GUI.

Pred začatím trasovania volá vizualizačný program metódu trasovača *Start,* (ktorej implementáciu vyžaduje verejné rozhranie *ITracer* ktoré trasovač

implementuje). Nami implementovaný trasovač v reakcií na volanie metódy *Start* vykoná tieto kroky:

1. **Číta projektový súbor** projektu jazyka C# (*.csproj), čo je XML súbor s informáciami o projekte (napríklad verzia projektového súboru, konkrétna cieľová platforma, referencované .NET zostavenia (assembly), zdrojové súbory projektu (*.cs), cieľový priečinok kompilátora)
2. Do projektu **doplní** zdrojový súbor **komponetu Monitor** a potrebné referencie na .NET zostavenia (assembly) a to tak, že modifikuje kópiu projektového súboru. Konkrétny spôsob modifikácie závisí od verzie projektu.
3. Zálohuje prípadný už skompilovaný spustiteľný súbor.
4. Spustí **proces kompilátora** príslušnej verzie .NET (cestu k nemu prečíta z registrov operačného súboru) s parametrom obsahujúcim cestu k modifikovanej kópií a čaká na jeho dokončenie. Súčasťou tohto procesu je aj automaticky spustené vtkanie aspektov dodaných v komponente Monitor. Výsledkom činnosti kompilátora je vytvorenie spustiteľného súboru rekompilovaného programu v priečinku uvedenom v projektovom súbore, čím premaže prípadný pôvodný spustiteľný súbor.
5. Po dokončení procesu kompilácie trasovač **spustí rekomplikovaný program**.
6. Nadviazanie komunikácie s procesom rekompoilovaného programu cez .NET Remoting.

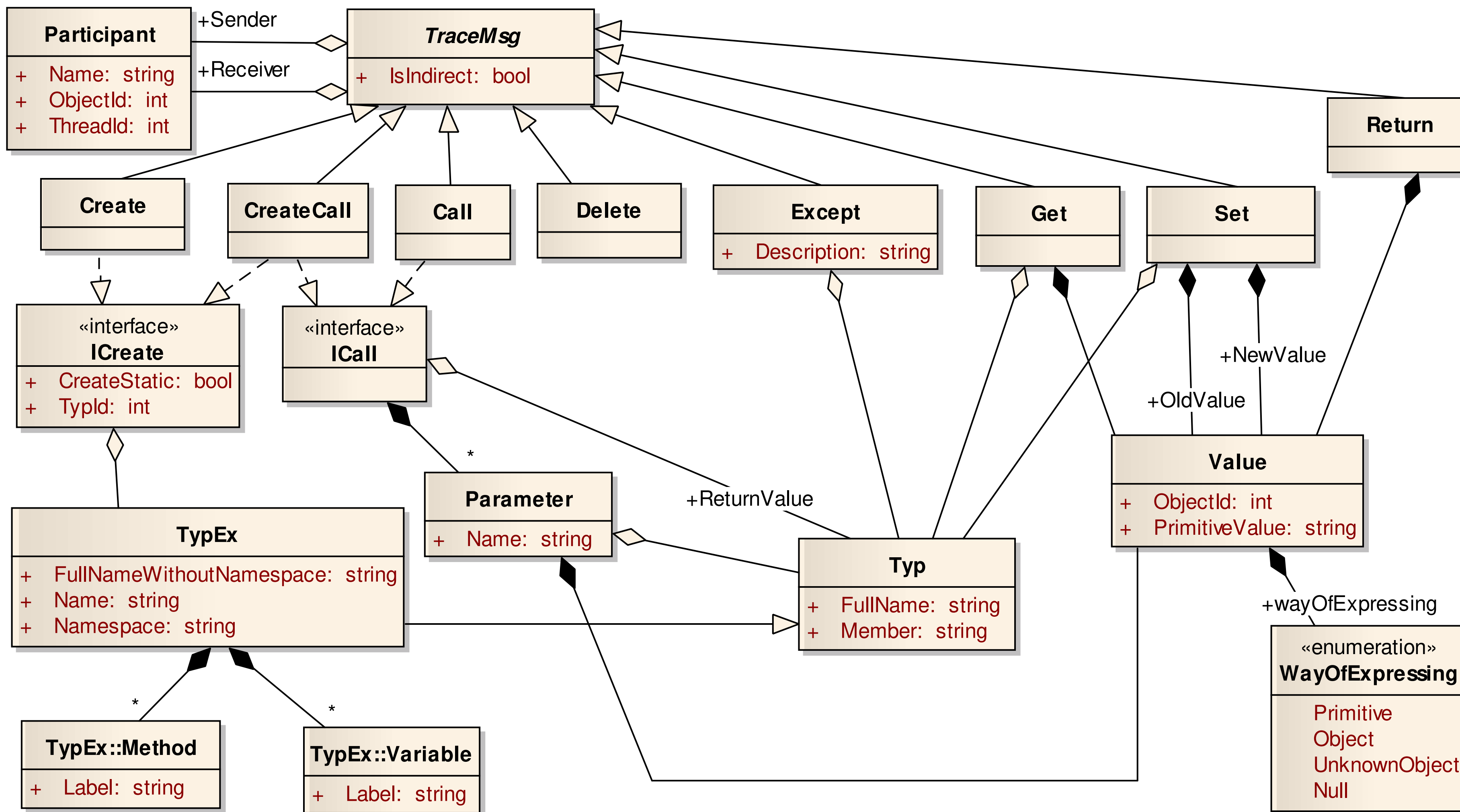
Posledné dva kroky sa vykonajú tiež po volaní metódy trasovača *Reload*.

Následne (**počas trasovania**) vizualizačný program volá metódu trasovača *NextStep*, vždy keď od trasovača požaduje aktuálnu informáciu (správu) o dynamike sledovaného programu. Metóda NextStep musí tieto informácie vrátiť vo forme objektu typu zdedeného od abstraktnej triedy *TraceMsg*. Konkrétny typ vráteného objektu zodpovedá typu vracanej správy. Možné typy správ sú špecifikované v UML diagr. č. 6 a sú definované v .NET knižnici SoftDynamik.SI.dll, ktorú trasovač referencuje. V prípade, že momentálne nie sú k dispozícií žiadne ďalšie správy k vráteniu musí metóda NextStep vráti null. V prípade, že došlo k chybe musí metóda vyhodiť výnimku *TracerException*.

Nami implementovaný trasovač vracia vizualizačnému programu správu, ktorú si prečíta z fronty správ komponenty Monitor prostredníctvom technológie .NET Remoting.

Informácie o postupe implementácie vlastného trasovača používateľom a jeho registrácií vo vizualizačnom programe obsahuje Príloha č. 2 Používateľská príručka.

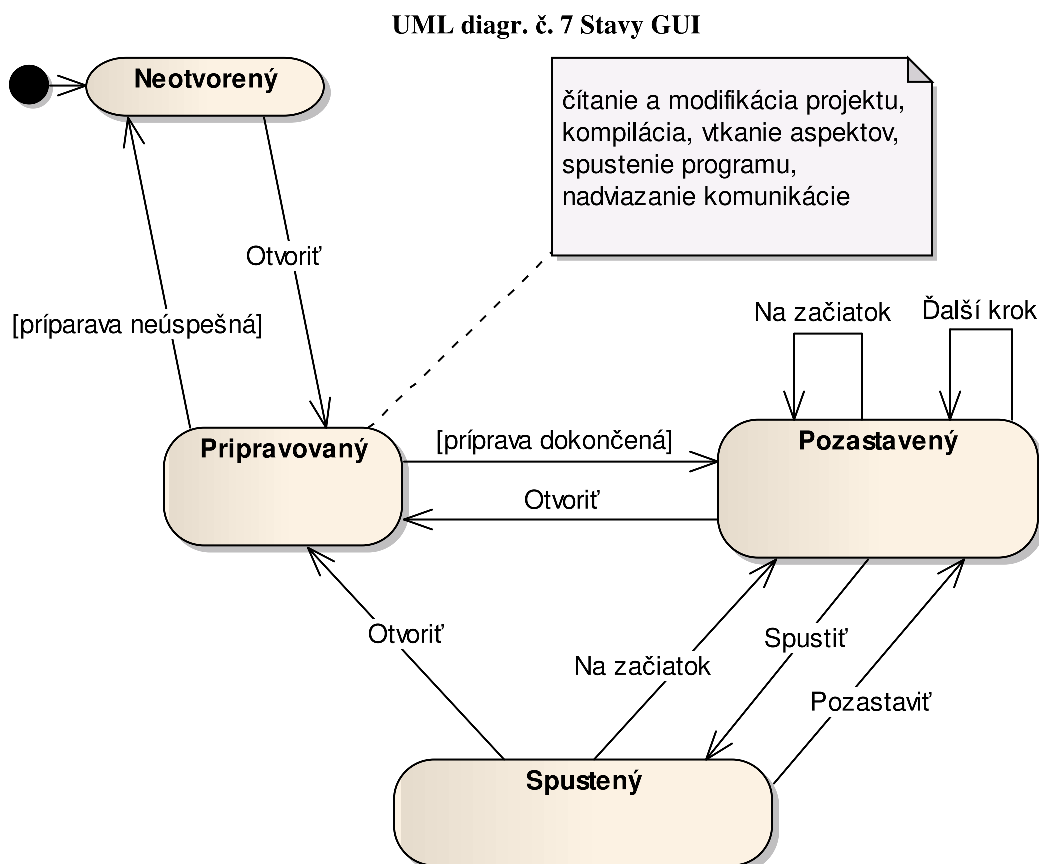
UML diagr. č. 6 Štruktúra trasovacích správ (návrh komunikačného protokolu)



7.5 Menný priestor GUI

Jednou z hlavných úloh menného priestoru GUI je umožniť používateľovi krokovať sledovaný program. To si vyžaduje definovať **stavy krokovania**, a možné prechody medzi nimi:

UML diagr. č. 7 Stavy GUI



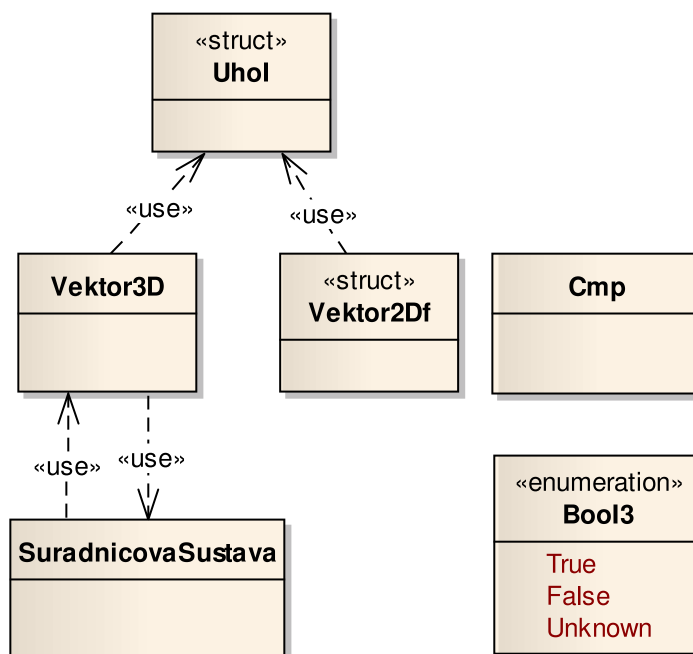
Návrh vzhľadu grafického používateľského rozhrania programu SoftDynamik a jeho podrobný **popis** obsahuje Príloha č. 2 Používateľská príručka, ktorú je potrebné považovať za súčasť tejto kapitoly.

7.6 Menný priestor Matematika

Pre účely tejto práce sme implementovali aj triedy pre pohodlné geometrické výpočty ako počítanie s uhlami, vektormi (napr. vektorový súčin, otáčanie vektora,...) a súradnicovými sústavami (transformácie a spätné transformácie bodov a vektorov). Triedu Cmp používame k bezpečnému porovnávaniu čísel v pohyblivej rádovej čiarke

(pred porovnaním čísla zaokrúhli) a výpočtový zoznam Bool3 sme implementovali pre potreby trojhodnotovej logiky.

UML diagr. č. 8 Triedy menného priestoru Matematika

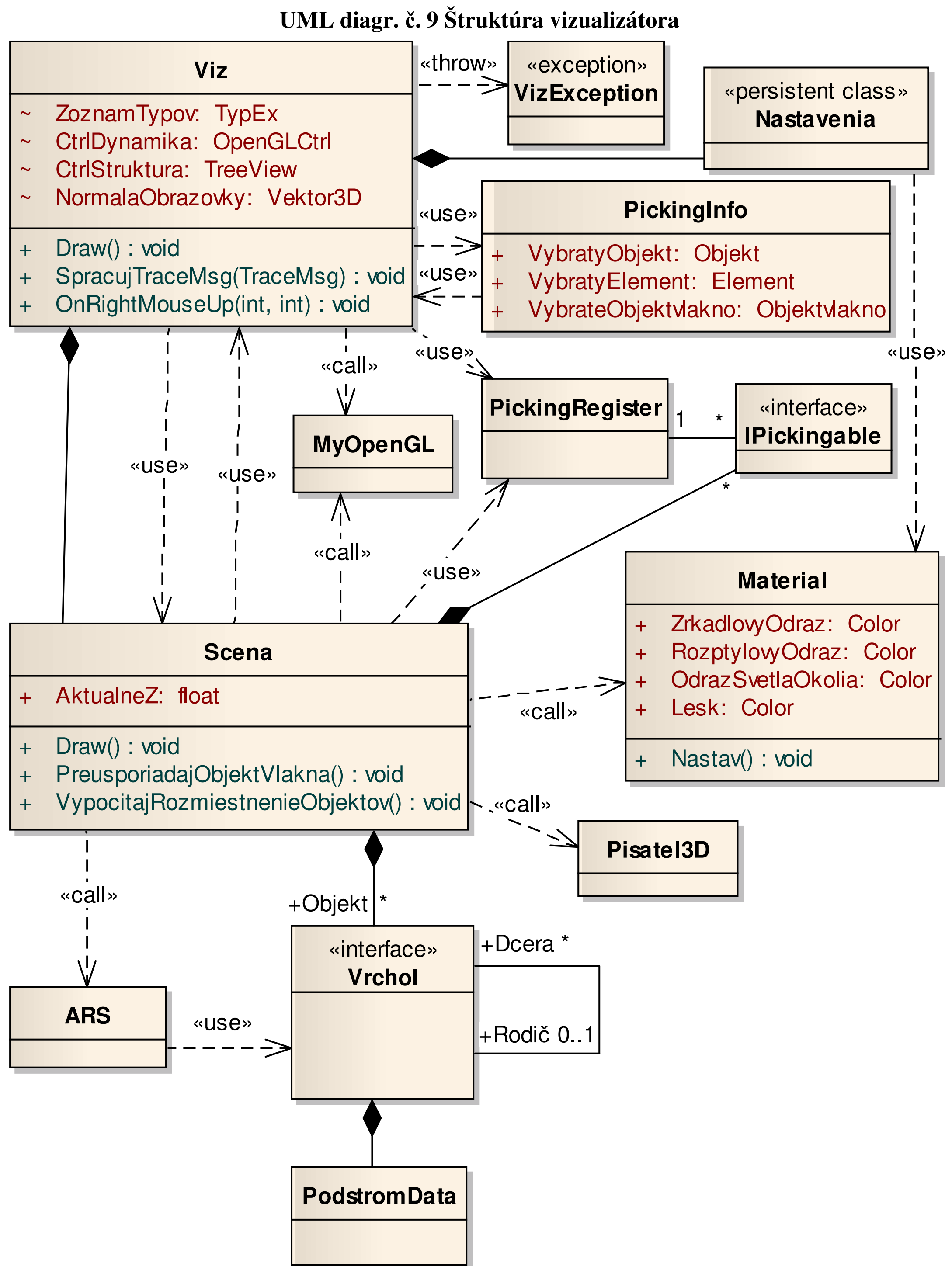


7.7 Menný priestor Vizualizátor

Spracovanie získaných údajov o monitorovanom programe do zobraziteľnej podoby má za úlohu menný priestor vizualizátor. Najdôležitejšie triedy vizualizátora môžete vidieť v UML diagr. č. 9. Rozhranie vizualizátora tvorí trieda *Viz*. V diagrame sú u tejto triedy (rovnako ako aj u ostatných) uvedené len niektoré metódy a vlastnosti, ktoré obsahuje, kvôli naznačeniu jej úlohy. V skutočnosti je táto trieda oveľa rozsiahlejšia. Prijaté trasovacie správy (objekty abstraktného typu *TraceMsg*) sú menným priestorom GUI odovzdávané metóde *Viz::SpracujTraceMsg(TraceMsg msg)*, ktorá číta správu a samostatne spracuje údaje o štruktúre (ktoré následne zobrazí v GUI .NET komponente TreeView) a samostatne údaje o dynamike. Na základe údajov o dynamike modifikuje objektový model scény (v zmysle konceptu vizualizácie, ktorý popisujeme v kapitole 7.1), ktorý je reprezentovaný triedou *Scéna* a na základe ktorého je následne prekreslená scéna v GUI komponente *OpenGLCtrl* (poskytnutej knižnicou SharpGL, ktorá umožňuje pohodlne volať funkcie grafickej knižnice OpenGL z programov napísaných v jazyku C#). Modifikácia objektového modelu scény

(dopĺňanie a hlavne odstraňovanie jednotlivých prvkov vizualizácie) je programátorsky najnáročnejšou časťou celej práce.

UML diagr. č. 9 Štruktúra vizualizátora



Objektový model scény (graf scény) je tvorený hierarchickou štruktúru tried, ktoré nie sú kvôli prehľadnosti znázornené v UML diagr. č. 9, sú však znázornené v UML

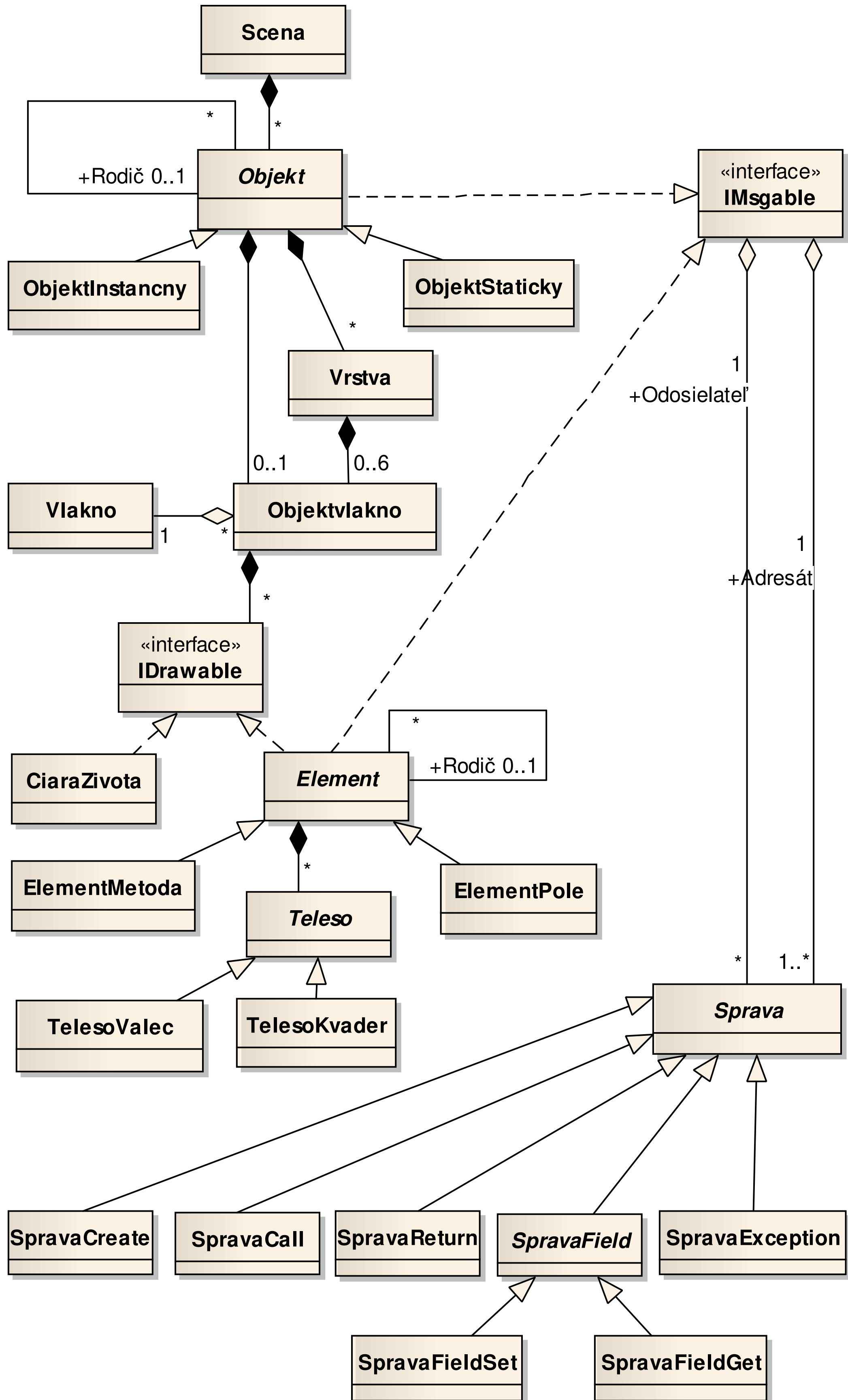
diagr. č. 10. Štruktúra objektového modelu scény je veľmi dôležitá - do značnej miery vyjadruje filozofiu navrhnutého konceptu vizualizácie.

Z oboch UML diagramov môžeme vidieť, že trieda Scéna obsahuje zoznam vizualizovaných objektov, ktorých horizontálne rozmiestnenie v priestore scény je vypočítavané nami navrhnutým algoritmom ARS (popísaný v kapitole 7.7.1), ktorý je obsiahnutý v triede *ARS*.

Vizualizácia je používateľom ovládaná cez viaceré parametre, ktorých hodnoty sú uložené v triede *Nastavenia*. Táto trieda je perzistentná, čo je zabezpečené jej serializáciou do súboru v okamihu ukončenia vizualizačného programu a deserializáciou v okamihu štartu vizualizačného programu.

Naše rozšírenia knižnice OpenGL sú uložené v triede MyOpenGL a v triede Pisatel3D, ktorá zabezpečuje výpis popiskov v scéne. Pomocou objektov triedy Material uchovávame všetky údaje o materiáloch telies v scéne, potrebné k výpočtu farby pixlov týchto telies pri zapnutom osvetlení.

UML diagr. č. 10 Prvky scény a základné vzťahy medzi nimi (graf scény)



7.7.1 Algoritmus rozmiestnenia objektov v scéne - ARS

Dôležitou súčasťou navrhovanej vizualizácie je algoritmus rozmiestňujúci objekty v scéne. Objekty tvoria acyklický orientovaný graf a treba ich vhodne (prehľadne) rozmiestniť v 2D priestore (v dimenziách x, y)

Za týmto účelom sme navrhli všeobecný algoritmus **určený** pre:

- ľubovoľný acyklický orientovaný graf (strom)
- 2D priestor
- vrcholy aj s nenulovým polomerom – vrcholy sú kruhové, nie bodové

Pri jeho návrhu sme si kládli nasledovné základné **požiadavky**:

- Lineárna asymptotická časová zložitosť, aby bol použiteľný dynamicky v priebehu vizualizácie (napríklad pri každom pridaní nového objektu)
- Zachovanie viditeľnosti hierarchie stromu – vetvy stromu sa nesmú vzájomne preplietať.
- Možnosť nastaviť minimálnu vzdialenosť okrajov kruhových vrcholov.

Navrhnutý algoritmus sme pre účely tejto práce nazvali Algoritmus Rozmiestňujúci Strom (skratka ARS).

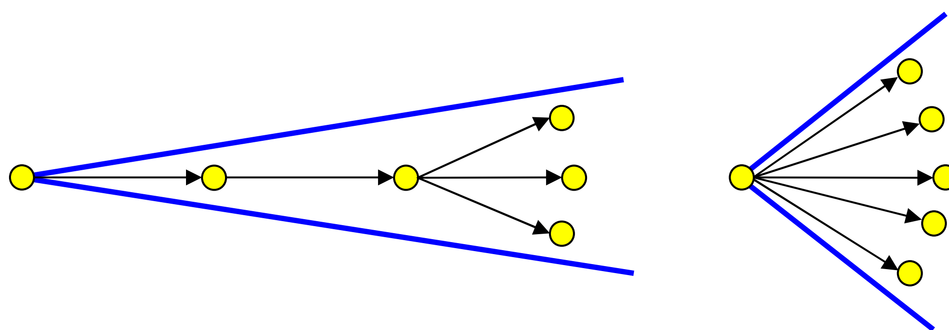
Základným princípom ARS je rozdeľovanie uhlového výseku podstromu dcérskym podstromom v pomere určenom pomerom ich váh, ktoré sme dopredu vypočítali. Koreň stromu má k dispozícií k rozdeľovaniu uhlový výsek o veľkosti 360°. Dcérsky podstrom má uhlový výsek menší, nanajvýš rovný (ak je to jediný nasledovník rodiča) ako rodič.

ARS pozostáva z dvoch prechodov stromom, preto jeho **asymptotická časová zložitosť** je O(2n), kde n je počet vrcholov v strome. Prvý prechod je rekurzívny prechod zdola nahor (postorder) počítajúci váhy všetkých podstromov v strome. Druhý prechod je rekurzívny prechod zhora nadol (preorder) počítajúci uhlové výseky podstromov na základe ich váh a z týchto uhlových výsekov polohy vrcholov.

Veľkosť uhľového výseku podstromu je priamoúmerný jeho váhe. Navrhujeme, že je vhodné, aby väčší uhlový výsek dostal ten podstrom, ktorý:

- Obsahuje viac vrcholov.
- Jeho vrcholy sú v priemere bližšie ku koreňu podstromu (má menšiu priemernú hĺbku). Význam priemernej hĺbky demonštruje Obr. č. 29, na ktorom sú znázornené dva podstromy s rovnakým počtom vrcholov ale s rozdielnou priemernou hĺbkou:

Obr. č. 29 ARS - Podstromy s rozdielnou priemernou hĺbkou – význam činiteľa $\overline{h}$



- Jeho kruhové vrcholy majú väčší priemerný polomer.

Váha podstromu sa preto vypočíta nasledovne:

$$w = n^a \overline{h}^b \ \overline{r}^c$$

kde:

- n je počet vrcholov v podstrome
- a je parameter (exponent) nastavený používateľom s preddefinovanou hodnotu 1.
- $\overline{h}$ je aritmetický priemer hĺbok vrcholov v podstrome vzhľadom ku koreňu podstromu. Hĺbka koreňa podstromu je 1.

$$\overline{h} = \frac{\sum_{i=1}^{n} h_i}{n}$$

Pre výpočet sumy hĺbok vrcholov v podstrome využívame nasledovaný rekurzívny vzťah:

$$\sum_{i=1}^{n} h_i = s = n + \sum_{j=1}^{|D|} s_j$$

kde:

- D je množina nasledovníkov koreňa podstromu.
- s_j je suma hĺbok vrcholov v podstrome j – teho nasledovníka vzhľadom ku koreňu, ktorým je j - ty nasledovník.

- b je parameter nastavený používateľom s preddefinovanou hodnotou -1, čím je zabezpečené, že dcérskemu podstromu s väčšou priemernou hĺbkou bude pridelený menší uhlový výsek.
- $\bar{r}$ je aritmetický priemer polomerov kruhových vrcholov v podstrome.
- c je parameter nastavený používateľom s preddefinovanou hodnotou 1.

Poloha dcérskeho podstromu P_D je od rodičovského podstromu P posunutá o vektor $\vec{d}$, ležiaci na ose uhlového výseku podstromu u_P, smerujúci do vnútra uhlového výseku. Tento vektor začína v strede vrcholu rodičovského podstromu a končí v strede vrcholu dcérskeho podstromu. Výpočet dĺžky tohto vektora závisí od dvoch parametrov algoritmu ARS nastavených používateľom:

- k_1 je minimálna vzdialenosť okrajov kruhových vrcholov spojených hranou
- k_2 je minimálna vzdialenosť okrajov kruhových vrcholov nespojených hranou

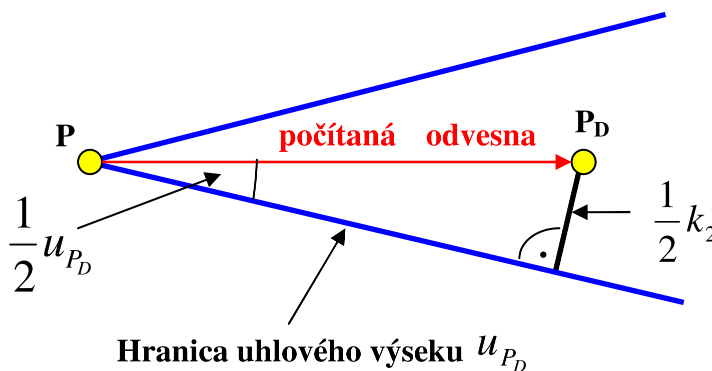
a vypočíta sa nasledovne:

- Ak je uhlový výsek $u_P \leq 180°$

$$d = \max\left(r_P + k_1 + r_{P_D}, \quad \frac{r_{P_D} + \frac{1}{2}k_2}{\sin\left(\frac{1}{2}u_{P_D}\right)} \right)$$

kde r je polomer koreňa podstromu a u je uhlový výsek podstromu.
Druhý argument funkcie max je výpočet odvesny pravouhlého trojuholníka znázorneného na Obr. č. 30:

Obr. č. 30 ARS - Výpočet druhého argumentu funkcie max



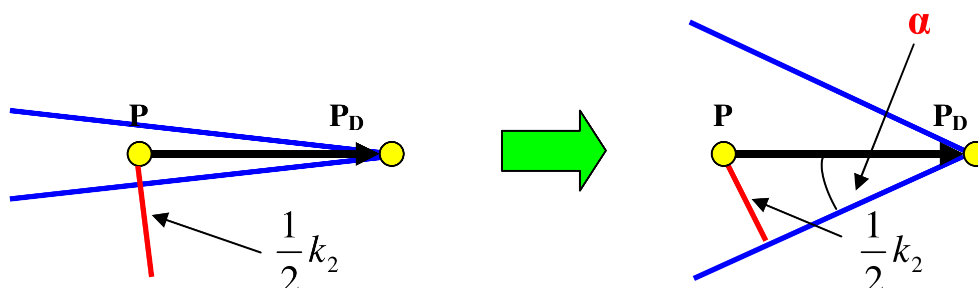
Táto odvesna je tak dlhá, že je zaručené, že vzdialenosť medzi okrajmi súrodeneckých vrcholov nebude menšia ako k_2.

- Inak:

$$d = r_P + k_1 + r_{P_D}$$

Po tom, čo sme vypočítali $\vec{d}$ musíme ešte v prípade, že veľkosť uhlového výseku dcérskeho podstromu $u_{P_D} > 180°$ skontrolovať, či tento **uhlový výsek nezasahuje do priestoru rodičovského podstromu**. V prípade, že zasahuje tak ho zmenšíme tak, aby nezasahoval, ako je to znázorenené na Obr. č. 31:

Obr. č. 31 ARS – Dcérsky podstrom P_D nesmie zasahovať do priestoru P



$$\alpha = \arcsin \frac{r_P + \frac{1}{2}k_2}{d}$$

Ak $u_{P_D} > 360° - 2\alpha$ tak $u_{P_D} = 360° - 2\alpha$

Pri výpočte rozmiestnenia objektov sa **skryté objekty ignorujú**.

Nasleduje **pseudokód** algoritmu ARS:

```
ARS(Koren, k1, k2, a, b, c)
{
	OvahujPodstromy(Koren, a, b, c);
	Koren.Poloha = (0,0);
	Koren.UhlovyVysek = 360°;
	Koren.Smer = (1,0);
	VypocitajPolohyVrcholov(Koren, k1, k2);
}
OvahujPodstromy(P:podstrom, a, b, c)
```

```
{
    Foreach Dieta in P do OvahujPodstromy(Dieta, a, b, c);
    P.PocetVrcholov = 1;
    P.SumHlbka = 0;
    P.SumR = P.r; //r je polomer kruh. vrcholu
    Foreach Dieta in P do
    {
        P.PocetVrcholov = P.PocetVrcholov + Dieta.PocetVrcholov;
        P.SumHlbka = P.SumHlbka + Dieta.SumHlbka;
        P.SumR = P.SumR + Dieta.SumR;
    }
    P.SumHlbka = P.SumHlbka + P.PocetVrcholov;
    P.PriemHlbka = P.SumHlbka / P.PocetVrchlov;
    P.PriemPolomer = P.SumR / P.PocetVrcholov;
    P.Vaha = P.PocetVrcholov^a * P.PriemHlbka^b * P.PriemPolomer^c;
}
VypocitajPolohyVrcholov(P:podstrom, k1, k2)
{
    Smer0 = P.Smer – P.UhlovyVysek / 2;
    Foreach Dieta in P do
    {
        Dieta.UhlovyVysek = P.UhlovyVysek * Dieta.Vaha / SumaVahDeti;
        If Dieta.UhlovyVysek <= 180° then
            Dieta.d = max(P.r+k1+Dieta.r,
                          (Dieta.r+k2/2)/sin(Dieta.UhovyVysek/2));
        Else Dieta.d = P.r+k1+Dieta.r;

        If Dieta.UhlovyVysek > 180° then
        {
            Alfa = arcsin( (P.r+k2)/Dieta.d );
            If Dieta.UhlovyVysek > (360° – 2*Alfa) then
                Dieta.UhlovyVysek = 360° – 2*Alfa;
        }

        Dieta.Smer = Smer0 + (Dieta.UhlovyVysek/2);
        Smer0 += Dieta.UhlovyVysek;
        Dieta.Poloha = P.Poloha + Dieta.d * Dieta.Smer;
    }
    Foreach Dieta in P do VypocitajPolohyVrcholov(Dieta, k1, k2);
}
```

7.7.2 Algoritmus romiestnenia periférmych objekt-vlákien – ARPV

Okrem algoritmu rozmiestňujúceho objekty v scéne, je tiež potrebné navrhnúť algoritmus rozmiestňujúci periférne objekt-vlákna jednotlivých objektov vo vrstvách okolo ich jadrových objekt-vlákien. Cieľom algoritmu je:

- Minimalizovať vzájomné pretínanie šípok vizualizovaných správ a ich pretínanie iných objekt-vlákien.
- Zvýšiť prehľadnosť vizualizácie.

Za týmto účelom sme navrhli algoritmus, ktorý sme pre účely tejto práce pomenovali Algoritmus Rozmiestnenia Periférnych objekt-Vlákien (ARPV).

Pre každé objekt-vlákno sa vypočíta 2D **vektor intenzity komunikácie** $\vec{K}$ (obsahuje súradnice x, y) definovaný vektorovou funkciou:

$$\vec{K}(\vec{t}, S, R) = \sum_{\vec{s} \in S} \frac{\vec{s} - \vec{t}}{\left|\vec{s} - \vec{t}\right|} + \sum_{\vec{r} \in R} \frac{\vec{r} - \vec{t}}{\left|\vec{r} - \vec{t}\right|}$$

kde:

- $\vec{t}$ je 2D poloha objekt-vlákna
- S je množina 2D polôh (súradnice x, y) všetkých odosielateľov medzi - objektových správ určených danému objekt-vláknu.
- R je množina 2D polôh (súradnice x, y) všetkých prijímateľov medzi – objektových správ odoslaných objekt-vláknom.

Smer vektora intenzity komunikácie je najčastejším smerom komunikácie daného objekt-vlákna a jeho veľkosť udáva mieru prevahy tejto komunikácie v danom smere nad inou komunikáciu objekt-vlákna.

Periférne objekt-vlákna daného objektu sa zoradia podľa veľkosti vektora intenzity komunikácie zostupne (od najdlhšieho) a v tomto poradí im je prideľovaná najvýhodnejšia ešte nepridelená možná poloha v okolí jadrového objekt-vláka. **Možné polohy** v polárnej súradnicovej sústave (usporiadaná dvojica (uhol, polomer)) v rovine rovnobežnej s rovnou x, y (v horizontálnej rovine) so stredom v strede objektu definuje vektorová funkcia:

$$\vec{p}(i, j, L, k) = \left(\frac{60°}{L} i + 60° j, \quad R_i(k) \right)$$

pričom k pochopeniu je veľmi dôležité si uvedomiť, že parametre i, j (nasleduje ich popis) môžu nadobúdať len určité celočíselné hodnoty čím sú definované možné polohy (metafora kvantových čísel pri výstavbe elektrónového obalu atómu).

Význam veličín v uvedenej funkcií:

- L je počet vrstiev daného objektu
- i je index vrstvy indexovaný od 0 (vrstva najbližšia k jadrovému object-vláknu má index 0) do L – 1.
- j je index polohy v rámci vrstvy indexovaný od 0 do 5.
- $R_i(k)$ je polomer i-tej vrstvy. Je daný rekurzívnym vsťahom:

$$R_i(k) = R_{i-1} + r_{\max_{i-1}} + k + r_{\max_i}$$

kde:

- k je parameter algoritmu udávajúci vzdialenosť medzi vrstvami. Tento parameter nastavuje používateľ.
- $r_{\max_i}$ je polomer objekt-vlakna s najväčším polomerom na i-tej vrstve.

Výhodnosť polohy pre konkrétne objekt-vlákno súvisí so smerom vektora intenzity komunikácie tohto objekt-vlákna.

Súradnice danej polohy sa transformujú do pomocnej súradnicovej sústavy S', ktorej stred sa nachádza v strede objektu, jednotkový vektor v smere osi x smeruje smerom zhodným s vektorom intenzity komunikácie daného objekt-vlákna, súradnica z' smeruje rovnakým smerom ako súradnica z scény (nahor) a sústava je pravotočivá. Poloha je tým výhodnejšia, čím je jej súradnica x' väčšia. Ak existujú dve polohy s rovnakou súradnicou x', tak výhodnejšia je tá, ktorá má menšiu hodnotu |y'| (viď pseudokód).

Posledná vrstva je zaplnená čo najmenším počtom objekt-vlákien (takým, ktorý sa už nezmestí na nižšie vrstvy).

Nasleduje **pseudokód** algoritmu ARPV:

```
ARPV(Objekty, k:real)
{
     Foreach Objekt in Objekty
     {
          Foreach v in Objekt.PeriferneObjektvlakna
          {
               v.K = K⃗(v.Poloha, v.S, v.R);
```

```
            PrioritnyFront.Add(v, |v.K|);//kľúčom je |v.K|
        }
        Objekt.Polohy.Empty();
        While(PrioritnyFront.count > 0)
        {
            v = PrioritnyFront.Pop();
            (i,j) = NajvyhodnejsiaPoloha(Objekt, v.K, k);/*vracia
                                              dvojicu celých čísel*/
            Objekt.Polohy[i,j] = v;
        }
    }
}

NajvyhodnejsiaPoloha(Objekt,K:vektor,k:real) /*K je vektor intenzity
                                         komunikacie*/
{
    i, j, x‘, y‘; //celé čísla
    c = Objekt.PocetPerifernychObjketvlakien-6*Objekt.PocetVrstiev;
    for(a = Objekt.PocetVrstiev+1; a >= 0; a--)
    {
        If(a je navysia vrstva a uz c vlakien) continue;
        for(b = 0; b <= 6; b++)
        {
            If(Objekt.Polohy[a,b].IsEmpty == false) continue;
            If(K/|K| je nedefinovany vektor) return (a, b);
            S‘;//pomocna suradnicova sustava
            S‘.stred = Objekt.Poloha;
            S‘.i = K/|K|;//S’i,S‘.j,S’k sú jednotkové vektory osí S’
            S‘.k = S.k; //S je sustava sceny
            S‘.j = - (S‘.i × S‘.k); //vektorovy sucin
            Poloha‘ = S‘ << p⃗(a,b,Objekt.PocetVrstiev,k);/*trasformácia
                                         do súradnicovej sústavy S’*/
            If(Poloha‘.x‘ > x‘)
            {
                x‘ = Poloha‘.x‘; y‘ = Poloha‘.y‘;
                i = a; j = b;
            }
            Else if(Poloha‘.x‘ == x‘)
            {
               If(|Poloha‘.y‘| < |y‘|)
               {
```

```
                        y‘ = Poloha‘.y‘;
                        i = a; j = b;
                  }
              }
         }
     }
     return(i,j); //vracia dvojicu celých čísel (ako vektor)
}
```

8 Príklady aplikácií implementovaného vizualizačného sytému

V tejto časti práce uvádzame niekoľko ukážok vizualizácií dynamík programov napísaných v jazyku C# a komentáre k nim. Na týchto príkladoch demonštrujeme hlavné rysy vyvinutého systému.

8.1 *Binárny vyhľadávací strom*

Na tomto príklade demonštrujeme vizualizáciu znázorňujúcu:

- Vytváranie objektov v programe, strom vytvárania objektov.
- Volanie metód v jednovláknovom programe.
- Rekurziu
- Prístup k poliam objektov.
- Mechanizmus vyhodenia a šírenia výnimky.

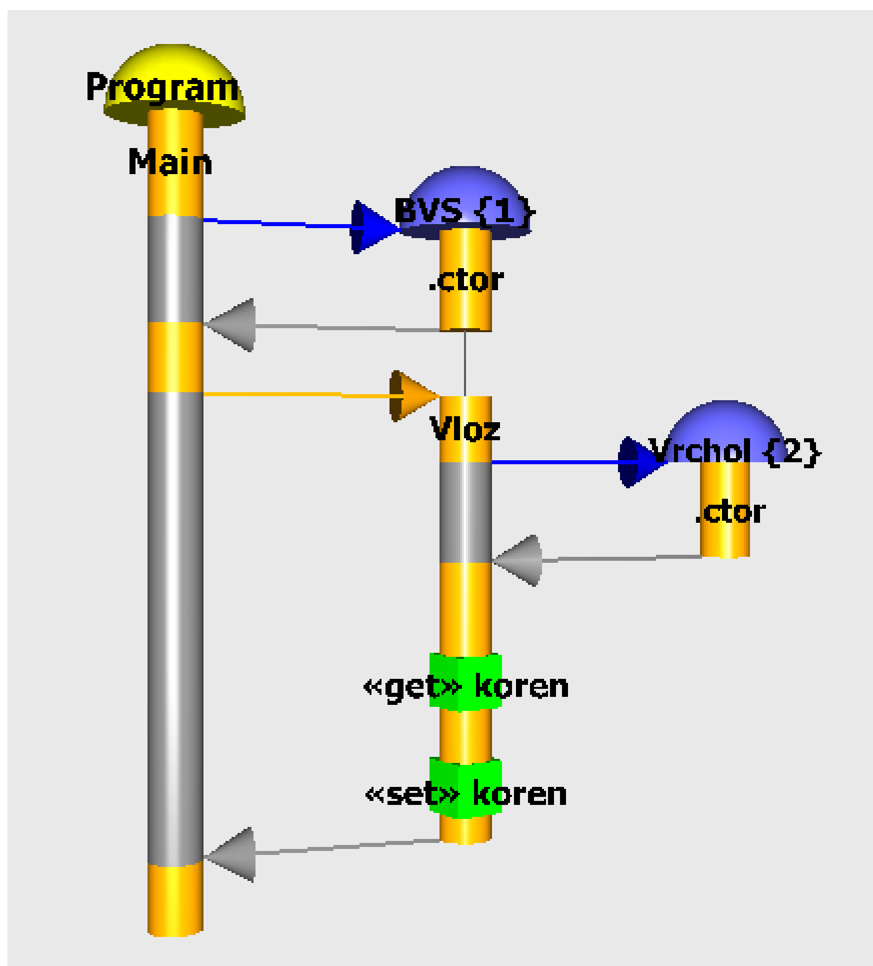
Budeme vizualizovať dynamiku nasledovného programu:

```
using System;
namespace TestBVS
{
    class Program
    {
        static void Main(string[] args)
        {
            BVS strom = new BVS();
            Console.WriteLine("Vkladam 5."); strom.Vloz(5);
            Console.WriteLine("Vkladam 1."); strom.Vloz(1);
            Console.WriteLine("Vkladam 7."); strom.Vloz(7);
            Console.WriteLine("Vkladam 10."); strom.Vloz(10);
            Console.WriteLine("Vkladam 9."); strom.Vloz(9);
            try
            {
                Console.WriteLine("Vkladam 10."); strom.Vloz(10);
            }
            catch (RovnakyKluc ex)
            {
                Console.WriteLine("Ošetrenie výnimky: " + ex.Message);
            }
            strom.VypisInorder();

            Console.WriteLine("Program skončil.");
            Console.Read();
        }
    }

    class Vrchol
    {
        public int kluc;
        public Vrchol lavy = null;
        public Vrchol pravy = null;
```

```
        public Vrchol(int kluc)
        {
            this.kluc = kluc;
        }
    }

    class BVS
    {
        private Vrchol koren = null;
        public void Vloz(int kluc)
        {
            Vrchol v = new Vrchol(kluc);

            if (this.koren == null) koren = v;
            else this.VlozRekurzia(this.koren, v);
        }
        private void VlozRekurzia(Vrchol podstrom, Vrchol novy)
        {
            if (novy.kluc < podstrom.kluc)
            {
                if (podstrom.lavy == null) podstrom.lavy = novy;
                else VlozRekurzia(podstrom.lavy, novy);

            }
            else if (novy.kluc > podstrom.kluc)
            {
                if (podstrom.pravy == null) podstrom.pravy = novy;
                else VlozRekurzia(podstrom.pravy, novy);
            }
            else
            {
                throw new RovnakyKluc();
            }
        }

        public void VypisInorder()
        {
            Console.Write("{ ");
            if(this.koren != null)
                this.VypisInorderRekurzia(this.koren);
            Console.WriteLine("}");
        }
        private void VypisInorderRekurzia(Vrchol Podstrom)
        {
            if(Podstrom.lavy != null)
                this.VypisInorderRekurzia(Podstrom.lavy);
            Console.Write(Podstrom.kluc.ToString() + " ");
            if (Podstrom.pravy != null)
                this.VypisInorderRekurzia(Podstrom.pravy);
        }

        public void ZmazStrom()
        {
            this.koren = null;
        }
    }
    class RovnakyKluc:Exception {}
}
```

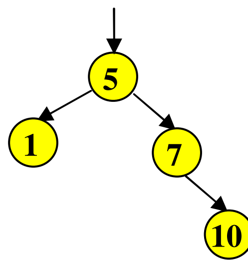
Na Obr. č. 32 vidíme začiatok vizualizácie dynamiky programu. Vidíme, že najprv bola spustená statická metóda *Main* triedy *Program*. To, že ide o statickú metódu vieme podľa toho, že hlavica "objektu" *Program* je žltá. Metóda *Main* následne vytvorila objekt triedy *BVS* pričom sa spustil jeho konštruktor. Počas vykonávania konštruktora bola metóda *Main* neaktívna, čo je znázornené šedým úsekom metódy. Po vykonaní konštruktora sa riadenie opäť vrátilo metóde *Main*, ktorá následne zavolala metódu *Vloz* nad objektom triedy *BVS*. Metóda *Vloz* vytvorila objekt triedy *Vrchol* a prečítala pole *koren* objektu triedy *BVS*, pričom zistila že strom je prázdny, preto do premennej *koren* nastavila referenciu na objekt triedy *Vrchol*, ktorý práve vytvorila.

Obr. č. 32 Vytvorenie BVS a vloženie prvej položky



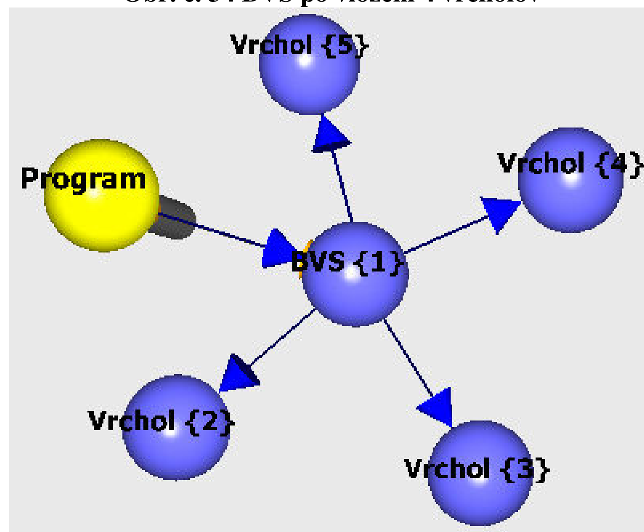
Program opakovane volá metódu *BVS::Vloz*, ktorá vytvára nové objekty triedy *Vrchol*. Po vložení kľúčov 5, 1, 7, 10 budú v programe 4 objekty triedy *Vrchol*, pričom všetky boli vytvorené objektom triedy *BVS*. Toto je znázornené na Obr. č. 34. Štruktúra BVS v tomto okamihu je znázornená na Obr. č. 33

Obr. č. 33 Štruktúra BVS pred vložením čísla 9

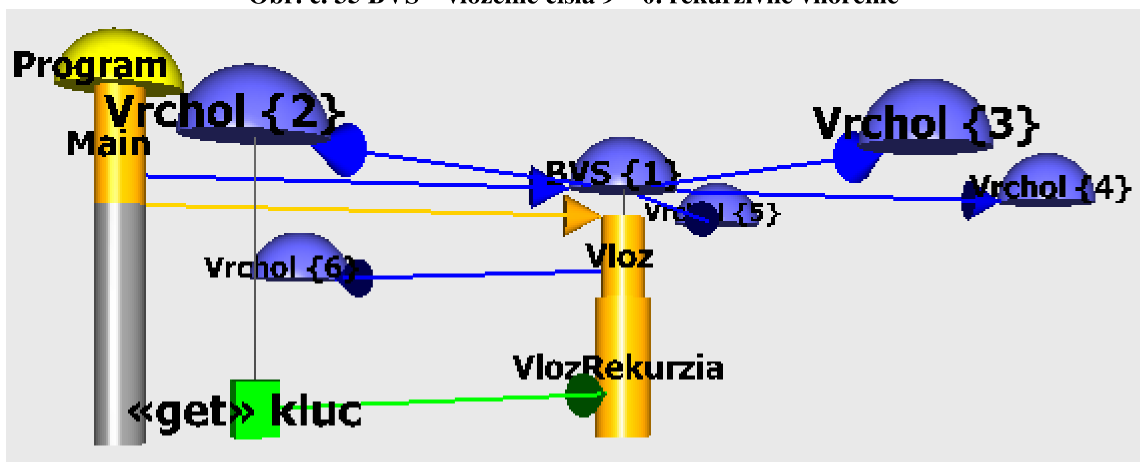


Následne do tohto stromu vložíme kľúč s hodnotu čísla 9. Na Obr. č. 35, Obr. č. 36 a Obr. č. 37 vidíme že metóda *BVS::Vloz* vytvorila 5. vrchol a následne zavolala rekurziu *BVS::VlozRekurzia*, ktorá prechádza binárnym vyhľadávacím stromom, pričom číta kľúče jednotlivých vrcholov. Až nakoniec (Obr. č. 38) vloží nový vrchol do ľavého podstromu vrcholu s kľúčom 10.

Obr. č. 34 BVS po vložení 4 vrcholov

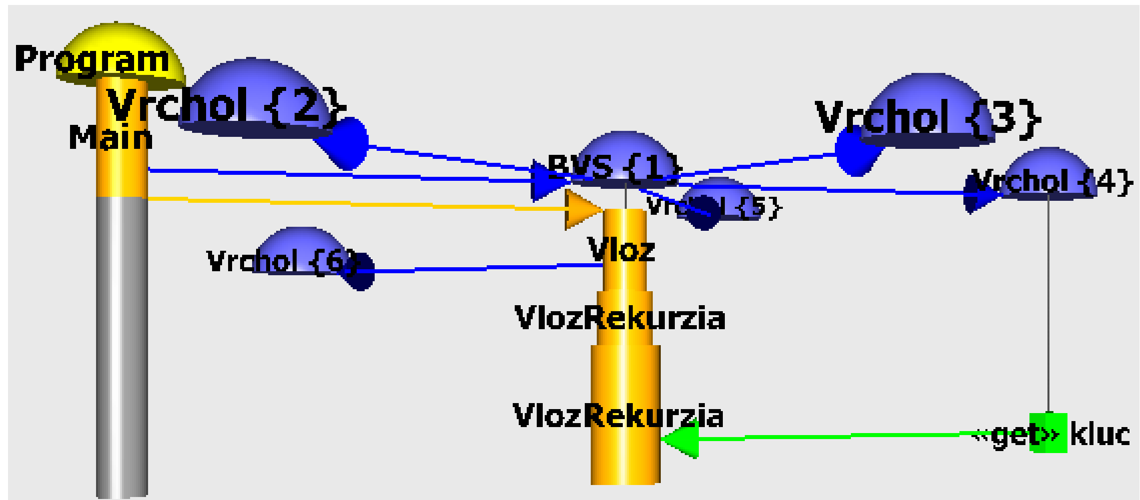


Obr. č. 35 BVS – vloženie čísla 9 – 0. rekurzívne vnorenie

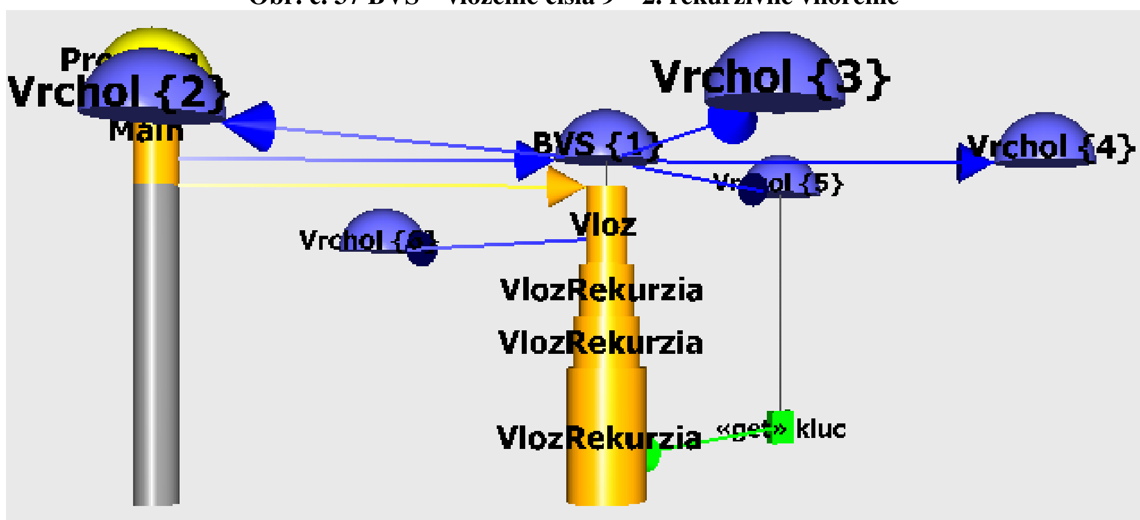


Na Obr. č. 39 vidíme, čo sa stane pri pokuse vložiť do stromu kľúč s už vloženou hodnotu. Pri treťom rekurzívnom vnorení sa zistilo, že daný kľúč už existuje a tak rekurzia vytvorila a následne vyhodila výnimku *RovnakyKluc*. Výnimku neošetrili predchádzajúce vnorenia rekurzie a ani metóda *Vloz*, preto bola vyhodená až k metóde *Main*, ktorá ju ošetrila a ďalej pokračovala volaním metódy *BVS::VypisInorder*. Celá cesta šírenia vyhodenej výnimky je zvýraznená červenou farbou. Vizuálne prázdne objekty sme kvôli prehľadnosti skryli.

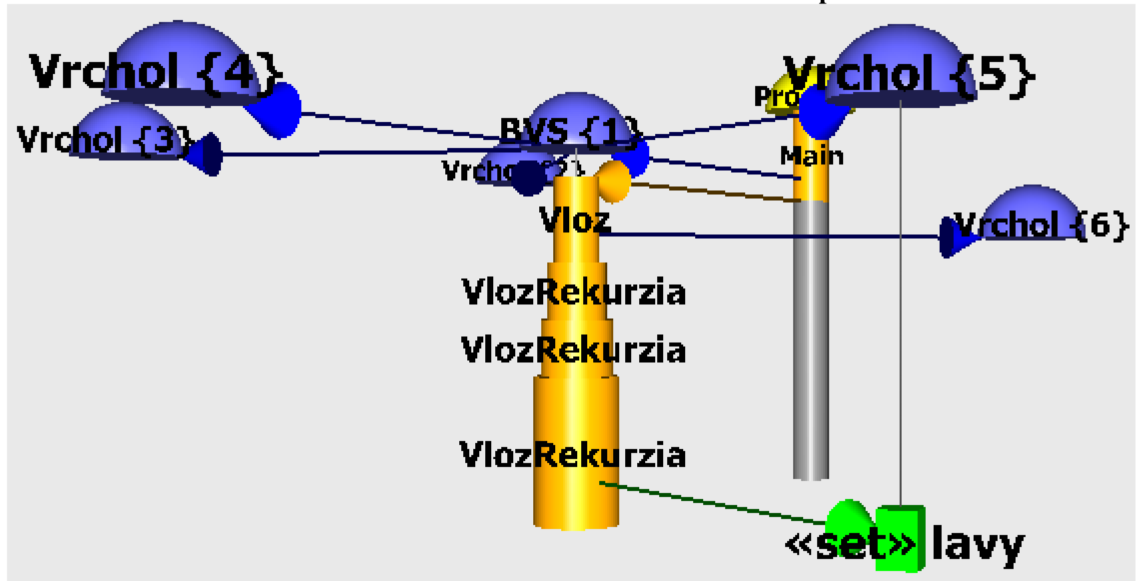
Obr. č. 36 BVS – vloženie čísla 9 – 1. rekurzívne vnorenie



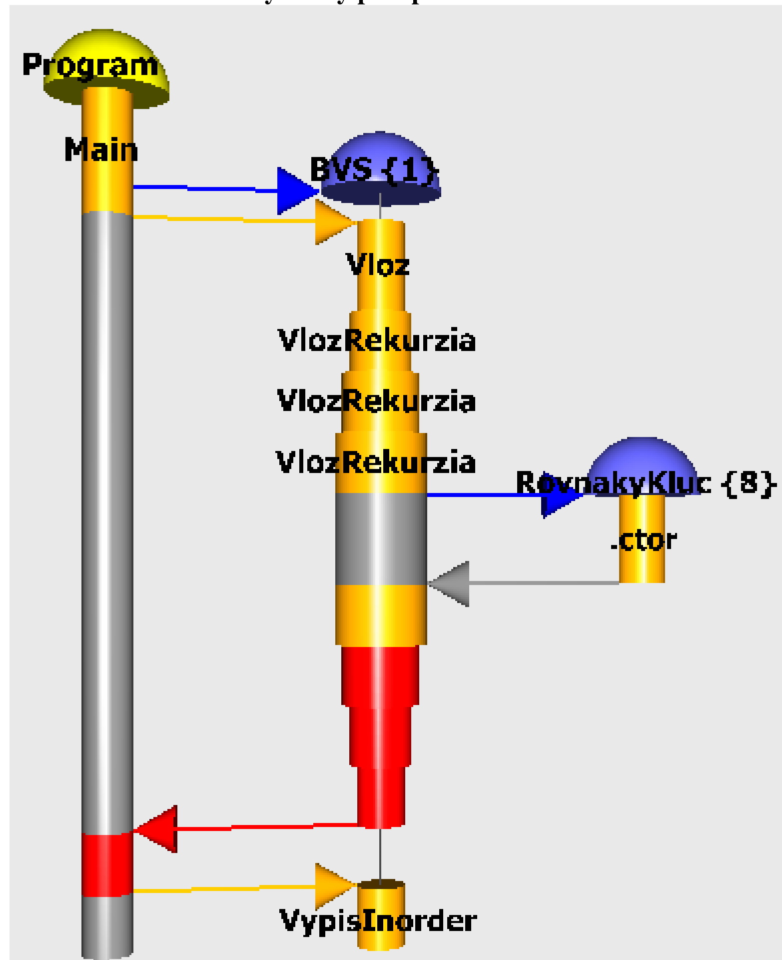
Obr. č. 37 BVS – vloženie čísla 9 – 2. rekurzívne vnorenie



Obr. č. 38 BVS – vloženie čísla 9 – Vloženie do ľavého podstromu

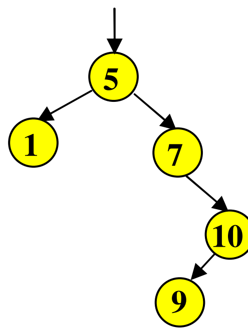


Obr. č. 39 BVS – Šírenie výnimky pri opakovanom vložení rovnakého kľúča



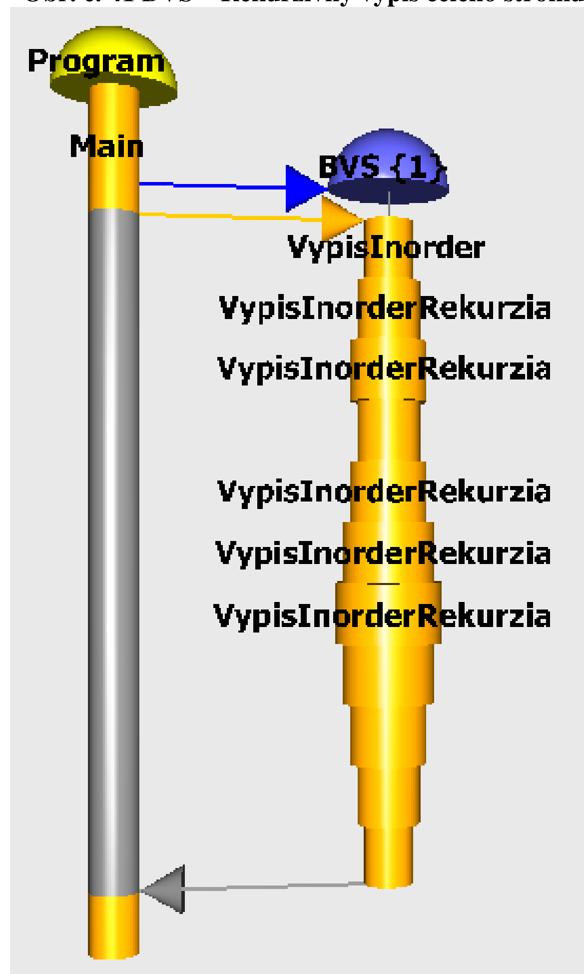
Na Obr. č. 41 vidíme vizualizáciu rekurzívneho výpisu stromu so štruktúrou znázornenou na Obr. č. 40.

Obr. č. 40 Štruktúra BVS počas výpisu



Rekurzia *BVS::VypisInorderRekuzia* začína v koreni stromu čo je 0. – té vnorenie. pokračuje najprv do ľavého podstromu, čo je prvé vnorenie. Toto je znázornené obalením 0. – tého vnornia prvým. Tam je však len 1 vrchol, takže následne sa vynára na úroveň 0. – tého vnorenia. Potom nasledujú až 3 vnorenia (obaľovania) vpravo a následné vynárania (odbaľovania). Môžeme vidieť že rekurzia vo svojom tele obsahuje (minimálne) dve rekurzívne volania, keďže 0. – té vnorenie je obalené až dva krát. Nakoniec rekurzia skončí a odbalí sa metóda *VypisInorder*, ktorá vráti riadenia metóde *Main*.

Obr. č. 41 BVS – Rekurzívny výpis celého stromu



8.2 Behaviorálny návrhový vzor Observer a viacvláknová aplikácia

Program ktorý budeme vizualizovať ukazuje:

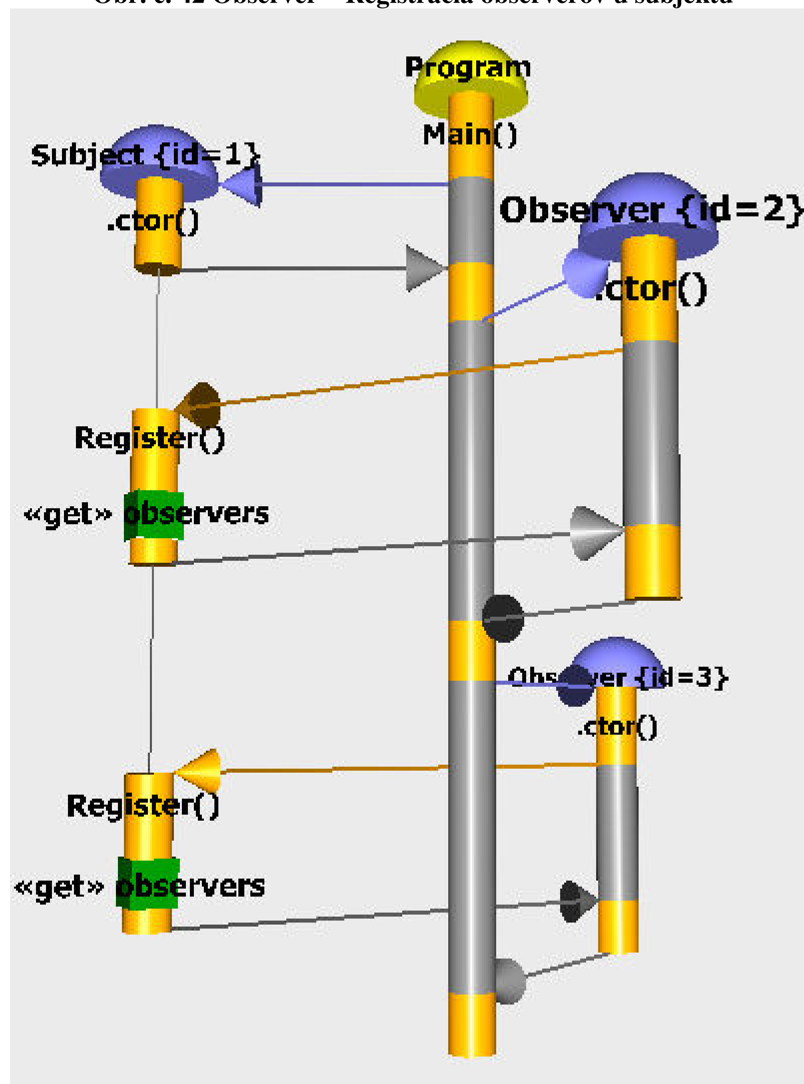
- príklad vizualizácie návrhového vzoru
- vizualizáciu viacvláknového programu
- zobrazovanie informácií o prvkoch vizualizácie v dialógovom okne vyvolanom cez kontextové menu daného prvku

Nasleduje zdrojový kód vizualizovaného programu:

```
using System;
using System.Collections.Generic;
using System.Threading;
namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            Subject s = new Subject();
            Observer o1 = new Observer(s);
            Observer o2 = new Observer(s);

            Thread vlakno = new Thread(new ThreadStart(s.ChangeState));
            vlakno.Start();

            Console.WriteLine("Metóda Main skončila.");
        }
    }
    public interface IObserver
    {
        void Update();
    }
    class Observer:IObserver
    {
        public Observer(Subject monitor)
        {
            monitor.Register(this);
        }

        public void Update()
        {

        }
    }
    class Subject
    {
        private List<IObserver> observers = new List<IObserver>() ;
        public Subject()
        {

        }
        public void Register(IObserver observer)
        {
            this.observers.Add(observer);
```

```
        }
        public void ChangeState()
        {
            this.Notify();
        }
        private void Notify()
        {
            foreach(IObserver o in this.observers) o.Update();
        }
    }
}
```

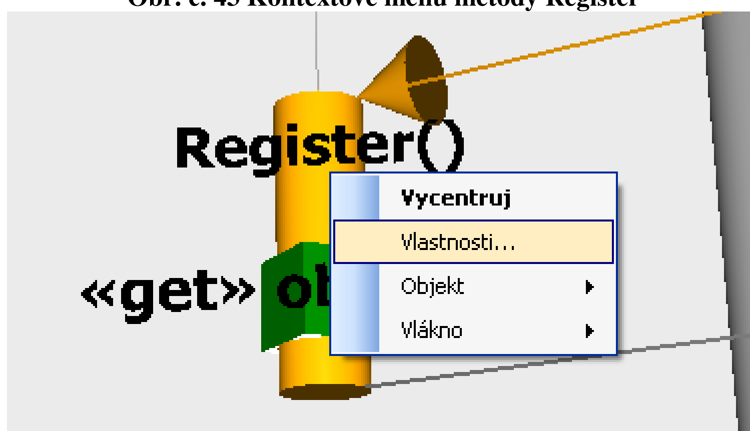
Obr. č. 42 Observer – Registrácia observerov u subjektu



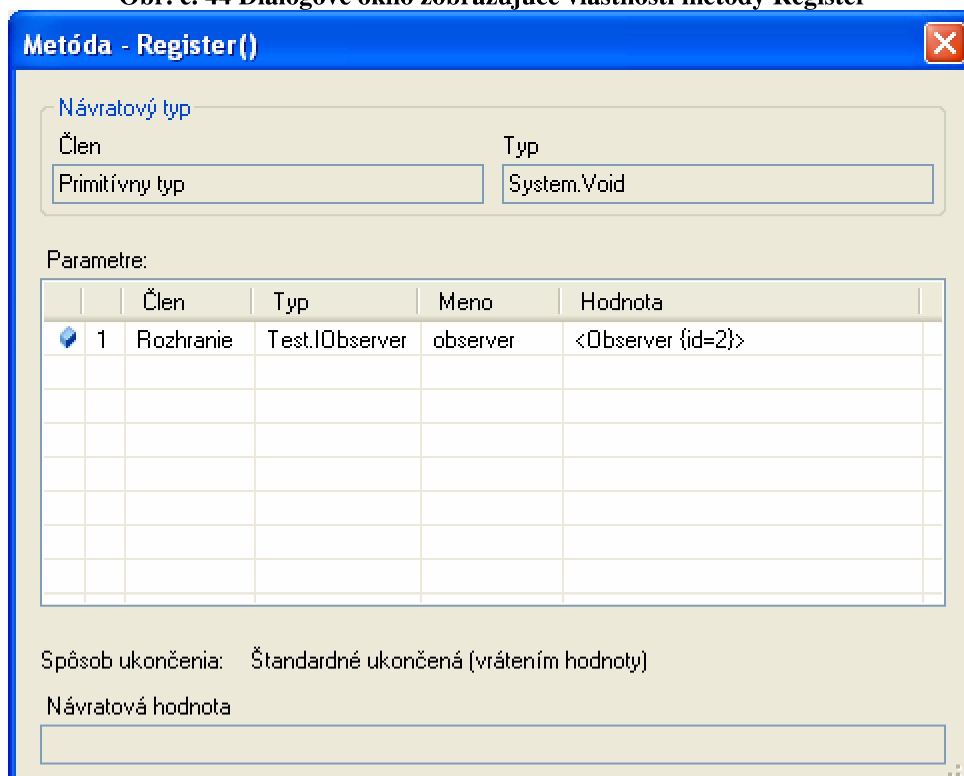
Na Obr. č. 42 vidíme, že statická metóda *Main* triedy *Program* (vstupný bod do programu) najskôr v primárnom vlákne programu vytvorila subjekt a následne dva objekty observerov. Vidíme, že súčasne s vytvorením každého observera je vykonaný jeho konštruktor, ktorý volá metódu subjektu *Register*, čím sa registruje u subjektu aby ho ten mohol neskôr notifikovať. Po kliknutí pravým tlačidlom myši na príslušnú metódu *Register* v scéne, sa vyvolá kontextové menu príslušného prvku (v tomto

prípade metódy), ktoré je zobrazené na Obr. č. 43 a ktoré umožňuje vyvolať dialógové okno (viď Obr. č. 44) udávajúce doplňujúce informácie o danom prvku. V prípade metódy je okrem iného uvedený aj zoznam jej parametrov, z ktorého môžeme vidieť, že metóda *Register* má práve jeden parameter typu IObserver, cez ktorý jej registrujúci sa observer poskytuje referenciu na samého seba. Podobne je možné získať informácie aj o objektoch, objekt-vláknach a premenných v scéne. Na obrázku Obr. č. 42 ďalej vidíme, že metóda Register číta premennú subjektu *observers* obsahujúcu referenciu na zoznam registrovaných observerov (pridáva referenciu na registrujúci sa observer do zoznamu).

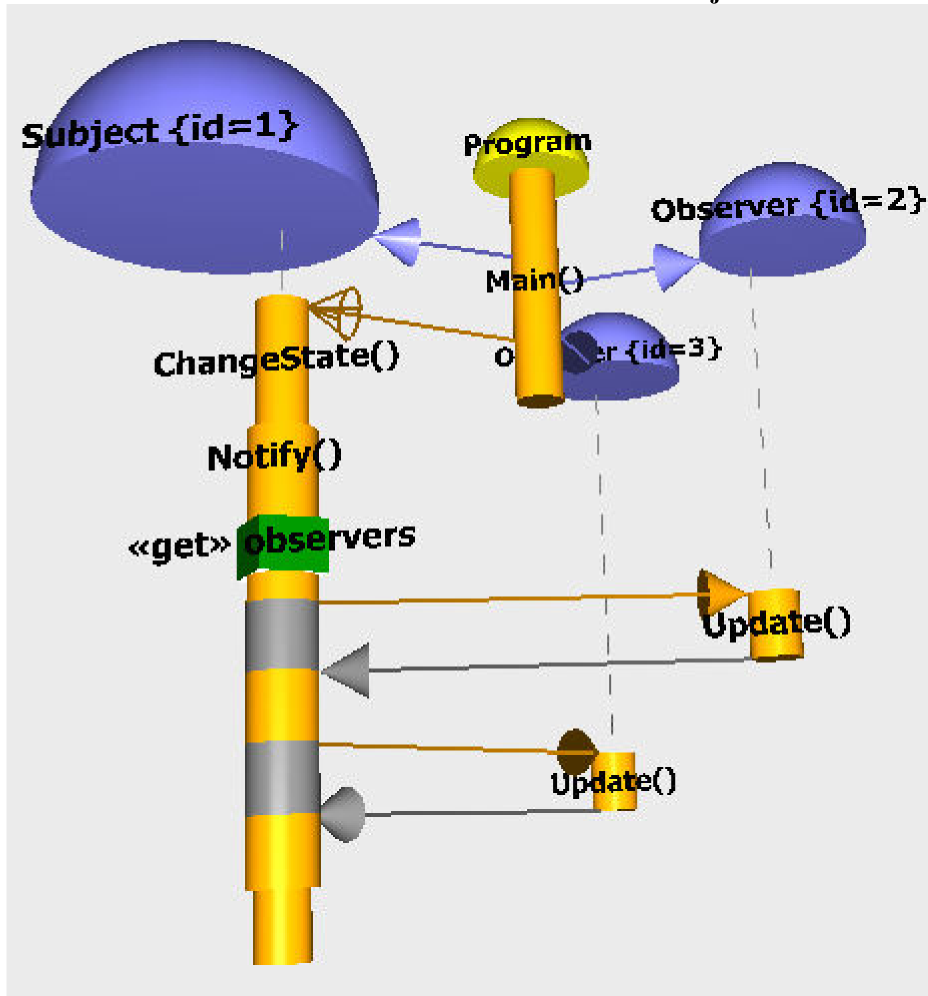
Obr. č. 43 Kontextové menu metódy Register



Obr. č. 44 Dialógové okno zobrazujúce vlastnosti metódy Register



Obr. č. 45 Observer – Notifikácia observerov subjektom



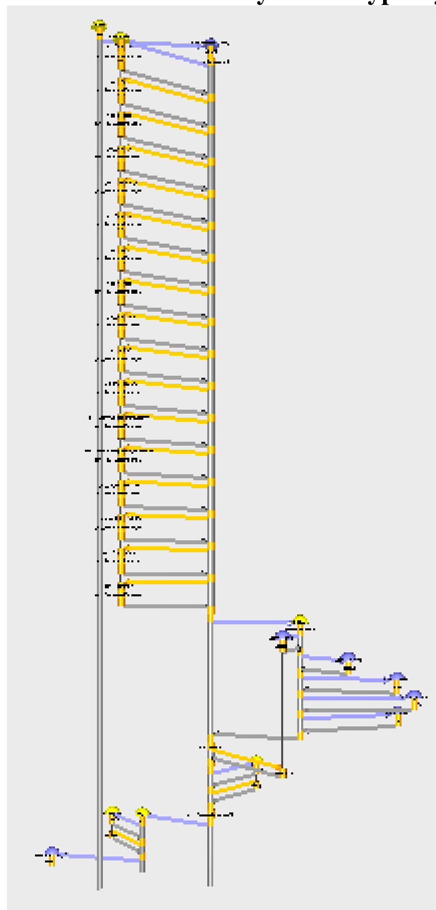
Na Obr. č. 45 vidíme, že metóda *Main*, bežiaca v primárnom vlákne programu, ďalej asynchrónne volá metódu subjektu *ChangeState*, čím vytvára sekundárne vlákno. Že ide o asynchrónne volanie je znázornené odlišným tvarom hrotu šípky predstavujúcej volanie. Keďže metóda *ChangeState*, je vykonávaná inom vlákne ako v tom v ktorom bol vytvorený objekt subjektu, je táto metóda umiestnená do periférneho (nejadrového) objekt-vlákna objektu subjekt – objekt-vlákno sa nenachádza v strede objektu, ale na okraji. Vidíme že metóda Main nečaká na dokončenie metódy ChangeState, ale istý čas beží paralelne s ňou – obe metódy sú v rovnakom čase aktívne (vyfarbené oranžovou farbou, nie neaktívnou šedou). Ďalej vidíme, že metóda *Main* končí (čím končí celé primárne vlákno programu) a ďalej beží už len metóda *ChangeState* (beží už len sekundárne vlákno). Následne metóda *ChangeState* volá metódu subjektu (metódu toho istého objektu) *Notify* a táto číta premennú subjektu *observers* a notifikuje jednotlivé observery synchrónnym volaním ich metódy *Update*. Tieto volania sa nedejú vo vlákne, ktoré vytvorilo objekty observerov, preto sú

umiestnené do periférnych objekt-vlákien. Ďajej vidíme ukončenie metódy Notify a následne metódy ChangeState, čím končí sekundárne vlákno programu a zároveň aj program samotný.

8.3 Demonštrácia riešenia problému škálovatelnosti aplikáciou navrhnutých obmedzení

Bez použitia nami navrhnutých dynamických obmedzení priestoru a času vizualizácie, ktoré popisujeme v kapitole 7.1.4, by vizualizácia bola použiteľná len pre reprezentáciu vykonávania programov na veľmi krátkych škálach ich dĺžky.

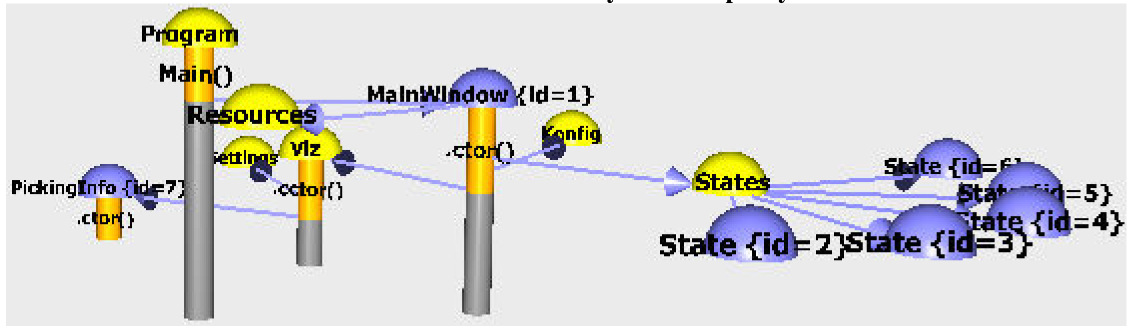
Obr. č. 46 Zásobníkový mód – vypnutý



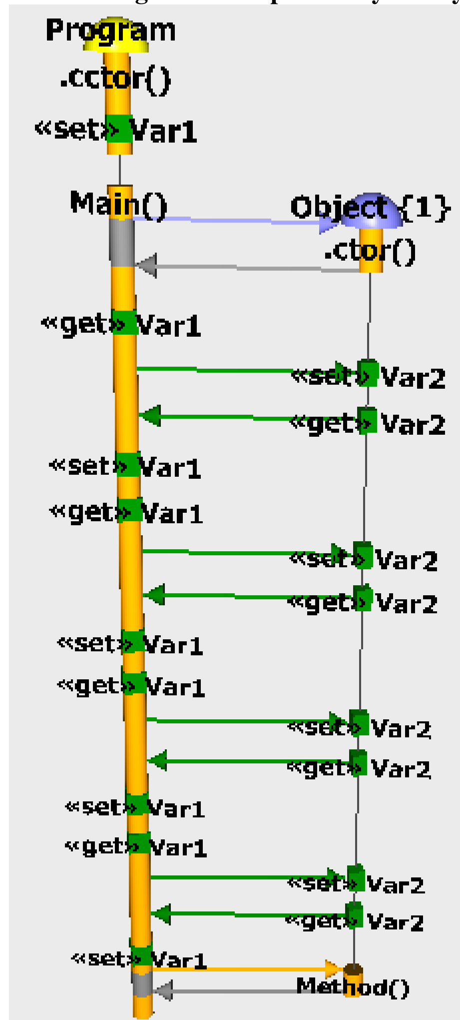
Na Obr. č. 46 vidíme, že bez použitia zásobníkového módu sa scéna v **horizontálnom** smere (v dimnezií z) veľmi rýchlo stane tak rozsiahla, že aby ju bolo možné vidieť celú (a získať tak prehľad o celkovej situácií) bolo by potrebné sa tak veľmi oddialiť, že by bolo veľmi obtiažne rozoznávať jednotlivé detaily. Ďalej by sa táto sa táto situácia veľmi rýchlo zhoršovala. V prípade zapnutého zásobníkového módu

(viď Obr. č. 47) sa nezobrazujú metódy, ktoré už boli ukončené (a premenné, ku ktorým bolo pristúpené v minulosti). Takto je vynechaný značný priestorový interval a tak je scéna podstatne menej rozsiahla a prehľadná.

Obr. č. 47 Zásobnikový mód - zapnutý

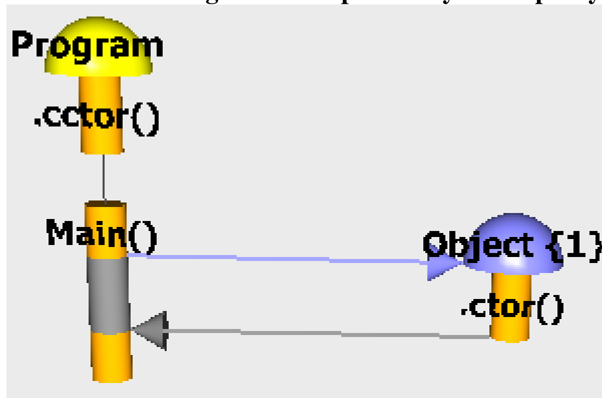


Obr. č. 48 Mód ignorovania premenných – vypnutý



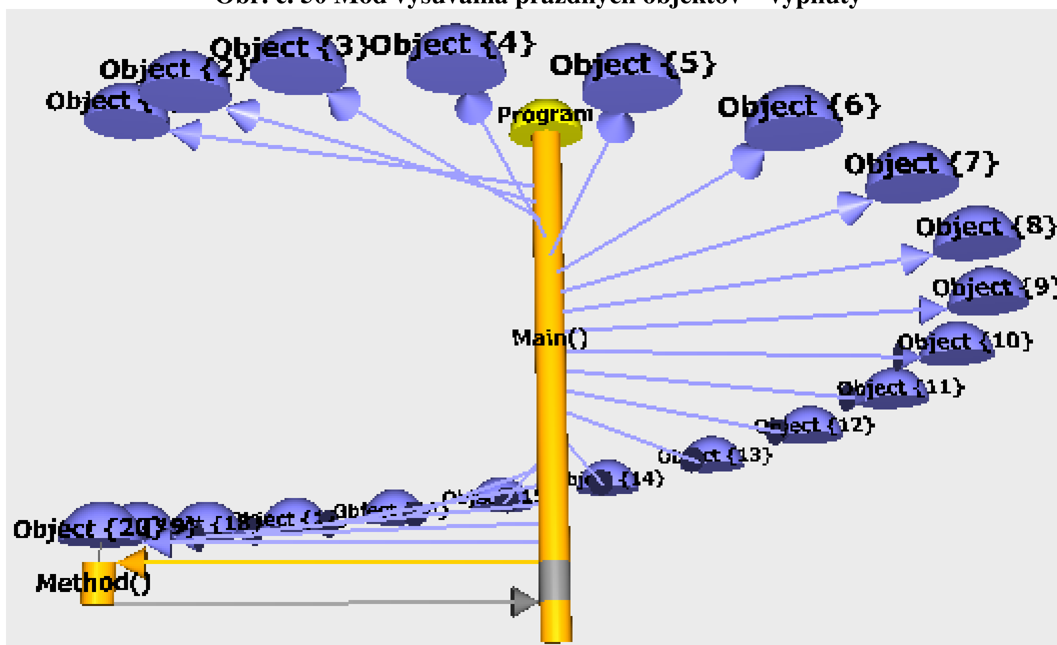
Obr. č. 48 a Obr. č. 49 ukazujú, že podobný (ak keď nie až tak rozsiahly) efekt, má tiež vypnutie zobrazovania prístupu k premenným. Toto obmedzenie okrem priestoru šetrí tiež **čas** priebehu vizualizácie (ktorá je animáciou), pretože sa počas nej nečaká na postupné zobrazenie prístupu k premenným, ktorých môže byť veľké množstvo.

Obr. č. 49 Mód ignorovania premenných – zapnutý

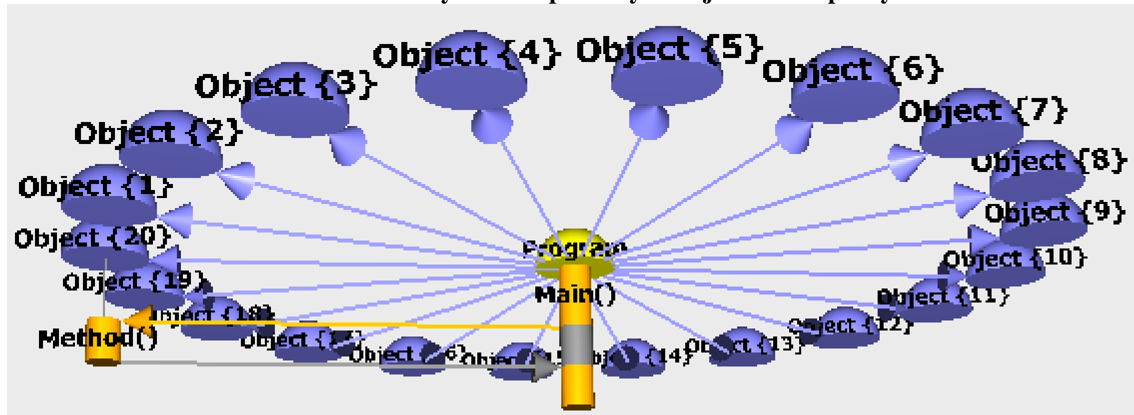


Skrátiť vizualizovaný vertikálny interval metódy, ktorá vytvára väčšie množstvo objektov vizualizovaných len hlavicami, je možné aplikáciou obmedzenia znázorneného na Obr. č. 50 a Obr. č. 51.

Obr. č. 50 Mód vysúvania prázdnych objektov – vypnutý

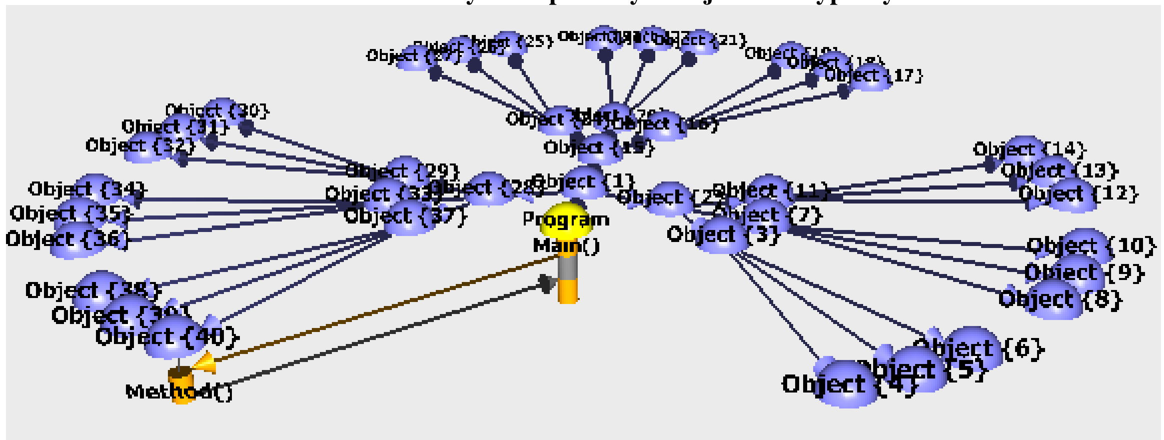


Obr. č. 51 Mód vysúvania prázdnych objektov – zapnutý

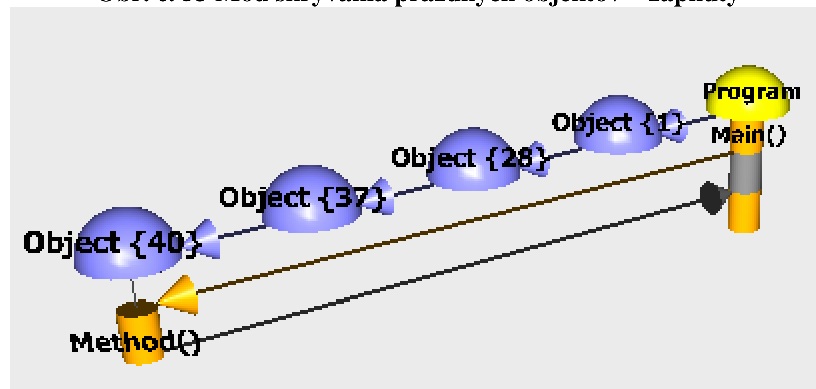


Obr. č. 52 a Obr. č. 53 ukazujú ako je možné scénu sprehľadniť obmedzením priestoru v **horizontálnom** smere (v dimenziách x, y). Tiež je ukázané, že aplikáciou tohto obmedzenia nestrácame informácie o pôvode a vzájomných vzťahoch dôležitých (práve komunikujúcich) objektov.

Obr. č. 52 Mód skrývania prázdnych objektov – vypnutý



Obr. č. 53 Mód skrývania prázdnych objektov – zapnutý



9 Záver

Navrhli sme koncept vizualizácie dynamiky objektovo – orientovaného programu založený na modifikáciách UML sekvenčného diagramu. Navrhnutá vizualizácia je animácia v 3D priestore.

Navrhli sme vizualizačný systém implementujúci navrhnutý koncept vizualizácie. Vizualizačný systém zabezpečuje monitorovanie dynamiky programu napísaného v jazyku C#, je však používateľom rozšíriteľný aj pre iné jazyky a spôsoby monitorovania. Vizualizačný systém zabezpečuje prenos získaných informácií z procesu sledovaného programu do procesu vizualizujúceho programu, kde ich v reálnom čase (synchrónne) zobrazuje. Používateľ vizualizačného programu môže riadiť rýchlosť vykonávania sledovaného programu a krokovať ho. Tiež je možná asynchrónna vizualizácia práve bežiaceho sledovaného programu, aby tento nebol spomaľovaný. Používateľ si môže zvoliť vhodné obmedzenia času a priestoru vizualizácie tak, aby sa zobrazovali len pre neho dôležité časti a vizualizácia zostávala prehľadná.

Za účelom implementácie vizualizačného systému sme tiež navrhli algoritmy rozmiestnenia objektov (v lineárnom čase, čo umožňuje ich preusporiadavanie v reálnom čase) a vlákien v priestore.

Vizualizačný systém sme implementovali v jazyku C# s využitím .NET rámca pre podporu aspektovo-orientovaného programovania PostSharp, grafickej knižnice OpenGL a komponenty SharpGL, ktorá umožňuje pohodlné volanie funkcií OpenGL z programu napísaného v jayzku C#. Výsledný zdrojový kód má rozsah cca 10 000 riadkov a obsahuje 96 tried.

Vyvinutý systém sme overili monitorovaním a vizualizovaním dynamiky programov implementujúcich binárny vyhľadávací strom, návrhový vzor Pozorovateľ (anglicky Observer) a monitorovaním samého seba.

Riešenie sme publikovali na medzinárodnej vedeckej konferencii **Spring Conference on Computer Graphics** 2013 (SCCG) článkom:

Grznár F. – Kapec, P.: Visualizing dynamics of object oriented programs with time context.

Zborník z tejto konferencie plánuje publikovať **ACM Publishing House**. Článok uvádzame v prílohách k tejto diplomovej práci.

Nami navrhnutý spôsob vizualizácie **rozširuje UML sekvenčný diagram nasledovne**:

- Definuje umiestnenie objektov zúčastňujúcich sa komunikácie na základe ich "vytváracích" vzťahov (dcérske objekty sa nachádzajú v okolí rodičovského objektu, ktorý ich vytvoril). Vzniká tak zároveň graf znázorňujúci postupnosť vytvárania objektov.
- Zobrazuje vlákna bežiace v programe. Jednotlivé metódy vizuálne priraďuje okrem objektov ,nad ktorými sú vykonávané zároveň aj k vláknam, v ktorých sú vykonávané.
- Odlišuje časť metódy, ktorá je aktívne vykonávaná od častí metódy, ktorá čaká na dokončenie inej metódy.
- Odlišuje časť metódy, ktorá vyhadzuje výnimku, čím znázorňuje šírenie výnimiek.
- Zobrazuje tiež čítanie a nastavovanie členských premenných objektov.
- Definuje znázornenie nepriameho volania.

Oproti práci (Nedecký, 2010) nami zrealizovaný vizualizačný systém prináša najmä tieto vylepšenia:

- Umožňuje vizualizáciu v reálnom čase (synchrónne) a riadenie toku sledovaného programu používateľom vizualizačného programu. (Je však možná aj asynchrónna vizualizácia ,aby sledovaný program nebol spomaľovaný). Používateľ tak nie je obťažovaný vytváraním záznamu, ktorý by následne nechal vizualizovať.
- Zobrazuje časový kontext – teda nie len jednu (aktuálnu) správu ,ale aj správy zaslané v minulosti, čo významným spôsobom rozširuje množstvo poskytnutých informácií v jednom časovom okamihu.
- Umožňuje monitorovať programy napísané v jazyku C#.

Hoci sme splnili ciele, ktoré sme si v tejto práci stanovili, implementovaný vizualizačný **systém je možné ďalej rozvíjať** napríklad v týchto smeroch:

- Definovať vlastný formát súboru záznamu získaných trasovacích informácií, ktorý by bolo možné neskôr prehrať bez nutnosti spustenia sledovaného programu
- Rozšíriť monitorovanie aj na programy napísané v iných objektovo – orientovaných jazykoch (môže to urobiť aj samotný používateľ)
- Umožniť zmenu hodnôt členských premenných a tiel metód v sledovanom programe počas jeho behu používateľom vizualizačného programu.

- Priestor vnútornej komunikácie objektu oddeliť od okolia polopriehľadným valcom, čo by mohlo vizualizáciu ešte viac sprehľadniť.

Okrem **využitia** implementovaného vizualizačného systému, ktorý sme pomenovali SoftDynamik, ako CASE nástroja vývojármi, vidíme veľký potenciál v jeho využití pri výučbe objektovo – orientovaného programovania na stredných a vysokých školách.

Za významný **osobný prínos** tejto diplomovej práce považujeme tiež skutočnosť, že sme v rámci jej vypracovania získali nasledovné vedomosti a osobne sme po prvý krát vyskúšali nasledovné technológie:

- PostSharp vo verzií 2.1 (.NET rámec podporujúci aspektovo – orientované programovanie)
- Vzdialené volanie metód (RMI) – konkrétne pomocou .NET Remoting. Základ pre tvorbu distribuovaných aplikácií a elegantná možnosť komunikácie medzi procesmi.
- Čítanie a modifikácia XML súboru (pomocou DOM) a formát projektového súboru jazyka C# (*.csproj).
- Adaptácia predchádzajúcich znalostí ohľadom práce s bitmapou vo Windows API do prostredia .NET Framework (tlač, kopírovanie do schránky Windows, uloženie do súboru).
- Vizuálna dedičnosť (dedenie medzi oknami) v jazyku C# .
- Rozšírenie vedomostí v oblasti počítačovej grafiky a technológie OpenGL:
    - Transformačné matice a priama práca s nimi.
    - Detekcia kliknutia na vykreslený objekt technológiou “picking“.
    - Vykresľovanie 3D písma.
    - Osvetlenie
    - Otáčanie scény myšou technikou “arc-ball”.
    - Orezávanie telies rovinou.
    - Kreslenie prerušovanej čiary.

Zoznam použitej literatúry

- Arlow, J. - Neustadt, I.: **UML 2 a unifikovaný proces vývoje aplikací**. 1. vyd. Brno: Computer Press, 2007. 364 s. ISBN 978-80-251-1503-9.
- Bieliková, M.: **Softvérové inžinierstvo: Princípy a manažment.** 1. vyd. Bratislava: STU v Bratislave, 2000. 220 s. ISBN 80-227-1322-8.
- Booch, G. – Jacobson, I. – Rumbaugh, J.: **OMG Unified modeling language specification**. 1. vyd. Object Management Group, 2000.
- Caserta, P. – Zendra, O. – Bodenes, D.: **3D Hierarchical Edge bundles to visualize relations in a software city metaphor**. In: 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011), IEEE, 2011, s. 1-8.
- Diehl, S.: **Software Visualization: Vizualizing the Structure, Behaviour and Evolution of Software**. New York: Springer 2007. 187 s. ISBN 978-3-540-46504-1.
- Fleming, S. – Kraemer, E. – Stirewalt, R. – Dillon, L.: **Debugging Concurrent Software: A Study Using Multithreaded Sequence Diagrams**. In: Symposium on Visual Languages and Human-Centric Computing, IEEE, 2010.
- Fowler, M.: **Destilované UML**. 1. vyd. Praha: Grada Publishing, 2009. 176 s. ISBN 978-80-247-2062-3.
- Gamma, E. - Helm, R. - Johnson, R. - Vlissides, J.: **Návrh programu pomocí vzoru: Stavební kameny objektově orientovaných programu**. Grada, 2003. 338 s. ISBN 8024703025.
- Hope, G. – Woolf, B.: **Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions**. Addison-Wesley, 2004.
- Krajčovič, T.: **Počítače**. 2. vyd. Bratislava: STU v Bratislave, 2000, 157 s. ISBN 80-227-1399-6.
- Meško, D. – Katuščák, D. – Findra, J.: **Akademická príručka**. Bratislava: Osveta, 2004. 316 s. ISBN 80-8063-200-6.
- Mehner, K.: **JaVis: A UML-Based Visualization and Debugging Environment of Concurrent Java Programs**. In: Software Visualization, volume 2269 of LNCS State-of-the-Art Survey. Springer Verlag, 2002, s. 164-175.

- Mehner, K. – Wagner, A.: **Visualizing the synchronization of java-threads with UML**. In Visual Languages. In: 2000 IEEE International Symposium, IEEE, 2000, s. 199 – 206.
- Nedecký, R. I.: **Vizualizácia vykonávania programu v 3D priestore**. [Diplomová práca]. Bratislava: FIIT STU v Bratislave, 2010, 65 s.
- Smith, M. P. – Munro, M.: **Runtime Visualisation of Object Oriented Software**. In: Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02) 0-7695-1662-9/02. IEEE, 2002.
- Šešera, Ľ. – Gerec, P. – Návrat, P.: **Architektúra softvérových systémov Architektúra internetových systémov a architektúra orientovaná na služby**. Bratislva: Slovenská technická univerzita v Bratislave, 2011. 385 s. ISBN 978-80-227-3546-9.
- Teyseyre, A. – Campo, M.: **An overview of 3D software visualization**. In: IEEE transactions on visualization and computer graphics, vol. 15, no. 1, January – February 2009, IEEE Computer Society, 2009. s. 87 – 105.
- Vaníček, J. – Papík, M. – Pergl, R. – Vaníček T.: **Teoretické základy informatiky**. 1. vyd. Praha: Kernberg Publishing, 2007. 436 s. ISBN 978-80-903962-4-1.
- Vranič, V.: **Objektovo – orientované programovanie Objekty, Java a aspekty**. Bratislava: Slovenská technická univerzita v Bratislave, 2008. 223 s. ISBN 978-80-227-2830-0.

Prílohy

Príloha č. 1 Obsah DVD

DVD je dôležitou integrálnou súčasťou práce. Obsahuje:

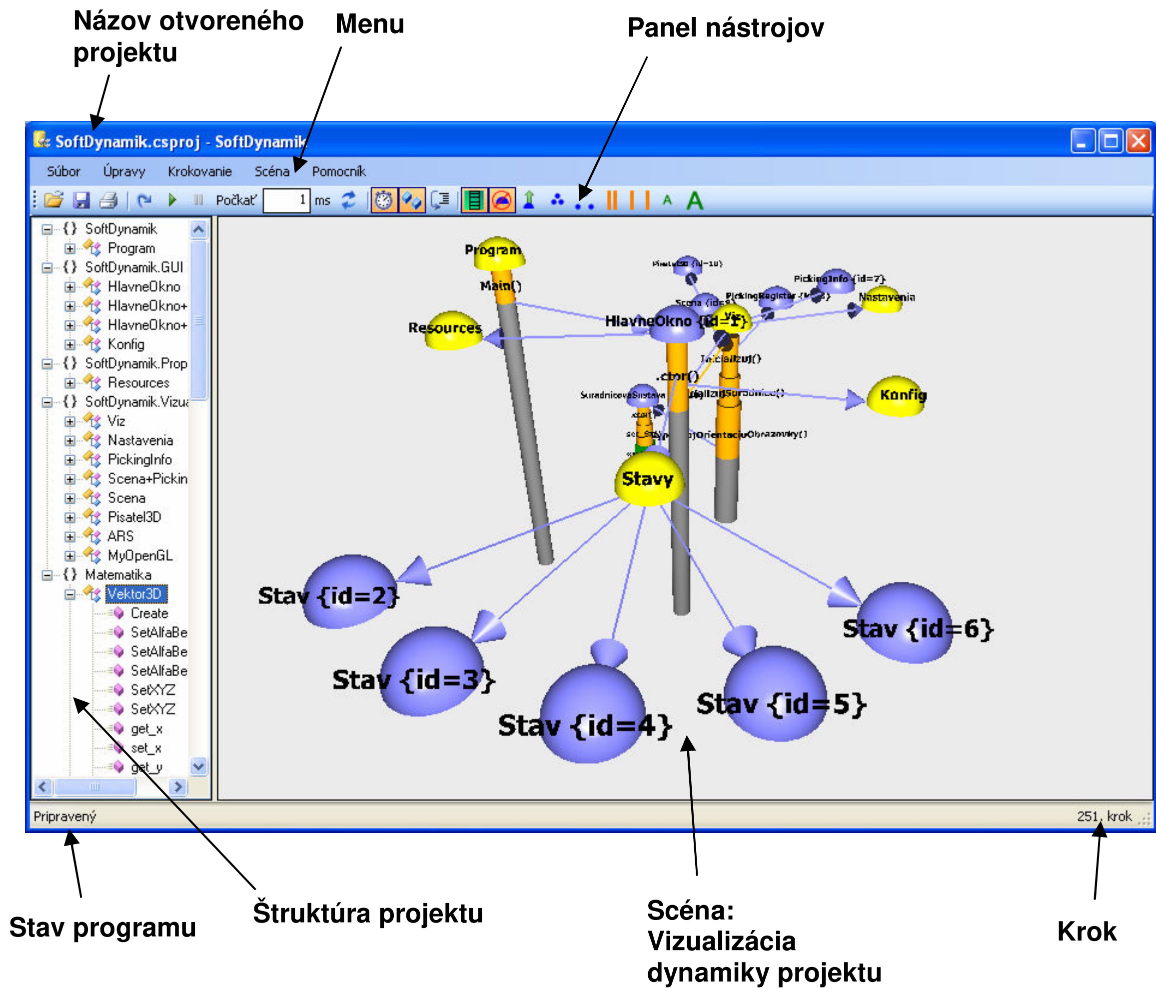
- Inštalátor nami vyvinutého systému SoftDynamik (vyžaduje .NET Framework 2.0)
- Zdrojový kód nami vyvinutého systému SoftDynamik (IDE: Microsoft Visual Studio 2005 a novšie)
- Technickú dokumentáciu vo formáte EAP (Enterprise Architect 8.0)
- Inštalátori .NET rámcov podporujúcich aspektovo – orientované programovanie:
    - PostSharp 1.5 (pre .NET 2.0)
    - PostSharp 2.1 (pre .NET 3.5, .NET 4.0)

  Tieto rámce sú potrebné k správnemu fungovaniu systému SoftDynamik (presnejšie k správnemu fungovaniu trasovača dodaného so systémom).
- Textovú časť diplomovej práce vo formátoch DOC, PDF

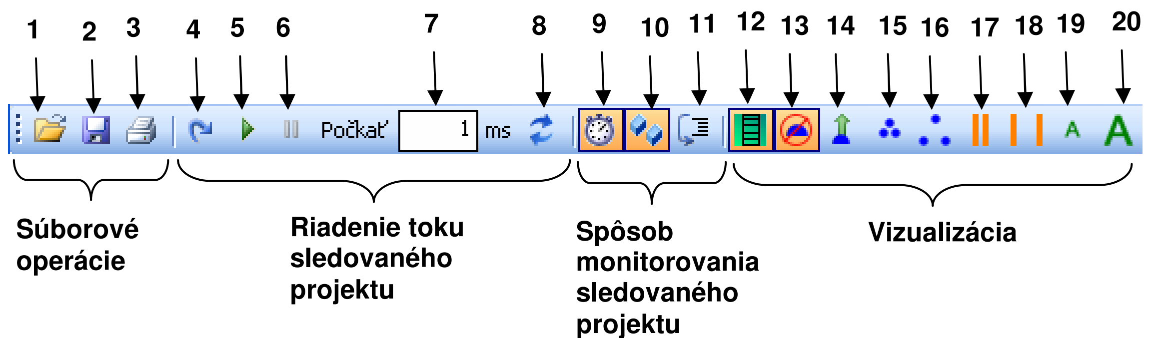
Príloha č. 2 Používateľská príručka

SoftDynamik

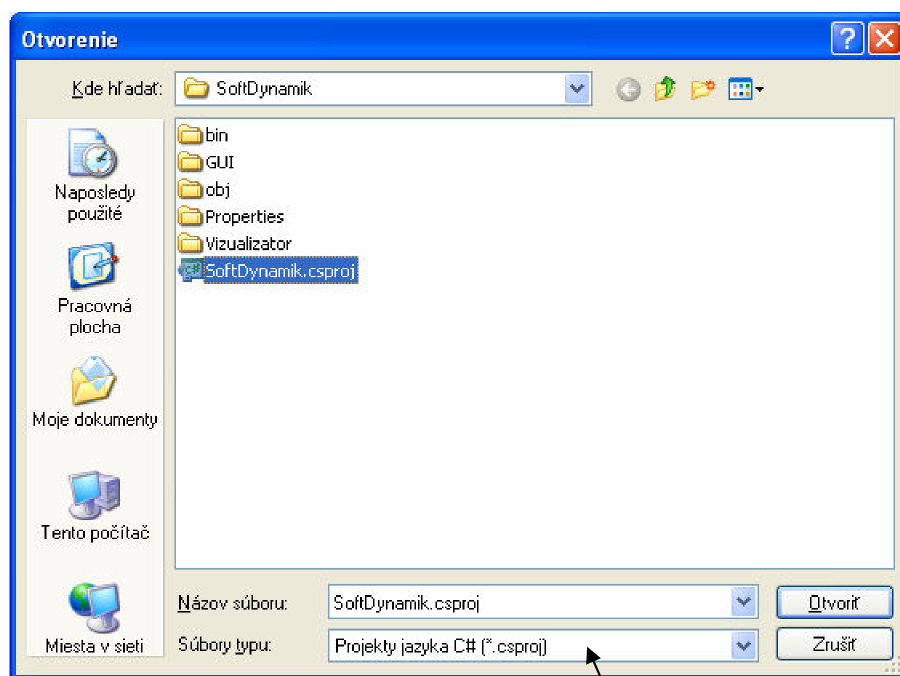
Používateľská príručka



Panel nástrojov



1 **Otvorenie** nového projektu:



2 **Uloženie** scény do súboru obrázka BMP
3 **Vytlačenie** scény

4 **Dopredu** len o 1 krok
5 Spustiť **automatické krokovanie**
6 **Pozastaviť**
7 Nastavenie **doby čakania** v milisekundách medzi jednotlivými krokmi automatického krokovania
- Po zmene číselnej hodnoty stlačte Enter.

8 Spustiť sledovaný program **od začiatku** a pozastaviť

9 Zapnúť mód **synchrónneho monitorovania** sledovaného programu
10 Zapnúť mód monitorovania **prístupu k členským premenným** objektov
11 Zapnúť **preskakovací mód**:
- nebudú sa monitorovať volania ďalších metód a konštruktorov
- nebude sa monitorovať prístup k ďalším členským premenným

12 Zapnúť **zásobníkový mód**:
- Metódy zavolané od okamihu zapnutia módu, budú po skončení (po odstránení zo zásobníku volaní operačného systému) odstránené zo scény
- Členské premenné ,ku ktorým bolo pristúpené od okamihu zapnutia módu budú v nasledujúcom kroku zo scény odstránené.

13 Zapnúť **skrývanie prázdnych objektov**:
- Hlavice (pologule) objektov, ktorých všetky metódy a členské premenné boli po zapnutí módu odstránené zo scény, budú skryté.
  Opätovne ich môžete zobraziť príkazom menu:
  Scéna → Zobraziť všetky objekty

14 Zapnúť **výsúvanie prázdnych objektov na úroveň rodičovského objektu**:

- Hlavice (pologule) objektov, ktorých všetky metódy a členské premenné boli po zapnutí módu odstránené zo scény, budú vysunuté na hor – na úroveň hlavice objektu, ktorý ich vytvoril.

15 **Priblížiť objekty** zobrazené v scéne bližšie k sebe.
- Presné hodnoty môžete nastaviť cez menu:
  Scéna → Nastavenia… → Rozmiestnenie

16 **Oddialiť objekty** zobrazené v scéne ďalej od seba.
- Presné hodnoty môžete nastaviť cez menu:
  Scéna → Nastavenia… → Rozmiestnenie

17 **Priblížiť vlákna** objektov zobrazené v scéne bližšie k sebe.
- Presnú hodnotu môžete nastaviť cez menu:
  Scéna → Nastavenia… → Rozmiestnenie

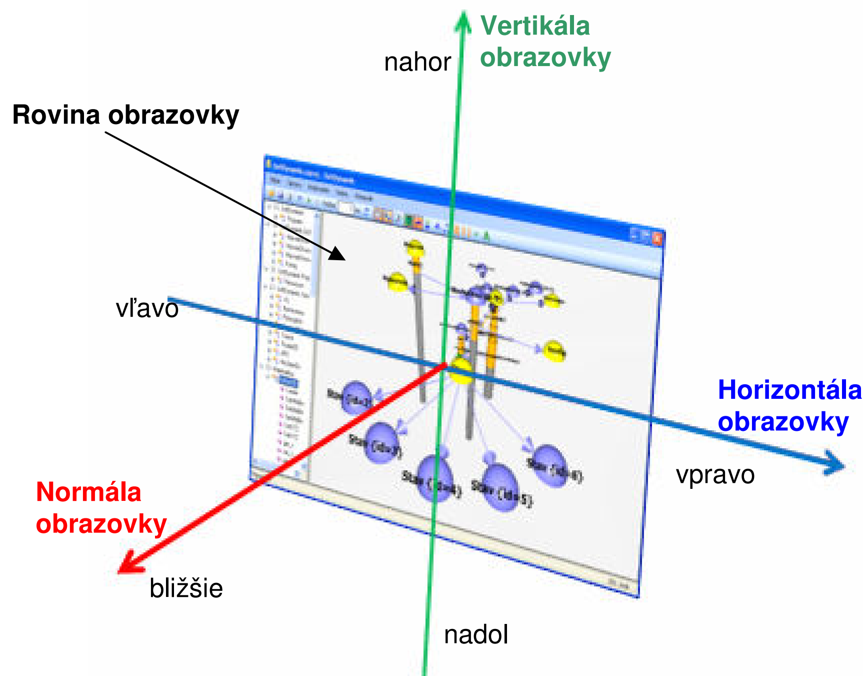
18 **Oddialiť vlákna** objektov zobrazené v scéne ďalej od seba.
- Presnú hodnotu môžete nastaviť cez menu:
  Scéna → Nastavenia… → Rozmiestnenie

19 **Zmenšiť text** popiskov v scéne
- Presnú hodnotu môžete nastaviť cez menu:
  Scéna → Nastavenia… → Vzhľad

20 **Zväčšiť text** popiskov v scéne
- Presnú hodnotu môžete nastaviť cez menu:
  Scéna → Nastavenia… → Vzhľad

Pohyb v scéne:

Vysvetlenie pojmov, ktoré budú použité:



- **Myšou**:
  - Posun v rovine obrazovky:
    Stlačte ľavé tlačítko myši, držte ho stlačené a posúvajte myšou po scéne
  - Posun v smere normály obrazovky (v smere kolmom na rovinu obrazovky):
    - Jemný:

Otáčajte kolieskom myši:
- Dopredu: Pre približovanie scény
- Dozadu: Pre odďaľovanie scény

- Rýchly:
  Stlačte stredné tlačítko (koliesko) myši, držte ho stlačené a posuňte myš:
  - Nahor: Pre približovanie scény
  - Nadol: Pre odďaľovanie scény

  Miera posunu myši vzhľadom k bodu kliknutia je mierou rýchlostí posunu

- Otáčanie:
  - Stredom otáčania je vždy hlavica (pologuľa) najstaršieho (prvého) objektu v scéne.
  - Pre otáčanie sme zvolili techniku "arc – ball", ktorá vychádza z predstavy namapovania scény na povrch pologule (lopty), ktorej najvyššia časť sa nachádza nad stredom scény a obvod leží v rovine obrazovky – na obvode scény. Pri otáčaní v scéne pomyslene otáčate touto loptou.
  - Stlačte ľavé a súčasne pravé tlačítko myši, držte ich stlačené a pohybujte myšou:
    - Po vertikálnej ose obrazovky:
      Otáčanie okolo horizontálnej osi obrazovky.
    - Po horizontálnej ose obrazovky:
      Otáčanie okolo vertikálnej osi obrazovky.
    - Po krajoch scény, respektíve v rohoch scény:
      Otáčanie okolo normály obrazovky.
- Vystredenie zvoleného elementu scény:
  Vami zvolený element scény (hlavicu objektu, metódu, člensků premennú) presuniete do stredu scény:
  - Dvojklikom (dva krát kliknúť ľavé tlačítko myši) na daný element.
  - Pomocou voľby v kontextovom menu daného elementu (vyvoláte kliknutím pravým tlačítkom na daný element)

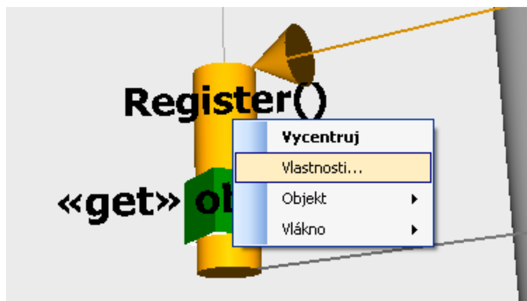
- **Klávesovými skratkami**:
  Aby fungovali klávesové skratky musí byť súčasne aktívne okno scény. Ak nie je kliknite do scény ľavým tlačítkom myši, čím ho aktivujete.

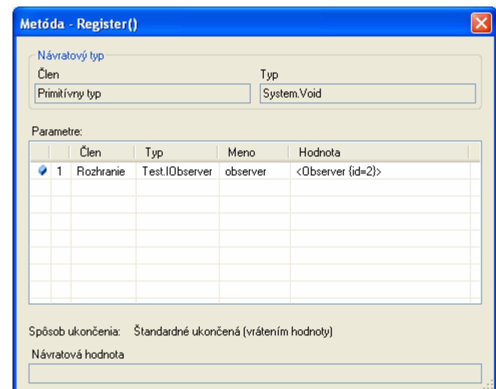
  - Posun v rovine obrazovky:
    Pomocou šípok.
  - Posun v smere normály obrazovky (v smere kolmom na rovinu obrazovky):
    Pomocou kláves „+" a "-"
  - Otáčanie:
    - Stredom otáčania je vždy hlavica (pologuľa) najstaršieho (prvého) objektu v scéne.
    - Shift + šípka vpravo:
      Otáčanie okolo vertikálnej osi obrazovky v kladnom smere (proti smeru hodinových ručičiek)
    - Shift + šípka vľavo:
      Otáčanie okolo vertikálnej osi obrazovky v zápornom smere
    - Shift + šípka hore:
      Otáčanie okolo horizontálnej osi obrazovky v zápornom smere
    - Shift + šípka vpravo:
      Otáčanie okolo horizontálnej osi obrazovky v kladnom smere
    - Shift + "+":
      Otáčanie okolo normály obrazovky v kladnom smere
    - Shift + "-":
      Otáčanie okolo normály obrazovky v zápornom smere

- **Pomocou menu**:
  Pomocou položky *Scéna* v menu je možné prepínať sa medzi pohľadmi:
  - zhora
  - spredu
  - zospodu
  - zošikma

Získanie informácií o vizualizovaných elementoch

Pomocou kontextového menu daného elementu, ktorý vyvoláte kliknutím na daný element pravým tlačítkom myši.





Kopírovanie scény do schránky operačného systému ako bitmapy

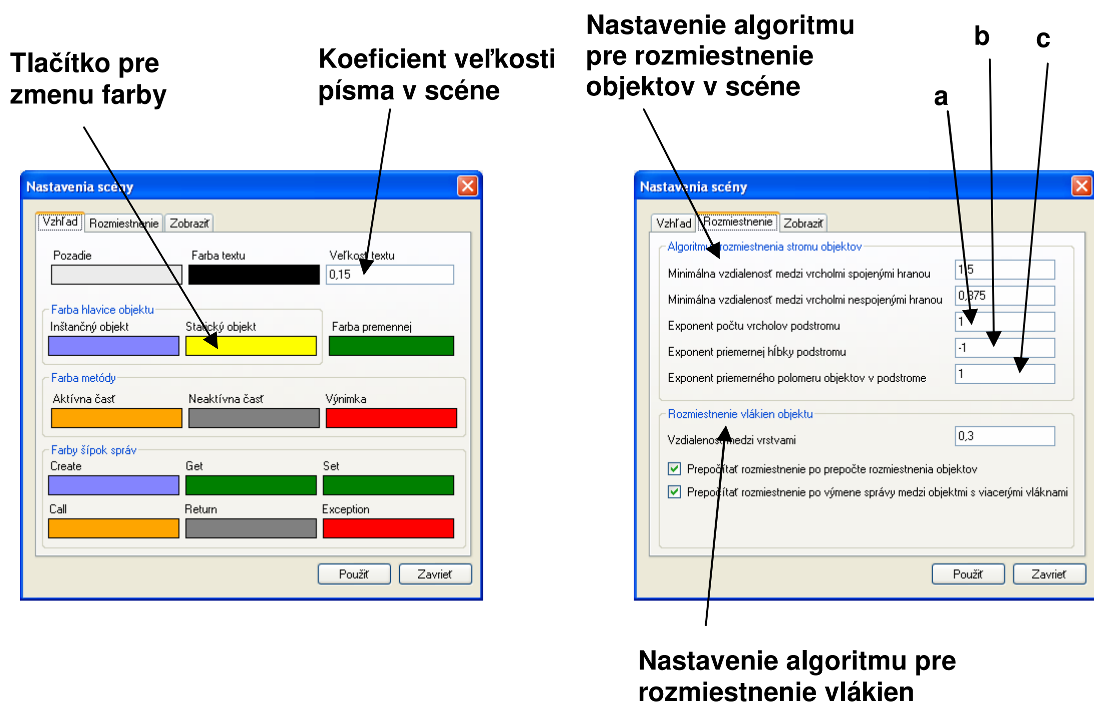
- **Klávesovou skratkou**:
  Ctrl + C
- **Cez menu**:
  Úpravy kopírovať scénu

Scénu je tiež možné uložiť do súboru BMP a vytlačiť.

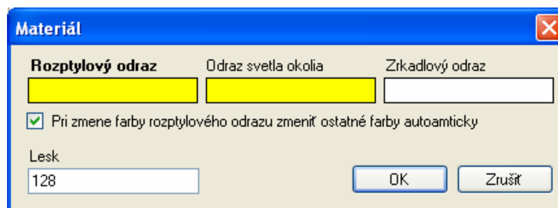
Zmena nastavení scény

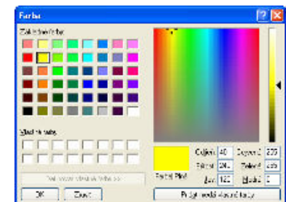
Zmenené nastavenia budú perzistentne uložené, takže pretrvajú aj po ukončení programu

- **Nastavenie vlastných nastavení**:
  Cez dialógové okno, ktoré vyvoláte cez menu:
  Scéna → Nastavenia…



Pre zmenu farby kliknite na "tlačítko pre zmenu farby" a v následne otvorenom dialógovom okne nastavte farby pre jednotlivé druhy svetla a hodnotu lesku:





Súčasťou nastavenia algoritmu počítajúceho rozmiestnenie objektov v scéne je aj nastavenie troch konštantných exponentov a, b, c (viď obrázok) vystupujúcich vo vzorci pre výpočet váhy w podstromu (objekty sú v scéne rozmiestnené v stromovej štruktúre) daného objektu:

$$w = n^a \overline{h}^b \, \overline{r}^c$$

n je počet vrchlov (objektov) v podstrome, $\overline{h}$ je priemerná hĺbka podstromu a $\overline{r}$ je priemerný polomer pologúľ objektov v podstrome.
Váha w je priamo úmerná veľkosti uhlového výseku, ktorý bude pridelený podstromu daného objektu.

- **Obnova pôvodných nastavení**:
  Cez menu:
  Scéna → Obnoviť pôvodné nastavenia

Systémové požiadavky

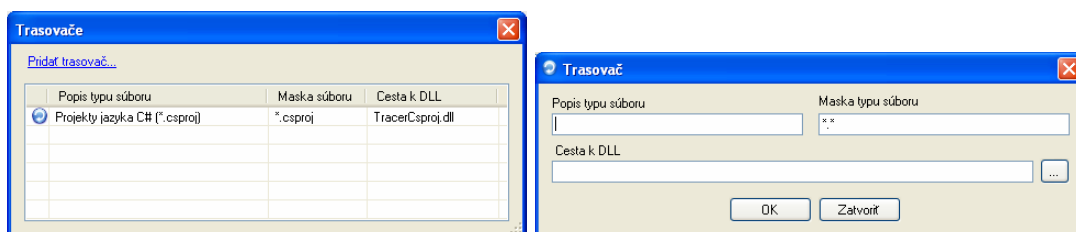
- **Behové prostredie platformy .NET Framework 2.0**

- **Knižnica Windows API opengl32.dll**
  je súčasť inštalácie operačného systému Windows.
- **Myš** majúca okrem ľavého a pravého tlačítka aj koliesko a stredné tlačítko (respektíve koliesko, ktoré je zároveň použiteľné aj ako stredné tlačítko).
- Trasovač dodaný s programom (trasujúci C# spustiteľné programy) vyžaduje inštaláciu .NET rámca PostSharp:
  - Pre trasovanie projektov vo Visual Studio 2005:
    PostSharp 1.5
  - Pre trasovanie projektov vo Visual Studio 2008 a Visual Studio 2010:
    PostSharp 2.x

Pridanie vlastného trasovača

Systém SoftDynamik je možné rozširovať o monitorovanie programov napísaných v iných jazykoch a inými spôsobmi implementáciou vlastného trasovača a jeho zaregistrovaním. To, ktorý registrovaný trasovač sa pri trasovaní má použiť volí používateľ v dialógovom okne ako typ otváraného súboru.

- **Registrácia trasovača**: Cez dialógové okno vyvolateľné cez menu: Krokovanie → Trasovače



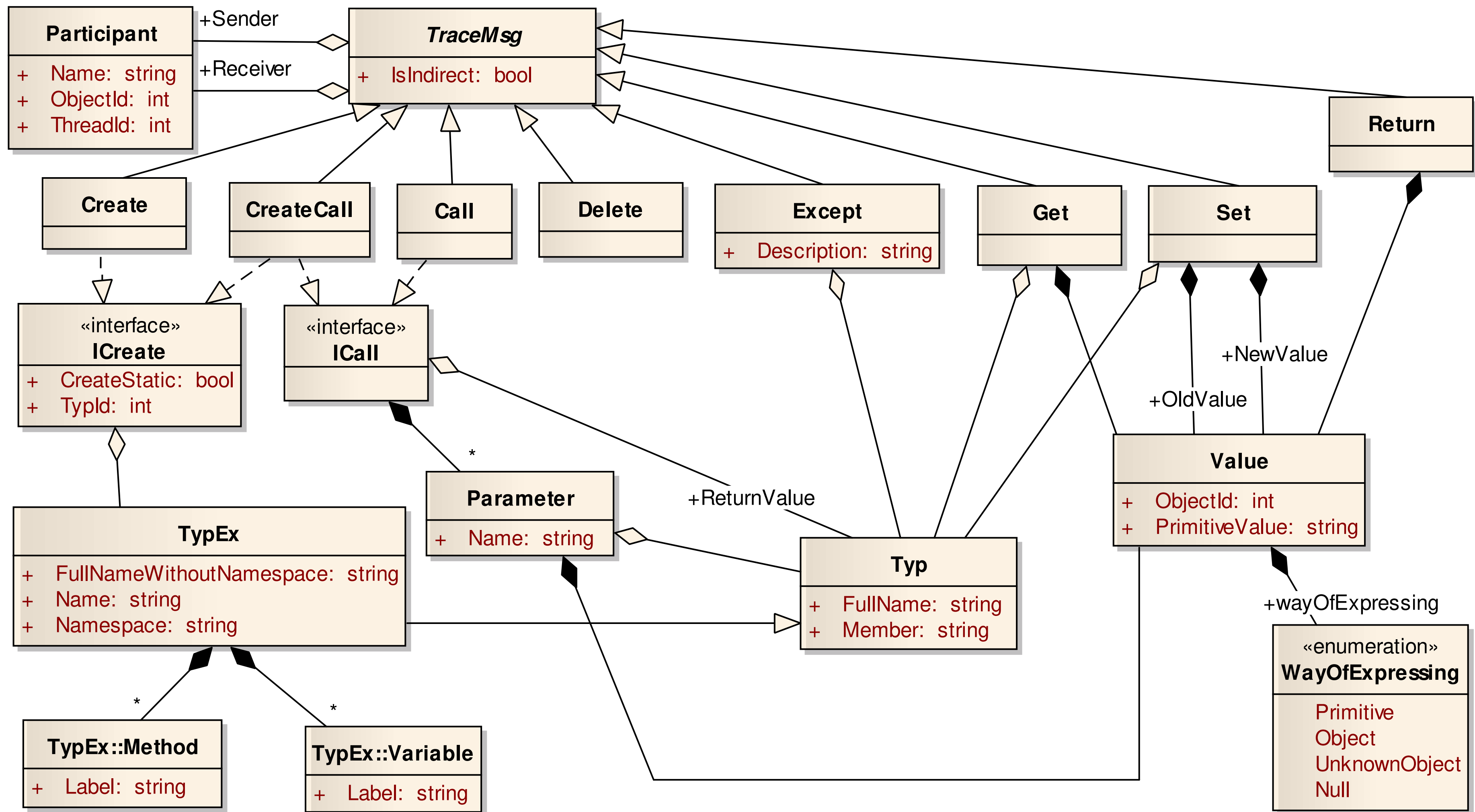
  Pre pridanie nového trasovača kliknite na "Pridať trasovač..." a vyplňte následne otvorené dialógové okno. Kliknutím na položku v zozname trasovačov môžete editovať a odoberať registrované trasovače.

- **Implementácia vlastného trasovača:**
  - Trasovač musí spĺňať podmienky:
    - .NET zostavenie (anglicky assembly), napríklad DLL
    - Obsahuje menný priestor Tracer a v ňom verejnú triedu Tracer, ktorá implementuje rozhranie ITracer definované v DLL SoftDynamik.SI.dll (ktorú treba referencovať), ktorá je súčasťou inštalácie systému SoftDynamik.
  - Šablóna C# zdrojového kódu trasovača s popisom rozhrania ITracer:

```
using System;
using SoftDynamik.SI;//referencovat SoftDynamik.SI.dll
namespace Tracer
{
    //V pripade chyby vyhodit vynimku TracerException
    public class Tracer : ITracer
    {
        //VLASTNOSTI
        public bool Fields
        {
            //sledovat pristup k clenskym premennym objektov?
            get{return false;}
```

```
            set{}
        }
        public bool Synchronous
        {
            //monitorovat synchronne?
            get{return true;}
            set{}
        }
        public bool StepOver
        {
            //je zapnuty preskakovaci mod?
            get{return false;}
            set{}
        }
        public bool StepOverAllow
        {
            //moze byt zapnuty preskakovaci mod?
            //umoznuje to tento trasovac?
            get{return false;}
        }

        //METODY
        public void Start(string file)
        {
            //Zaciatok trasovania
            //file: cesta k suboru ktory ma byt trasovny

            //Priklad vypisu spravy do stavoveho
            //riadku programu SoftDynamik.exe
            this.InfoToStatusBar("Kompilujem...");
        }
        public TraceMsg NextStep()
        {
            //Vratit spravu o dalsom kroku v monitorovanom
                  programe
            //Ak aktualne nie je ziadna sprava k odoslaniu
                  vratit null
            return null;
        }
        public void Reload()
        {
            //Pretocit program na zaciatok (znova spustit a
                  pozastavit)
        }

        public void Close()
        {
           //Koniec monitorovania
        }

        //UDALOSTI
        //Zavolat ak chceme nieco vypisat do stavoveho riadku
                  SoftDynamik
        public event InfoToStatusBar InfoToStatusBar;
    }
}
```

- Definícia trasovačom vracanej trasovacej správy TraceMsg UML diagramom. TraceMsg je abstraktná trieda. Všetky typy sú definované v knižnici SoftDynamik.SI.dll


Participant
+ Name: string
+ ObjectId: int
+ ThreadId: int
+Sender
+Receiver
TraceMsg
+ IsIndirect: bool
Return
Create
CreateCall
Call
Delete
Except
+ Description: string
Get
Set
«interface»
ICreate
+ CreateStatic: bool
+ TypId: int
«interface»
ICall
+NewValue
+OldValue
*
Parameter
+ Name: string
+ReturnValue
Value
+ ObjectId: int
+ PrimitiveValue: string
TypEx
+ FullNameWithoutNamespace: string
+ Name: string
+ Namespace: string
Typ
+ FullName: string
+ Member: string
+wayOfExpressing
«enumeration»
WayOfExpressing
Primitive
Object
UnknownObject
Null
*
*
TypEx::Method
+ Label: string
TypEx::Variable
+ Label: string

Príloha č. 3 Článok publikovaný na SCCG 2013

Nasleduje článok, ktorým sme spoločne s naším vedúcim diplomovej práce publikovali výsledky, ktoré sme dosiahli v rámci riešenia tejto diplomovej práce.

Článok (Grznár F. – Kapec, P.: Visualizing dynamics of object oriented programs with time context.) bol publikovaný na medzinárodnej vedeckej konferencii **Spring Conference on Computer Graphics** 2013 (SCCG). Zborník z tejto konferencie plánuje publikovať tiež **ACM Publishing House.**