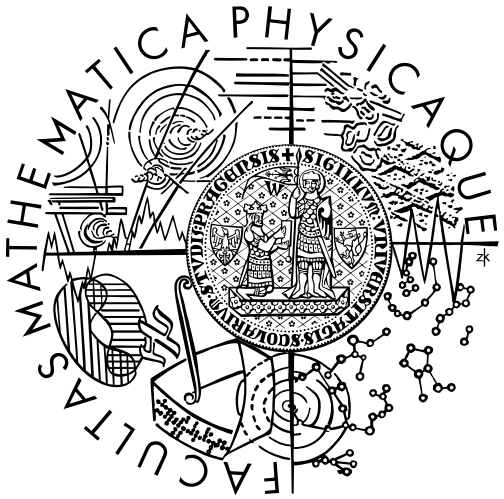
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Štefan Šimon

Generování sekvenčních diagramů UML z kódu

Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc.Ing. Karel Richta, CSc.
Studijní program: Informatika

Ďakujem vedúcemu diplomovej práce doc. Ing. Karlovi Richtovi, CSc. za cennú odbornú pomoc a pripomienky poskytnuté pri písaní tejto práce.

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

V Praze dne 5.8.2008 Štefan Šimon

Obsah

Název práce: Generování sekvenčních diagramů UML z kódu
Autor: Štefan Šimon
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: doc. Ing. Karel Richta, CSc.
e-mail vedoucího: richta@ksi.mff.cuni.cz
Abstrakt: Sekvenčné diagramy sú jedným typom UML diagramov. Špecifikácia UML 2.1 prináša nové vlastnosti sekvenčných diagramov umožňujúce znázorniť správanie systému na úrovni programového kódu. S príchodom týchto vlastností vzniká priestor pre vytvorenie nástroja umožňujúceho reverzné inžinierstvo sekvenčných diagramov z programového kódu.

Cieľom tejto práce je návrh a implementácia nástroja umožňujúca reverzné inžinierstvo UML 2.1 sekvenčných diagramov z programového kódu. Použitý prístup umožňuje reverzné inžinierstvo sekvenčných diagramov pre celú aplikáciu, ako aj pre jednotlivé metódy tried aplikácie. Pri generovaní diagramu aplikácie je použitá run-time dynamická analýza. Statická analýza grafu riadenia toku metódy sa využíva pri generovaní diagramu metódy. Súčasťou práce je i editor sekvenčných diagramov vytvorený pomocou konceptu doménovo-špecifických jazykov. Editor umožňuje export do štandardizovaného formátu XMI.
Klíčová slova: sekvenčný diagram, statická analýza, dynamická analýza, doménové jazyky

Title: Generating UML Sequence Diagrams from the Code
Author: Štefan Šimon
Department: Department of Software Engineering
Supervisor: Doc. Ing. Karel Richta, CSc.
Supervisor's e-mail address: richta@fel.cvut.cz
Abstract: Sequence diagrams are a subset of UML diagrams. UML 2.1 brings new features to sequence diagrams allowing them to model system behavior on the program code level. These features carry a possibility for creating tools allowing reverse engineering of sequence diagrams from program code.

The goal of this thesis is to analyze and implement a tool for reverse engineering of UML 2.1 sequence diagrams from program code. The presented approach allows reverse engineering of a whole application, as well as of individual application class methods. Run-time dynamic analysis is used for generating application diagrams. Static control-flow analysis is used for generating method diagrams. This work includes a diagram editor, created using domain-specific languages technology, which also allows diagram export to standardized XMI format.
Keywords: sequence diagram, static analysis, dynamic analysis, domain-specific languages

1 Úvod

1.1 Motivácia

Pri vývoji softwaru sa pomerne často stáva, že návrh a dokumentácia programu sa rozchádzajú. Udržovať aktuálnu dokumentáciu je pritom pracné a nákladné. Jej nedostatok spôsobuje problémy pri pochopení funkčnosti systému. Pri rozsiahlych systémoch, ktoré sa vyvíjajú a udržujú niekoľko rokov i dlhšie, je kvalitná dokumentácia navyše značne potrebná. Po niekoľkých rokoch sa údržba takýchto systémov môže stať extrémne náročná, zvlášť po prípadnom odchode pôvodných programátorov a analytikov.

Potreba aktuálnej dokumentácie je dôležitá aj pri iteratívnom vývoji, ktorý je v súčasnosti čoraz viac rozšírenou formou vývoja systémov. Pri tomto spôsobe vývoja aplikácií je vhodné aktualizovať dokumentáciu po každej iterácii aplikácie a porovnať ju s dokumentáciou predošlej iterácie pre zaistenie kontroly, či vývoj je založený na obsahu poslednej iterácie [1].

V súčasnosti je štandardom pre dokumentáciu programov modelovací jazyk UML[1] (Unified Modelling Language) vo svojej aktuálnej verzii 2. Jazyk UML sa používa pre špecifikáciu, dokumentáciu, vizualizáciu a návrh softvérových systémov. Obsahuje niekoľko typov diagramov, ktoré umožňujú zachytiť statickú štruktúru i dynamické chovanie programu. Príkladom UML diagramu je diagram tried (class diagram). Tento diagram zachycuje statickú štruktúru aplikácie, ako napríklad typy, triedy, ich metódy a členy a vzťahy medzi nimi.

S rozšírením jazyka UML vznikli nástroje uľahčujúce integráciu vývoja a modelovania – tzv. round-trip engineering, ktorý zahŕňa tzv. dopredné (forward) a spätné (reverse) inžinierstvo. Dopredné inžinierstvo umožňuje generovanie programových artefaktov z UML návrhu, napríklad možnosť generovať signatúry tried priamo z modelu diagramov tried. Reverzné inžinierstvo umožňuje zasa opačný postup ako dopredné – z programového kódu sú spätne generované diagramy UML. Techniky reverzného inžinierstva sa často využívajú pre pochopenie správania a štruktúry softvérových systémov, ktorým chýba dokumentácia alebo už je zastaraná. Pomocou reverzného inžinierstva je možné napríklad zo zdrojového kódu priamo generovať diagramy tried. V súčasnosti existuje veľké množstvo

[1] UML – www.uml.org

nástrojov podporujúcich dopredné i spätné inžinierstvo týchto diagramov. Podpora v prípade iných UML diagramov je však výrazne menšia.

Do skupiny často používaných dynamických UML diagramov patrí sekvenčný diagram. Diagram zachycuje interakciu medzi objektmi aplikácie, ktorá zaisťuje potrebnú funkčnosť. Je preto vhodný pre znázornenie chovania časti systému. Aktuálna verzia UML 2 obsahuje nové rozšírenia pre dôkladnejšie zachytenie tohto chovania. Pridáva *fragmenty (rámce)*. Príklad sekvenčného diagramu s fragmentmi znázorňuje Obr. 4.2.

Nové rozšírenia jazyka UML umožňujú pomocou sekvenčného diagramu namodelovať chovanie programu alebo jeho časti na úrovni programového kódu. V prípade integrácie s ostatnými UML diagramami, predovšetkým s diagramami tried, je takto možné namodelovať celú aplikáciu alebo jej časť. Triedy, typy a rozhrania aplikácie sa namodelujú s využitím diagramu tried, implementácie metód zasa pomocou sekvenčného diagramu. Z tohto modelu sa s využitím dopredného inžinierstva vygeneruje príslušná kostra implementácie. Reálne použitie by takáto funkčnosť mohla mať napríklad u komplexných a algoritmicky zložitých častí aplikácie. U reverzného inžinierstva je možný opačný postup – vygenerovanie sekvenčného diagramu z programového kódu. Takéto generovanie umožňuje spätne zachytiť a zdokumentovať chovanie systému. Proces je možné využiť aj pre automatické testovanie systému [2]. Táto práca prezentuje prístup pre reverzné inžinierstvo sekvenčných diagramov. Implementácia dokáže generovať diagramy na úrovni jednotlivých metód alebo celej aplikácie.

Nástroje pre reverzné a dopredné inžinierstvo sú rozšírené predovšetkým na platforme JAVA. V prípade platformy .NET je počet týchto nástrojov rádovo nižší. Dôvodom je predovšetkým to, že platforma .NET je mladšia a menej rozšírená než platforma JAVA. Naša práca využíva platformu .NET ako zdrojovú platformu a prezentuje pravdepodobne prvú implementáciu reverzného inžinierstva UML 2 sekvenčných diagramov na tejto platforme.

1.2 Ciele

Účelom tejto práce je preskúmať, popísať a implementovať reverzné inžinierstvo UML 2 sekvenčných diagramov na platforme .NET. Súčasťou práce je i implementácia nástroja pre editáciu vygenerovaných diagramov. Vygenerované diagramy je ďalej možné exportovať do štandardizovaného formátu XMI. Použitý prístup umožňuje generovať diagramy pre celú aplikáciu, ako aj pre jednotlivé metódy tried aplikácie.

Pri generovaní aplikačného diagramu je využitá dynamická analýza programu. Implementácia práce sa pripojí na bežiacu inštanciu zdrojovej aplikácie a odchytáva

jednotlivé volania metód medzi objektmi. Z týchto volaní je následne generovaný sekvenčný diagram. Táto analýza zachycuje jeden tok aplikácie – tok aktuálne bežiacej inštancie. Generovanie všetkých možných tokov by v prípade celej aplikácie viedlo k veľkému a neprehľadnému diagramu i v prípade použitia interakčného fragmentu (viď kapitola 2.1.1).

Pri generovaní diagramu metódy je použitá statická analýza skompilovaného tvaru metódy – medzikódu. Na rozdiel od aplikačného diagramu prebieha analýza všetkých možných tokov metódy. Pri použití tohto prístupu nie je nutné spúšťať zdrojovú aplikáciu. Náš nástroj prechádza graf riadenia toku (control-flow graph) medzikódu zdrojovej metódy a transformuje jednotlivé volania, vetvy a skoky v medzikóde do príslušných elementov sekvenčného diagramu. Graf riadenia toku je reprezentácia toku metódy, kde vrcholy predstavujú bloky volaní a hrany možné toky medzi vrcholmi. Po analýze toku metódy a vygenerovaní elementov sekvenčného diagramu nasleduje transformačný proces, ktorý transformuje vygenerované diagramy do prehľadnejšej formy a eliminuje redundantné informácie vzniknuté počas analýzy.

Z výstupov analýzy metódy, resp. aplikácie sú následne generované prezentačné elementy sekvenčného diagramu, ktoré tvoria vstup pre editor diagramov. Editor je implementovaný ako zásuvný modul do vývojového prostredia Visual Studio 2005[1]. Takáto implementácia umožňuje tesnú integráciu zdrojového kódu a generovaných diagramov.

Editor je implementovaný s využitím konceptu doménových jazykov (Domain Specific Languages - DSL)[2]. Doménové jazyky predstavujú nový spôsob riešenia skupiny problémov, ktoré je možné znázorniť pomocou jedného ad hoc jazyka. Pre sekvenčné diagramy sme vytvorili takýto doménový jazyk. Z tohto jazyka je potom jednoduché generovať ďalšie výstupné artefakty. Jedným z generovaných artefaktov je štandardizovaný XMI model sekvenčného diagramu, ktorý slúži pre úpravu diagramu v nástrojoch tretích strán.

1.3 Štruktúra práce

Kapitola 2 popisuje použité nástroje. Predstavuje jazyk UML. Hlavnú časť kapitoly tvorí rozbor sekvenčných diagramov. Nasleduje popis doménového vývoja - technológie použitej pri implementácii editora diagramov.

[1] Visual Studio 2005 - http://msdn.microsoft.com/en-us/vs2005/default.aspx
[2] Domain Specific Languages - http://en.wikipedia.org/wiki/Domain-specific_programming_language

Kapitola 3 zrovnáva existujúce prístupy pre reverzné inžinierstvo sekvenčných diagramov. V závere popisuje navrhované riešenie.

Kapitola 4 popisuje algoritmy použité pre reverzné inžinierstvo. Tieto algoritmy analyzujú štruktúru metódy a na jej základe produkujú dáta pre sekvenčný diagram metódy. Kapitola ďalej prináša rozšírenia umožňujúce precíznejšie zachytenie toku metódy.

Kapitola 5 sa venuje návrhu editora sekvenčných diagramov a jeho grafickej reprezentácie vytváranej z dát získaných z kapitoly 4.

Kapitola 6 uvádza niektoré zaujímavé postupy implementácie, ako aj diskutuje možné implementácie z rôznych hľadísk.

Kapitola 7 (záverečná) stručne hodnotí navrhnuté riešenie a pridáva možné odporučenia ďalšieho smeru práce.

2 Použité nástroje

Táto kapitola popisuje nástroje a technológie použité v tejto práci. Predstavuje jazyk UML, predovšetkým rozoberá štruktúru sekvenčných diagramov. Popisuje koncept doménového vývoja použitého pri implementácii editora sekvenčných diagramov.

2.1 UML

Špecifikácia OMG [3] definuje jazyk UML (Unified Modeling Language) ako *"grafický jazyk pre vizualizáciu, špecifikáciu, konštrukciu a dokumentáciu artefaktov softvérových systémov. UML poskytuje štandardizovanú cestu zápisu návrhu systému vrátane koncepčných nástrojov, ako biznis procesov alebo funkcionalít systému, ako aj konkrétnych nástrojov, akými sú bloky v programovacích jazykoch, databázové schémy a znovu použiteľné softvérové komponenty."* UML však nepredstavuje žiadnu metodológiu alebo procedúru pre vývoj systémov. Je to iba jazyk, v ktorom je znázornený návrh systému. UML môže byť použité niekoľkými spôsobmi k podpore vývojovej metodológie (ako napr. UP[1] či SCRUM[2]), ale samotné takúto metodológiu nešpecifikuje.

UML obsahuje niekoľko základných typov diagramov, ktoré sa delia do troch základných skupín

- Funkcionálne diagramy – znázorňujú požiadavky na funkčnosť systému z pohľadu užívateľa. Príkladom je diagram prípadov užitia (use case diagram).
- Statické diagramy – zobrazujú statickú štruktúru systému s využitím objektov, atribútov, operácií a vzťahov. Zahŕňajú napríklad diagram tried.
- Dynamické diagramy – prezentujú dynamické správanie systému tak, že ukazujú spoluprácu medzi objektmi a zmeny vo vnútornej štruktúre objektov. Príkladom je sekvenčný diagram, diagram aktivít.

Detailný popis všetkých UML diagramov je možné nájsť v špecifikácii UML [4]. Táto práca vychádza z aktuálnej verzie UML 2.1.

2.1.1 Sekvenčný diagram

Sekvenčný diagram (SD) patrí k jedným z najpoužívanejších UML diagramov. Primárne sa používa pre znázornenie interakcií medzi objektmi v poradí, v ktorom tieto interakcie vznikli. Diagramy sú vhodné pre zobrazenie vzájomnej komunikácie medzi

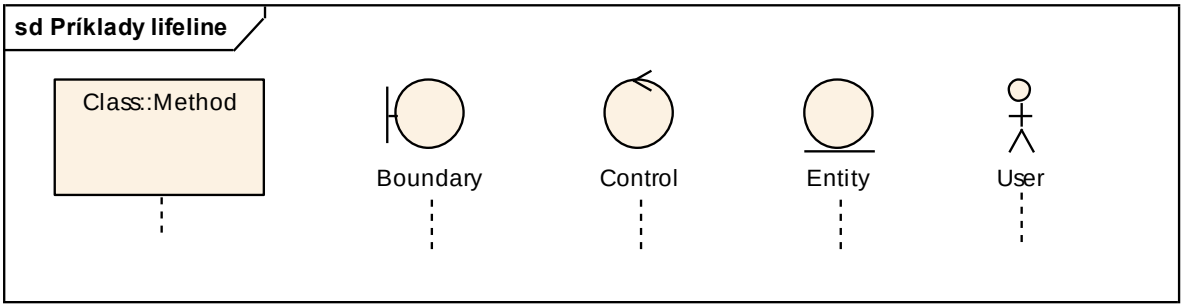
[1] Unified Process – iteratívna metodika soft. vývoja (http://en.wikipedia.org/wiki/Unified_Process)
[2] SCRUM – agílna metodika softvérového vývoja (http://www.controlchaos.com/)

objektmi a správami, ktoré túto komunikáciu spôsobujú. Táto komunikácia je znázornená pomocou vertikálnej a horizontálnej osy. Vertikálna os znázorňuje zdola nahor časovú postupnosť volaní - správ. Horizontálna os znázorňuje zľava doprava inštancie objektov, medzi ktorými sú správy posielané. Príklad sekvenčného diagramu znázorňuje Obr. 4.2.

Sekvenčný diagram sa skladá z niekoľkých typov elementov – napr. lifeline, správa atď. Detailný popis všetkých artefaktov je možné nájsť v špecifikácii UML Superstructure [4] v kapitole Interactions.

Lifeline

Lifeline (čiara života) predstavuje interakčnú entitu. Znázorňuje sa prerušovanou čiarou. Väčšinou zastupuje interakčný objekt. V tomto prípade obsahuje hlavička lifeline obdĺžnik s menom objektu. Ďalej môže lifeline znázorňovať osobu alebo inú externú entitu komunikujúcu so systémom – hlavička obsahuje symbol Actor. Pre znázornenie vzoru Model-View-Controller[1] sa používajú symboly Entity (Model), Boundary (View) a Control (Controller). Príklady lifeline znázorňuje Obr. 2.1.

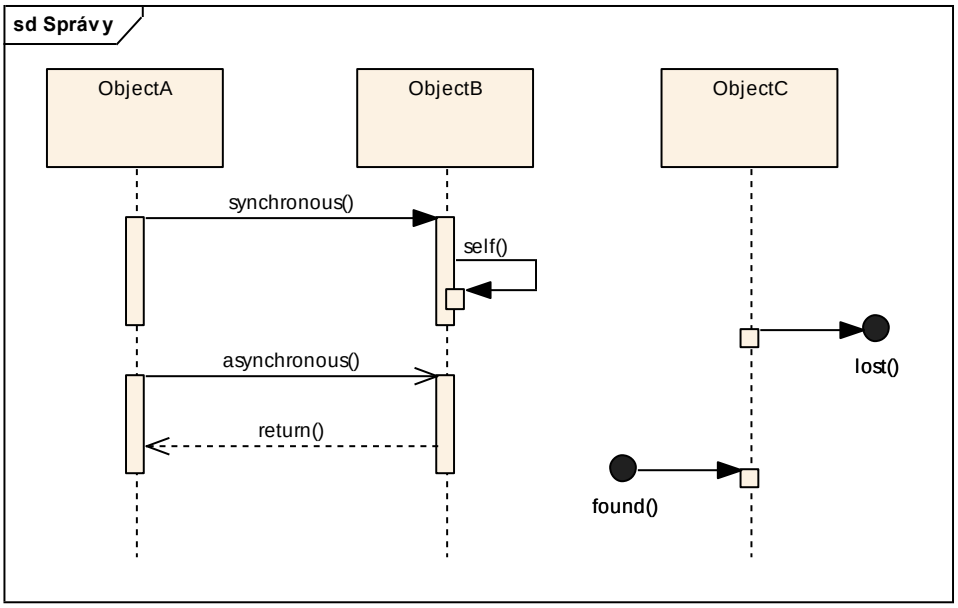


Obr. 2.1 - Príklady lifeline

Správa

Správa predstavuje interakciu medzi elementmi diagramu. Znázorňuje sa horizontálnou šípkou, nad ktorou je umiestnený jej názov. Správy môžu byť úplné, nájdené alebo stratené; synchrónne alebo asynchrónne. Na diagrame na Obr. 2.2 je správa synchronous() synchrónna (znázorňuje sa šípkou s tučným hrotom), správa asynchronous() asynchrónna (šípka s tenkým hrotom) a správa return() je návratová (prerušovaná šípka s tenkým hrotom).

[1] Model-view-controller – architektonický vzor pre oddelenie vrstiev systému



Obr. 2.2 - Príklady správ

Stratené a nájdené správy

Stratené správy sú odoslané správy bez príjemcu alebo končiace u príjemcu znázornenom na inom diagrame. Nájdené správy prichádzajú od neznámeho odosielateľa alebo od odosielateľa z iného diagramu. Stratené a nájdené správy sú znázornené ako správy vychádzajúce z/do bodu. Na Obr. 2.2 je správa lost() stratená a správa found() nájdená.

Vlastná správa

Vlastná správa (self-message) je druh správy, ktorá reprezentuje rekurzívne volanie alebo volanie rovnakého objektu. Vychádza a vchádza z tej istej lifeline. Na Obr. 2.2 je správa self() vlastná.

Aktivácia

Aktivácia sa znázorňuje obdĺžnikom umiestneným na lifeline. Zo sémantického hľadiska predstavuje vznik exekučného procesu. Z tohto procesu môžu vychádzať ďalšie správy. Na Obr. 2.2 vychádza správa synchronous z exekučného procesu na lifeline ObjectA a vyvoláva nový exekučný proces na lifeline ObjectB.

Fragmenty

Fragmenty (rámce) sú novým prvkom UML 2.0. Pomocou fragmentov je možné pridať do SD model procedurálnej logiky programového kódu. Fragmenty zachycujú možné exekučné toky v diagrame. S ich pomocou je možné zachytiť podmienený, cyklický, alternatívny alebo prerušujúci scenár toku. Fragmenty môžu byť vnárané. Každý fragment môže obsahovať *strážiacu podmienku* - *strážcu* (guard). Strážca je operand, ktorý musí byť

pri vstupe do fragmentu vyhodnotený ako pravdivý pre vykonanie interakcií (vnorených fragmentov a správ) vo fragmente. Každý fragment je definovaný jeho interakčným operátorom. Tento operátor ďalej určuje sémantiku fragmentu. Interakčný operátor fragmentu je znázornený v jeho ľavom hornom rohu. Pre zjednodušenie budeme ďalej v tomto texte fragment, ktorého sémantiku určuje interakčný operátor *foo* nazývať *fragmentom typu foo* alebo *"foo" fragment*.

Norma UML 2 definuje dva druhy fragmentov – kombinované a interaktívne. Interaktívny fragment predstavuje referenciu na iný diagram. Jeho interakčný operátor je *ref*. Pre spojenie diagramu s interakčným fragmentom sa používa *brána*. Medzi kombinované fragmenty patria fragmenty typu opt, alt, break a loop.

Fragment typu *opt* znázorňuje voliteľný tok. Interakcie v „opt" fragmente sú vykonané, iba ak strážca fragmentu je splnený. Pomocou tohto fragmentu je možné napríklad znázorniť programový konštrukt `if` bez `else` vetvy.

Fragment typu *alt* znázorňuje dve alebo viac možných alternatív toku. Jednotlivé alternatívy sú oddelené prerušovanou čiarou. Každá alternatíva obsahuje implicitného alebo explicitného strážcu. Pre vykonanie toku v rámci alternatívy musí byť splnený strážca alternatívy. Ak alternatíva neobsahuje strážcu, použije sa implicitný *true* strážca. Pomocou fragmentu typu *alt* je možné namodelovať programový `switch` konštrukt alebo `if (cond){ … } else { … }` konštrukt. V prípade `if (cond){ … } else { … }` konštruktu obsahuje fragment dve alternatívy. Prvá je strážená podmienkou *cond*. Druhá alternatíva je strážená *else* operandom. Každý alt fragment môže mať maximálne jeden else operand. Tento operand predstavuje podmienku, ktorá je negáciou disjunkcií všetkých ostatných alternatív fragmentu. Príklad takéhoto fragmentu znázorňuje Obr. 4.2. V tomto prípade fragment typu alt obsahuje dve alternatívy. Prvá alternatíva je strážená podmienkou `(reader.IsEmptyElement)`, druhá alternatíva obsahuje else operand, t.j. je strážená podmienkou `(!reader.IsEmptyElement)`. Fragment typu alt, ktorý po odstránení alternatív bez interakcií obsahuje iba jednu alternatívu je sémanticky ekvivalentný fragmentu typu opt.

Fragment typu *loop* slúži na zachytenie opakujúcich sa interakcií. Interakcie vo fragmente sa opakujú, dokiaľ platí strážca fragmentu. Na príklade na Obr. 4.2

sa opakujú všetky interakcie vo fragmente typu loop, kým je pravdivá strážiaca podmienka `!reader.EOF`.

Pomocou fragmentu typu *break* je možné znázorniť prerušené správanie. Sémantika fragmentu je podobná sémantike kľúčového slova break prítomného v programovacích jazykoch. Po vykonaní interakcií vo fragmente typu break sú zvyšné interakcie v okolitom fragmente preskočené. Ak diagram modeluje programový kód, fragment predstavuje skok do začiatku najbližšieho okolitého fragmentu. Vstup do fragmentu je podmienený splnením strážcu. Príklad fragmentu znázorňuje Obr. 4.2 .

Fragment typu *par* umožňuje zobraziť paralelné chovanie systému. V modeli programového kódu je možné pomocou tohto fragmentu znázorniť súčasný beh viacerých vlákien. Jednotlivé behy sú oddelené prerušovanou čiarou.

Fragment typu *critical* slúži pre znázornenie kritickej sekcie. V modeli programového kódu zachycuje atomickú sekciu. Napríklad v jazyku C# je takáto sekcia uzavretá do `lock { … }` bloku.

Norma UML 2 špecifikuje niekoľko ďalších druhov fragmentov. Ich detailné použitie a očakávané správanie je možné nájsť v norme UML 2 v časti Superstructure [4].

2.1.2 XMI

XML Metadata Interchange (XMI)[1] je otvorený a štandardizovaný formát určený k prenosu modelov medzi rôznymi nástrojmi. Je založený na formáte XML a definovaný štandardizačnou skupinou OMG. S pomocou XMI je možné exportovať a importovať UML modely a diagramy medzi rôznymi nástrojmi, ktoré tento formát podporujú. Aktuálna verzia XMI je 2.1. Táto verzia plne podporuje UML 2.0 i UML 2.1. Formát tejto verzie je značne odlišný od predchádzajúcich verzií. Prináša so sebou nové tagy a vlastnosti umožňujúce zapísať nové vlastnosti UML 2.

XMI neobsahuje žiadne elementy pre uchovanie dát popisujúcich rozloženie UML elementov v rámci diagramu. V prípade sekvenčných diagramov tak výsledné XMI obsahuje len dáta o jednotlivých elementoch diagramu. Dáta popisujúce rozloženie elementov sa obyčajne ukladajú v rozširujúcom elemente <XMI:extension>. Obsah tohto elementu nie je štandardizovaný špecifikáciou XMI a závisí na konkrétnej implementácii výrobcu.

[1] XMI - http://www.omg.org/technology/documents/formal/xmi.htm

2.2 Doménový vývoj

Doménový (Domain-Specific) vývoj je nové odvetvie softvérového vývoja založené na pozorovaní, že veľa problémov je možné jednoducho riešiť vytvorením ad hoc jazyka [5]. Takýto jazyk nazývame doménový jazyk (Domain Specific Language – DSL). Príkladom je textový jazyk regulárnych výrazov. Na platforme .NET s pomocou inštancie triedy `System.Text.RegularExpression.Regex` a regulárneho výrazu `(?<user>[^@]+)@(?<host>.+)` je takto možné nájsť v texte všetky emailové adresy, ďalej hodnotu pred znakom `@` priradiť do premennej `user`, hodnotu za znakom `@` do premennej host. Bez použitia regulárnych výrazov by vývojár musel sám napísať parsovací algoritmus a následné priradenie do premenných. Takýto postup je prácnejší a náročnejší než pri použití regulárnych výrazov.

Doménové jazyky (ďalej DSL) môžu byť textové alebo grafické. Grafické jazyky umožňujú priamočiaru a zrozumiteľnú vizualizáciu problému pomocou diagramu. Príkladom grafického jazyka je UML diagram tried. Na platforme .NET je možné grafické DSL jazyky vytvárať s pomocou API DSL Tools a následne ich aplikovať na široké spektrum problémov. Editor diagramov tried, ktorý je súčasťou Visual Studia 2005 bol taktiež implementovaný s pomocou DSL Tools.

3 Existujúce prístupy

V súčasnosti existuje niekoľko prác zaoberajúcich sa reverzným inžinierstvom sekvenčných diagramov. Nasledujúca kapitola popisuje niektoré z nich.

3.1 Run-time dynamická analýza

Jedným z možných spôsobov generovania sekvenčného diagramu je využitie dynamickej run-time analýzy programu. Diagram sa generuje z vytrasovanej postupnosti volaní vzniknutej počas behu programu. Takúto postupnosť je možné získať s využitím externého nástroja, napr. *mono --trace*[1] alebo implementáciou vlastného nástroja, tzv. *profilera*. Zo získanej postupnosti volaní je potom triviálne vygenerovať základný sekvenčný diagram bez fragmentov. Vygenerovanie komplexnejších UML 2.0 sekvenčných diagramov s využitím dynamickej analýzy predstavuje náročnejšiu úlohu.

Prístup popísaný v práci [6] rozširuje základný sekvenčný diagram o stav programu pred a po volaním každej správy – o tzv. stavový vektor. Potom na základe generovania niekoľkých inštancií sekvenčných diagramov s rôznymi vstupnými dátami kombinuje tieto diagramy do jedného. S využitím informácii získaných zo stavového vektora pridáva do diagramu fragmenty typu loop a alt.

Briand et al. [7] používa run-time trasovanie kombinované so statickou analýzou. Pre inštrumentovaný systém najprv vytvorí diagram tried. Na základe tohto diagramu potom doplní zdrojový kód o dodatočné informácie pomocou aspektového programovania. Na základe týchto informácií generuje výsledný diagram.

Použitie run-time analýzy má viaceré nedostatky. Vygenerovaný sekvenčný diagram nevytvára komplexný obraz programu - popisuje iba jeden jeho stav, ktorý vznikol na základe určitej podmnožiny vstupných hodnôt. Táto množina vstupných hodnôt dokonca nemusí byť dopredu vôbec známa. Takýto diagram má malú vypovedaciu hodnotu o funkčnosti programu a navyše môže byť pre vývojára mätúci. V prípade, ak nie je k dispozícii spustiteľná verzia programu ale je dostupná iba jeho časť (napr. vo forme knižnice), vygenerovať takéhoto diagramu môže byť značne zložité.

3.2 Statická analýza

Popri dynamickej analýze sa pri reverznom inžinierstve sekvenčných diagramov používa statická analýza. S použitím statickej analýzy je možné zachytiť kompletnú štruktúru

[1] Projekt Mono - *http://www.mono-project.com/Debugging*

a tok programu. Jednou z hlavných nevýhod použitia statickej analýzy je, kvôli prítomnosti dedičnosti, polymorfizmu a dynamickej väzby v programovacích jazykoch, zachytiť iba zo zdrojového kódu dynamický typ objektu a z toho vyplývajúce postupnosti volaní.

Rountev et al. [8] popisuje algoritmus, ktorý analyzuje graf riadenia toku metódy a mapuje ho na UML 2.0 elementy. Ďalej prináša sériu transformácii zvyšujúcich čitateľnosť a zrozumiteľnosť vygenerovaného diagramu. Uvedený prístup umožňuje generovať diagramy pre jednotlivé metódy tried programu. Z tohto článku vychádza prístup použitý v tejto práci týkajúci sa generovania sekvenčného diagramu metódy.

Z existujúcich komerčných nástrojov používa statickú analýzu napr. nástroj Borland Together[1] alebo nástroj IBM Rational[2]. Oba spomenuté nástroje sú dostupné iba pre platformu JAVA.

[1] Borland Together – www.borland.com/together

[2] IBM Rational – www.rational.com

3.3 Použitý prístup

Náš prístup umožňuje reverzne generovať diagramy pre celú aplikáciu a pre jednotlivé metódy tried.

3.3.1 Diagram aplikácie

Pre vygenerovanie diagramu aplikácie sa využíva jednoduchý prístup založený na run-time dynamickej analýze popísaný v kapitole 3.1. Diagram je generovaný pre jednu inštanciu aplikácie. Pod pojmom aplikácia v tomto texte rozumieme spustiteľný súbor platformy .NET (konzolová aplikácia, Windows aplikácia, atď.)

Generovací proces spočíva v nasledujúcich krokoch. K aplikácii sa po jej spustení pripojí nástroj - profiler, ktorý zachytáva jednotlivé volania metód medzi objektmi vzniknuté počas jej behu. Výstup nástroja je ďalej vstupom pre algoritmus, ktorý zo zachytených volaní vygeneruje sekvenčný diagram. Nakoľko dáta pre algoritmus sú získavané počas behu aplikácie, pre generovanie diagramu nie je potrebná prítomnosť zdrojových kódov.

Účelom generovania sekvenčného diagramu aplikácie je poskytnúť približný prehľad jej funkčnosti. Kvôli rozsahu diagramu nie je vhodné použitie tejto možnosti pre rozsiahlejšie aplikácie. Príklad sekvenčného diagramu aplikácie vygenerovaného implementáciou tejto práce z aplikácie na Obr. C.1 v prílohe, znázorňuje Obr. C.2 v prílohe.

3.3.2 Diagram metódy

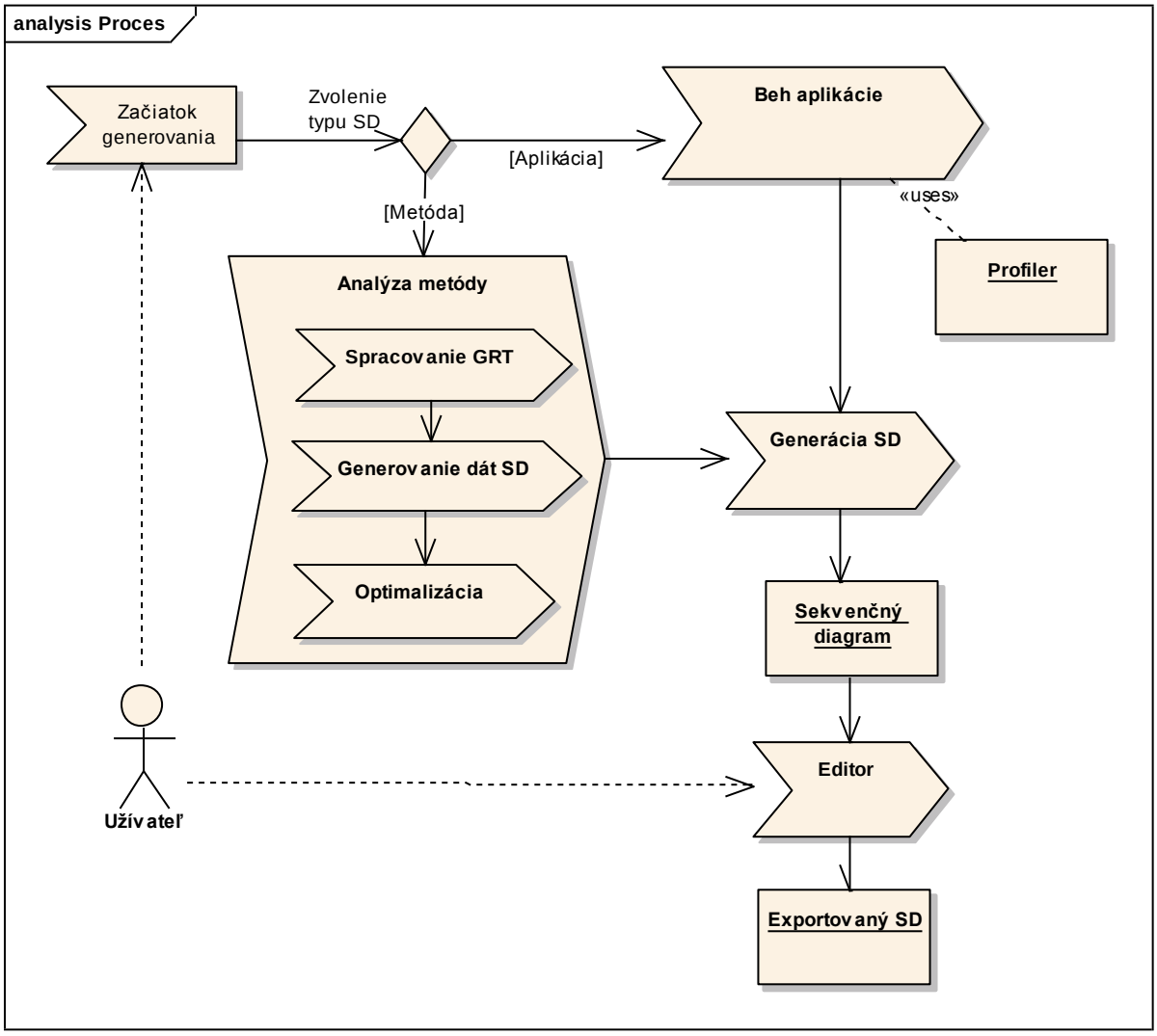
Pre konštrukciu sekvenčného diagramu na úrovni metód jednotlivých tried je použitý komplexnejší prístup s využitím statickej analýzy kódu. Podrobný popis algoritmu, ktorý vykonáva statickú analýzu, obsahuje kapitola 4.

3.3.3 Celkový proces

Prehľad celkového procesu implementácie generovania a editácie sekvenčných diagramov popísaný v tejto práci znázorňuje Obr. 3.1.

Užívateľ môže začať s vytvorením nového diagramu alebo jeho vygenerovaním z diagramu metódy či aplikácie. V prípade generovania SD metódy spustí implementácia proces „Analýza metódy“, ktorý v sérii krokov spracuje graf riadenia toku metódy, vygeneruje dáta sekvenčného diagramu a nakoniec tieto dáta zoptimalizuje. V prípade generovania diagramu aplikácie ju implementácia spustí, pričom sa na ňu pripojí modul „Profiler“ zachytávajúci jednotlivé volania v aplikácii. Výstup z procesu analýzy metódy či behu aplikácie tvorí vstup pre proces „Generácia SD“, ktorý transformuje získané dáta do

prezentačnej formy sekvenčného diagramu. Výstupom procesu je výsledný sekvenčný diagram, ktorý je možné ďalej upravovať v module „Editor“, prípadne vyexportovať do formátu XMI.



Obr. 3.1 - Schéma prístupu pre generovanie sekvenčných diagramov

4 Analýza toku riadenia

Táto kapitola popisuje postup použitý pre generovanie dát sekvenčného diagramu metódy. Náš prístup analyzuje možné toky kódu metódy a na základe získaných informácii generuje príslušné elementy sekvenčného diagramu. Z týchto dát je následne vytvorené grafické znázornenie diagramu popísané ďalej v tejto práci v kapitole 5.

4.1 Úvod

Náš postup vychádza z práce [9], ktorá popisuje algoritmus pre detekciu základných fragmentov (opt, loop, break, alt) sekvenčného diagramu. Algoritmus ďalej rozširujeme o detekciu viacerých výstupných blokov metódy. K tomu využívame *riadiacu závislosť toku*. Pôvodný algoritmus nedokáže pre takúto metódu vygenerovať diagram korektne. Ďalej pridávame podporu výnimiek, ktoré sú častým prvkom programového kódu. Následne pridávame rozšírenia UML umožňujúce znázorniť tieto programové konštrukty, nakoľko štandard UML podporu pre znázornenie výnimiek a viacerých výstupných blokov neobsahuje. Ďalej k pôvodnému algoritmu pridávame detekciu strážcov fragmentov. Algoritmus optimalizujeme pre platformu .NET.

Náš postup generuje sekvenčný diagram pre volania v rámci metódy. Nezachycuje už volania volaných metód, atď. Takýto prístup by viedol k značnej veľkosti diagramu a k jeho neprehľadnosti. Pre navigáciu medzi jednotlivými metódami je možné použiť editor diagramov, ktorý je súčasťou implementácie.

Algoritmus pozostáva z troch hlavných častí – príprava, konštrukcia a optimalizácia. Nasleduje popis jednotlivých častí.

4.2 Príprava

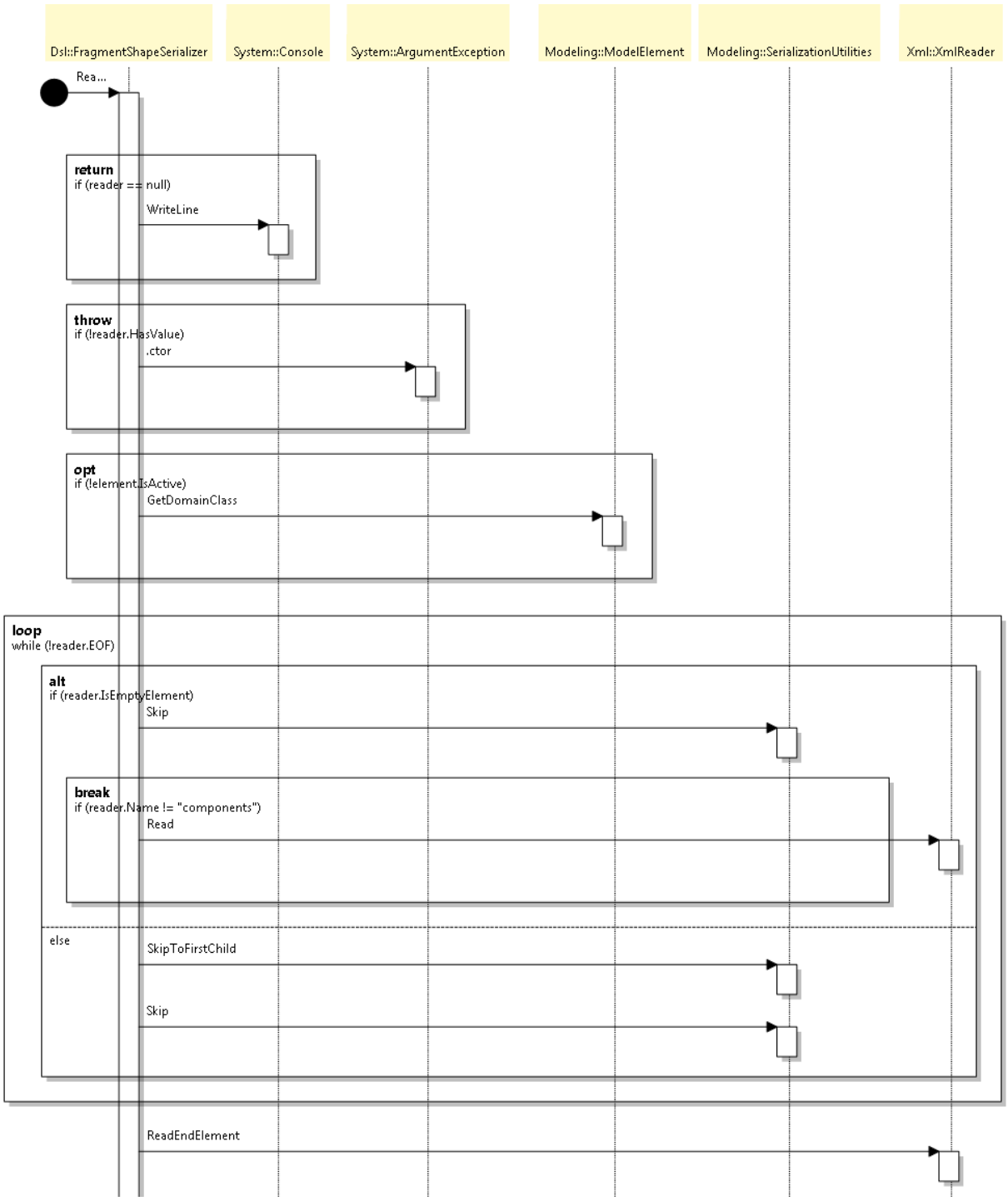
Vstupom algoritmu je graf riadenia toku (GRT) metódy. Algoritmus prechádza tento graf a následne transformuje získané dáta do elementov sekvenčného diagramu.

4.2.1 Príklad

Použitý algoritmus demonštrujeme na príklade na Obr. 4.1. Výsledný sekvenčný diagram vygenerovaný implementáciou práce znázorňuje Obr. 4.2. Kód je napísaný v jazyku C# a znázorňuje reálny kód metódy použitý v implementácii práce. Príklad je pre názornosť doplnený o dodatočný "dummy" kód tak, aby výsledný diagram pokrýval všetky fragmenty analyzované v tejto práci.

```
[1]    public void ReadChild(
[2]                SerializationContext serializationContext,
[3]                SequenceDiagram element,
[4]                XmlReader reader)
[5]            {
[6]                if (reader == null) //return
[7]                {
[8]                    Console.WriteLine("Reader is null");
[9]                    return;
[10]               }
[11]
[12]               if (!reader.HasValue) //throw
[13]                   throw new ArgumentException("reader");
[14]
[15]               if (!element.IsActive) //opt
[16]                   element.GetDomainClass();
[17]
[18]               while (!reader.EOF) //loop
[19]               {
[20]                   if (reader.IsEmptyElement) //alt
[21]                   {
[22]                       SerializationUtilities.Skip(reader);
[23]
[24]                       if (reader.Name != "components") //break
[25]                       {
[26]                           reader.Read();
[27]                           break;
[28]                       }
[29]                   }
[30]                   else
[31]                   {
[32]                    SerializationUtilities.SkipToFirstChild(reader);
[33]                    SerializationUtilities.Skip(reader);
[34]                   }
[35]               }
[36]
[37]               reader.ReadEndElement();
[38]           }
```

Obr. 4.1 - Kód použitého príkladu v jazyku C#



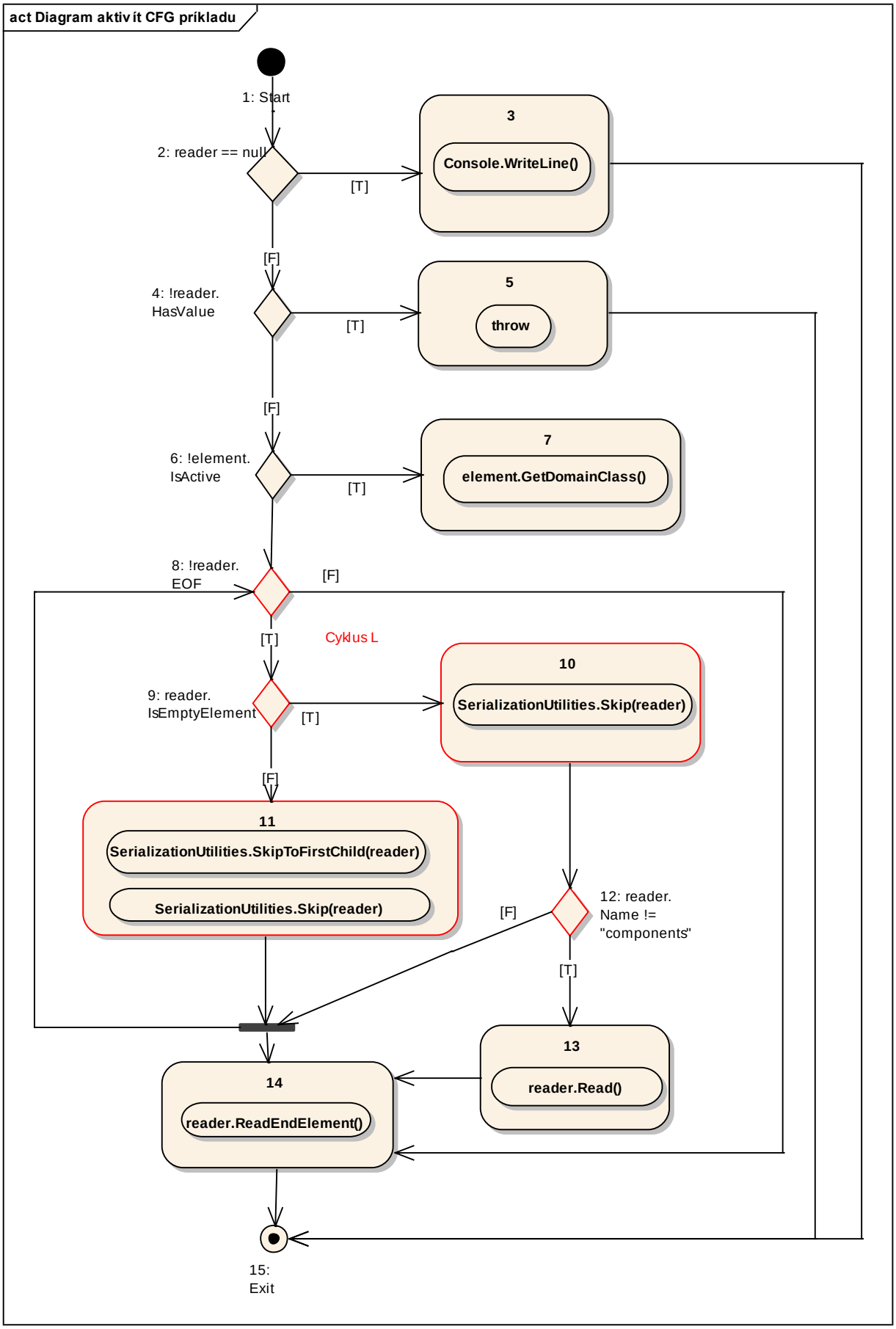
Obr. 4.2 – Výsledný sekvenčný diagram pre použitý príklad

4.2.2 Graf riadenia toku

Graf riadenia toku (Control-Flow Graf, GRT) je grafová reprezentácia toku programu alebo jeho časti. Vrcholy grafu tvoria súvislé postupnosti kódu ukončené inštrukciou skoku alebo vetvy. Hrany grafu reprezentujú možné toky medzi vrcholmi.

GRT obsahuje dva špeciálne vrcholy – *vstupný* (entry) a *výstupný* (exit) vrchol. Vstupný vrchol nemá žiadneho predchodcu. Výstupný vrchol nemá žiadneho nasledovníka. GRT má práve jeden vstupný vrchol, ale môže mať niekoľko výstupných vrcholov.

Redukovaný GRT je GRT, ktorý po odobraní všetkých spätných hrán neobsahuje žiadny cyklus [9]. Ďalej v tomto texte pod pojmom GRT rozumieme redukovaný GRT. GRT je možné znázorniť pomocou diagramu aktivít, kde jednotlivé aktivity zodpovedajú vrcholom grafu, hrany medzi aktivitami znázorňujú hrany grafu a vetvy diagramu predstavujú vetvy grafu. Diagram aktivít reprezentujúci GRT z použitého príkladu znázorňuje Obr. 4.3.



Obr. 4.3 - Diagram aktivít pre graf riadenia toku z použitého príkladu

4.2.3 Zdroj GRT

V našej implementácii negenerujeme GRT z programového kódu. Podkladom pre GRT je medzikód - vysokoúrovňový assembler, do ktorého sú kompilované všetky programy napísané na platforme .NET. Medzikód obsahuje i niektoré konštrukcie, ktoré sú prístupné až vo vyšších programovacích jazykoch – napr. podpora výnimiek, volanie virtuálnych funkcií, volanie konštruktorov atď. Vďaka tejto vlastnosti je reverzné inžinierstvo z medzikódu jednoduchšie, než v prípade reverzného inžinierstva z binárneho assembleru. Ďalšou výhodou tohto prístupu je možnosť použiť prezentovaný algoritmus pre ľubovoľný zdrojový programovací jazyk platformy .NET (C#, VisualBasic.NET, F#...). Pri generovaní diagramu z medzikódu navyše nie je nutná prítomnosť zdrojových kódov aplikácie.

Generovanie GRT z programovacieho jazyka je značne komplexné. Je potrebné rozlišovať množstvo syntaktických rozšírení daného programovacieho jazyka, ktoré sú sémantický ekvivalentné a pre výsledný sekvenčný diagram sú irelevantné. Napr. v jazyku C# je kód `j = i > 1 ? A.X() : B.Y()`, zapísaný pomocou ternárneho operátora[1] `?:`, sémanticky ekvivalentný bloku kódu `if (i>1) j=A.X(); else j=B.Y();` Oba uvedené tvary sú v sekvenčnom diagrame znázornené rovnakým fragmentom typu alt s rovnakým obsahom a oba sú kompilované do rovnakého tvaru medzikódu. V prípade parsovania už na úrovni programového kódu by sme museli rozlišovať takéto syntaktické odlišnosti.

Generovanie GRT (a následne SD) priamo z programovacieho jazyka na druhej strane vedie k presnému sekvenčnému diagramu. Kompilátor pri kompilácii programu do medzikódu optimalizuje niektoré konštrukty programovacieho jazyka. To môže v niektorých častiach výsledného sekvenčného diagramu spôsobiť znázornenie, ktoré nie je ekvivalentné príslušnému programovému zápisu.

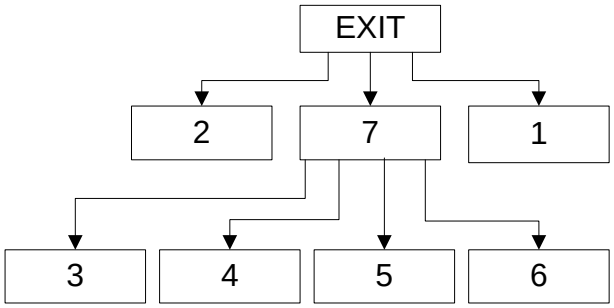
4.2.4 Post-dominancia

Základným termínom prezentovaného postupu je *post-dominancia* (post-dominance). Hovoríme, že vrchol v_1 *post-dominuje* vrcholu v_2, ak každá cesta z vrcholu v_1 do výstupného vrcholu obsahuje vrchol v_2. Vrchol v_2 *bezprostredne post-dominuje* vrcholu v_1, ak v_2 post-dominuje v_1 a akýkoľvek iný post-dominátor v_1, je post-dominátor v_2. V našom príklade na Obr. 4.3 napríklad vrchol 14 post-dominuje vrcholu 6 a vrchol 8 okamžite post-dominuje vrcholu 6.

[1] Ternárny operátor v jazyky C# - http://msdn.microsoft.com/en-us/library/ty67wk28(VS.80).aspx

4.2.5 Post-dominančný strom

Reláciu bezprostrednej post-dominancie je možné znázorniť post-dominančným stromom, kde rodič post-dominuje svojich potomkov a bezprostredne post-dominuje svoje deti. Koreň stromu predstavuje výstupný uzol. Pre konštrukciu post-dominančného stromu je v implementácii použitý algoritmus uvedený v [9]. Post-dominančný strom pre GRT z použitého príkladu (Obr. 4.3) znázorňuje Obr. 4.4.



Obr. 4.4 - Post-dominančný strom pre použitý príklad

4.2.6 Cykly

Programy trávia podstatnú časť svojho času vykonávaním cyklov. V programovacích jazykoch predstavujú cykly napríklad návestia `do-while`, `while`, `for`. V sekvenčnom diagrame sú cykly znázornené prostredníctvom fragmentu typu loop.

Množina vrcholov L v GRT je *cyklus*, ak platí, že [9]

- existuje vrchol $n \in L$ a hrana (m,n), $m \notin L$. Žiadny iný vrchol $v \in L$ okrem vrcholu n nemá predchodcu mimo L. Vrchol n označme ako *hlava(L).*
- pre každý vrchol $v \in L$ existuje cesta, ktorá je celá v L a končí v *hlava*(*L*). Množinu takýchto uzlov označme ako *telo(L).* Platí, že *hlava(L)* $\in$ *telo(L).*

Algoritmus 4.1: Detekcia cyklov v GRT
Vstup: Graf riadenia toku G
Podmienky: Graf riadenia toku je redukovaný
Výstup: Pole cyklov *l*
Postup:

```
[1]     Vytvor DFST strom pre vstupný GRT podľa algoritmu v [10]
[2]     back_edges = {(m,n) | (m,n) je spätná hrana v DFST strome}
[3]     foreach ((m,n) ∈ back_edges)
[4]             ProcessLoop((m,n), l)
void ProcessLoop(back_edge, loops)
[5]     pridaj nový cyklus L do loops
[6]     L.hlava = back_edge.end
[7]     visited = { back_edge.start }
[8]     SearchBack(L.hlava, visited)
[9]     L.telo = visited
void SearchBack(n, visited)
[10]    foreach (p je predchodca n v G)
[11]            if (p ∉visited)
[12]                    visited.add(p);
[13]                    SearchBack(p, visited)
```

Algoritmus 4.1 - Detekcia cyklov v GRT

Algoritmus 4.1 znázorňuje detekciu cyklov v GRT. Najprv prehľadá GRT do hĺbky v depth-first usporiadaní (riadok 1). V tomto usporiadaní je navštívený najprv uzol, následne jeho potomok, ktorý je najviac vpravo, jeho ľavý potomok atď. Všetky hrany navštívené počas prehliadania sú pridané do stromu – označme ho DFST (Depth-First Spanning Tree) pre GRT (riadok 1). Všetky hrany m→n v GRT, pri ktorých v DFST strome m je predchodcom n označme ako spätné hrany (riadok 2). Vrchol n v spätnej hrane m→n tvorí hlavu(L) cyklu L (riadok 6). Spätným prehľadaním do hĺbky z hlavy(L) (riadok 8) nájdeme všetky vrcholy, ktoré majú cestu do n bez toho, aby prechádzali vrcholom m (riadok 7). Tieto vrcholy tvoria množinu telo(L) (riadok 9).

Ďalej definujeme nasledujúce pojmy

- Pre každý cyklus L_i, *OkolnyCyklus*(L_i) je najmenší cyklus L_j, $L_i \neq L_j$, pre ktorý platí $telo(L_i) \subset telo(L_j)$. Ak L_j nie je súčasťou iného cyklu, $OkolnyCyklus(L_j) = null$.
- Pre každý vrchol *n*, $n \in GRT$, *OkolnyCyklus(n)* je najmenší cyklus L taký, že $n \in telo(L)$. Ak n nie je súčasťou žiadneho cyklu $OkolnyCyklus(n)=null$.

Niektoré informácie o cykle L z GRT na Obr. 4.3 znázorňuje Obr. 4.5.

hlava(L) = 8	OkolnyCyklus(9) = L
telo(L) = {8,9,10,11,12}	OkolnyCyklus(3) = null
NasledovnikVetvy(9)=8	NasledovnikCyklu(L)=14
NasledovnikVetvy(6)=8	

Obr. 4.5 - Niektoré hodnoty pre použitý príklad

4.2.7 Vetvy

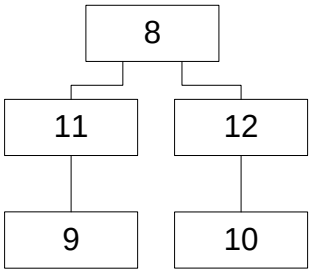
Vrchol n∈GRT je *vetva,* ak z neho vychádzajú aspoň dve hrany. Pre tieto vrcholy implementácia generuje fragmenty typu *alt* a *opt*. Pre každý tok, ktorý určuje vychádzajúca hrana, pridáva algoritmus do "alt" fragmentu *alternatívu*. Pre každú vetvu n ďalej definujeme nasledovníka vetvy n. *NasledovnikVetvy(n)* je vrchol, v ktorom končia všetky alternatívy vychádzajúce z vetvy n. V tomto vrchole pokračuje algoritmus po spracovaní alt fragmentu. *NasledovnikVetvy(n)* pre vrchol n závisí na tom, či *OkolnyCyklus(n)≠null*, t.j, či n je súčasťou nejakého cyklu L.

Ak *OkolnyCyklus(n)=null*, *NasledovnikVetvy(n)* musí postdomonivať všetkým vrcholom z $\{n_i, (n,n_i) \in GRT\}$. Z toho vyplýva, že *NasledovnikVetvy(n)* je najmenší spoločný predchodca v post-dominančnom strome GRT vrcholov z $\{n_i, (n,n_i) \in GRT\}$. Pre výpočet najmenšieho spoločného predchodcu n (LCA) vrcholov $n_i \in$GRT je použitý jednouchý algoritmus. Algoritmus pre každý vrchol n_i ofarbí v post-dominančnom strome GRT cestu z vrcholu do koreňa stromu. Pri poslednom prechode prvý dosiahnutý vrchol n ofarbený všetkými farbami je najmenší spoločný predchodca vrcholov n_i.

Ak *OkolnyCyklus(n)=L, NasledovnikVetvy(n)* má zmysel iba v rámci toku cyklu *L*. Pri analýze nasledovníka vetvy preto najprv eliminujeme všetky hrany, ktoré vedú mimo cyklus L. Takéto hrany reprezentujú buď návestia opúšťajúce cyklus (break) alebo metódu (return, throw). Tieto hrany definujeme ako $exitEdges(L)=(\{n,m\}|n \in telo(L) \wedge m \notin telo(L))$. Pre každú hranu z množiny *exitEdges* ďalej definujeme *BreaksFrom(e)* ako najväčší cyklus L', pre ktorý $e \in exitEdges(L')$. Ak zdrojová implementácia neobsahuje v tele cyklu žiadne riadené návestia skoku, ako napríklad značený break (JAVA) alebo návestie `goto`, tak *BreaksFrom*(*e*)=*OkolnyCyklus*(n); $e=(n,m), e \in exitEdges(L)$. Tieto hrany pri určovaní *NasledovnikVetvy*(*n*) algoritmus ignoruje.

Po odstránení hrán, ktoré vedú mimo cyklus L, môžeme definovať termín *post-dominancie vnútri cyklu.* Nech $n_1,n_2 \in$telo(L). Vrchol n_2 post-dominuje vrcholu n_1 v cykle L,

ak každá cesta z n_1 do hlavy(L) obsahuje n_2. S využitím tejto definície definujeme *post-dominančný strom cyklu L* ako strom, kde hlava(L) je koreňom stromu a každý uzol stromu je post-dominátorom svojich potomkov v cykle L. Post-dominačný strom pre cyklus L z použitého príkladu znázorňuje Obr. 4.6. Pri určovaní *NasledovnikVetvy(n), $n \in L$*, ďalej používame len hrany (n,n_i), ktoré nie sú v množine *exitEdges(L)*. Ak sú takéto hrany aspoň dve, definujeme *NasledovnikVetvy(n)* ako najmenšieho spoločného predchodcu vrcholov n_i v post-dominančnom strome cyklu L. Ak je takáto hrana iba jedna, je zrejmé, že *n* nie je vetva, a teda *NasledovnikVetvy(n)* je nedefinovaný.



Obr. 4.6 - Post-dominančný strom pre cyklus L z použitého príkladu

4.2.8 Nasledovník cyklu

Ak algoritmus detekuje hlavu(L) pre nejaký cyklus L, vytvorí pre neho fragment typu loop. Podobne ako v prípade vetvy, i v prípade cyklu musí algoritmus určiť vrchol, v ktorom bude pokračovať po spracovaní fragmentu typu loop cyklu L. Tento vrchol označíme ako *NasledovnikCyklu*(*L*). Ďalej definujeme *Jump(L)*, ako množinu všetkých hrán $e \in$ exitEdges(L) takých, že BreaksFrom(e) = L. Táto množina obsahuje všetky hrany e, ktoré vedú z cyklu L priamo do cyklu L', ktorý priamo obkolesuje L. Hrany, ktoré vedú cez viacero cyklov (napr. s použitím príkazu `goto` do cyklu L''), budú zaradené v množine Jump(L').

Ak *OkolnyCyklus*(L) = null, NasledovnikCyklu(L) je najmenší spoločný predchodca cieľov hrán z množiny Jump(L) v post-dominančnom strome GRT metódy.

Ak *OkolnyCyklus*(L) = L', NasledovnikCyklu(L) je najmenší spoločný predchodca cieľov hrán z množiny Jump(L) v post-dominančnom strome cyklu L'. Podobne ako v prípade pokračovateľa vetvy, uvažujeme len tok programu, ktorý zostáva v cykle L'. Ak Jump(L) = ∅, NasledovnikCyklu(L) = null. Tento stav nastáva zriedka, napr. ak jediná možnosť opustenia cyklu je skok cez viacero okolitých cyklov.

4.3 Výstupné vrcholy a výnimky

4.3.1 Rozšírenia UML

Jazyk UML 2 prináša výrazné zlepšenia pre zachytenie vyjadrovacej sily programovacieho jazyka. Jazyk UML je ale modelovacím jazykom. Jeho vyjadrovacia sila je nižšia než v prípade programovacieho jazyka. Pre precízne znázornenie toku metódy preto nestačia štandardné UML 2 konštrukty. UML 2 napríklad nedokáže znázorniť predčasné ukončenie toku diagramu zodpovedajúce predčasnému ukončeniu metódy (návestie `return` alebo neošetrené návestie `throw`).

Jedným riešením pre znázornenie predčasného ukončenia toku diagramu je použitie fragmentu typu break. Podľa špecifikácie UML tento fragment poskytuje scenár, ktorý *"je uskutočnený namiesto okolitého fragmentu"*[4]. Takáto definícia fragmentu nie je v našom prípade postačujúca. Navyše pri prítomnosti väčšieho počtu výnimiek a výstupných vrcholov by bol diagram neprehľadný a preplnený fragmentmi typu break.

Ďalšou možnosťou je pridanie nových fragmentov k už existujúcim UML fragmentom. Tieto fragmenty by umožnili presnejšie zachytiť potrebné scenáre. Nové fragmenty bude podporovať i editor diagramov, ktorý je súčasťou našej práce. Nakoľko však štandard UML 2 neumožňuje pridávanie nových fragmentov, pri exporte diagramov je potrebné zvoliť vhodné mapovanie transformujúce nové fragmenty do štandardných UML fragmentov. Pri znázornení sekvenčného diagramu znázorňujúceho programový kód je vždy potrebné zvoliť vhodný kompromis medzi zachovaním UML štandardov a pridávaním nových rozšírení.

Pre znázornenie predčasného ukončenia toku metódy reprezentovaného návestím `return` pridávame nový fragment typu *return*. Uskutočnenie scenára vo fragmente podlieha, podobne ako v prípade fragmentu typu alt či fragmentu typu opt, splneniu strážiacej podmienky.

Pre znázornenie výnimiek pridávame nový fragment typu *throw*. Výsledný diagram zobrazuje iba neošetrené výnimky, ktoré vedú k predčasnému ukončeniu toku metódy. Ostatné (ošetrené) výnimky nepredstavujú významnú zmenu chovania toku, a preto nie sú v diagrame zachytené. Interakcie, ktoré obsahujú bloky ošetrených výnimiek, diagram obsahuje.

Vďaka prítomnosti editora vygenerovaných diagramov je možné pridať i ďalšie fragmenty umožňujúce presnejšie zachytiť jednotlivé konštrukty programového kódu (napr.

fragment typu *switch* pre znázornenie `switch` bloku. Takéto dodatočné fragmenty môžu byť súčasťou prípadných ďalších rozšírení tejto práce.

4.3.2 Riadiace hrany

Pre detekciu fragmentov typu return a throw je najprv potrebné určiť hrany, v ktorých tieto fragmenty začínajú. V okamžiku dosiahnutia takýchto hrán pokračuje tok programu až k jeho predčasnému ukončeniu. Tieto hrany nazveme *riadiace hrany*. Pre nájdenie riadiacich hrán je potrebné definovať pojem *riadiacej závislosti*. Vrchol $n_2 \in GRT$ je *riadiaco závislý* na vrcholu $n_1 \in GRT$, ak existuje cesta z n_1 do n_2 taká, že n_2 post-dominuje všetkým vrcholom cesty s výnimkou vrcholu n_1. Z definície vyplýva, že vrchol n_1 musí byť vetva. Hrana (n_1, n_2) je *riadiacou hranou* vrcholu n, ak n je riadiaco závislý na n_1 a n post-dominuje n_2 alebo $n=n_2$. Každý výstupný vrchol má práve jednu riadiacu hranu. Množinu riadiacich hrán vrcholu x označíme ako *contrEdges(x)*.

Nájdenie riadiacich hrán pre výstupný vrchol x znázorňuje Algoritmus 4.2.

Algoritmus 4.2: Detekcia riadiacich hrán v GRT pre výstupný vrchol x
Vstup: Graf riadenia toku *G*, výstupný vrchol *x*, post-dominančný strom *t* pre GRT
Výstup: contrEdges(x) = {(m,n) | (m,n) je riadiaca hrana vrcholu x}
Postup:

```
[1]     predeccessors = {n | n je potomok x v t ∨ n == x}
[2]     foreach (n ∈ predeccessors)
[3]         foreach (m ∈ { m | (m,n) ∈ GRT})
[4]             if (m nie je potomok x v t)
[5]                 contrEdges.Add((m, n))
```

Algoritmus 4.2 - Detekcia riadiacich hrán v GRT

Pre výstupný vrchol x algoritmus nájde všetky vrcholy n také, že x=n alebo x post-dominuje n (riadok 1). Pre tieto vrcholy n ďalej nájdeme hrany (m,n) také, že x nie je post-dominátorom m. (riadok 4). Každá takáto hrana je rozhodovacou hranou vrcholu x (riadok 5).

4.3.3 Spracovanie násobných výstupných vrcholov

Pri prítomnosti viacerých výstupných vrcholov v GRT je pridaný nový *hlavný výstupný vrchol*. Do tohto vrcholu smerujú všetky ostatné výstupné vrcholy. Takto je možné zachovať post-dominančný strom z kapitoly 4.2.5, pre ktorý platí, že jeho koreň post-dominuje všetkým jeho potomkom.

Pri existencii viacerých výstupných blokov je ďalej potrebné určiť *primárny výstupný vrchol*, v ktorom končí hlavný tok a tzv. *predčasné výstupné vrcholy*. Toky vedúce k predčasným výstupným vrcholom budú znázornené ako fragmenty typu return a throw. Pre

určenie primárneho výstupného vrcholu je použitý Algoritmus 4.3. Pre každý predčasný vrchol sa ďalej vypočíta riadiaca hrana podľa algoritmu uvedeného v 4.3.2 a následne sa pridá do množiny *returnEdges*. Na GRT z Obr. 4.3 sú predčasnými výstupnými vrcholmi vrcholy 3 a 5. Primárny výstupný vrchol je vrchol 14 a hlavný výstupný vrchol je vrchol 15. Do vrcholu 15 smerujú všetky ostatné výstupné vrcholy.

Algoritmus 4.3: Určenie primárnych a predčasných výstupných vrcholov
Vstup: Graf riadenia toku *G*, množina výstupných vrcholov *exits*
Výstup: primárny výstupný vrchol *x*, predčasné výstupné vrcholy *exits*
Postup:
[1] foreach (v∈*exits*)
[2] prehľadaj spätne GRT do šírky od *v*
[3] x = vrchol s najvyšším počtom navštívených vrcholov z bodu 2
[4] odstráň x z *exits*

Algoritmus 4.3 - Určenie primárnych a predčasných výstupných vrcholov

4.3.4 Spracovanie výnimiek

Množina *returnEdges* obsahuje po spracovaní algoritmu z kapitoly 4.3.3 všetky riadiace hrany, ktoré vedú k predčasným výstupným vrcholom. Z tejto množiny odstránime hrany, ktoré vedú k toku, ktorý obsahuje neošetrenú výnimku. Odstránené hrany pridáme do množiny *throwEdges*. Pre nájdenie takýchto tokov využijeme to, že medzikód je vysokoúrovňový assembler (4.2.3) a obsahuje inštrukciu *throw*, ktorá vyvoláva výnimku.

Algoritmus začne prehľadaním všetkých hrán z množiny *returnEdges*. Ak na ceste nájde vrchol GRT obsahujúci inštrukciu *throw*, takúto hranu odstráni z množiny *returnEdges* a následne pridá do množiny *throwEdges*. Výslednú množinu *returnEdges* a *throwEdges* pre použitý príklad znázorňuje Obr. 4.7.

returnEdges = {2,3} throwEdges = {4,5}
contrEdges(3) = {(2,3)} contrEdges(5) = {(4,5)}

Obr. 4.7 - Riadiace hrany pre použitý príklad

4.3.5 Export rozšírení

Štandard UML 2 neumožňuje definovať vlastné fragmenty. Pri exporte diagramu je preto potrebné zvoliť vhodné mapovanie, ktoré transformuje prezentované rozšírenia na štandardné konštrukty jazyka UML. Z dostupných UML fragmentov je sémanticky najbližšie fragmentom typu return a throw fragment typu break. Pri exporte diagramu budú preto tieto fragmenty nahradené fragmentom typu break.

4.3.6 Detekcia strážcov

Algoritmus pri pridávaní nového fragmentu k nemu pridá aj strážcu, ak nejaký existuje. Strážcov detekuje na základe analýzy pomocných informácií, ktoré sú súčasťou skompilovanej aplikácie. Tieto informácie využíva napríklad ladiaci nástroj pri ladení aplikácie vo vývojovom prostredí. Pri neprítomnosti týchto informácií vygenerovaný sekvenčný diagram strážcov neobsahuje.

4.4 Generovanie

Algoritmus 4.4 popisuje postup použitý pre generovanie dát sekvenčného diagramu.

Algoritmus 4.4: Generovanie dát sekvenčného diagramu
Vstup: Graf riadenia toku G; predspracované dáta z kapitoly Analýza toku riadenia4
Výstup: kolekcia obsahujúce všetky interakcie (fragmenty a správy) v GRT
Postup:

```
[1]     vytvor prázdnu kolekciou interakcií s
[2]     ProcessCollection(s, G.start, G.exit)
[3]     AddMessages(G.exit, s)
void ProcessCollection(seq, start, stop)
[4]     L = EnclLoop(start)
[5]     n = stop
[6]     while  n ≠ stop and n ≠ null
[7]             if (EnclLoop(n) ≠ L)
[8]                     L' = EnclLoop(n)
[9]                     ProcessLoop(seq, L')
[10]                    n = NasledovnikCyklu(L')
[11]                    if n = start then n = null
[12]                    continue na [5]
[13]            if (n.messages > 0)
[14]                    AddMessages(s, n)
[15]            returns = {m | (n,m) ∈ returnEdges}
[16]            foreach (m ∈ returns)
[17]                    ProcessReturn(m)
[18]            throws = {m | (n,m) ∈ throwEdges}
[19]            foreach (m ∈ throws)
[20]                    ProcessThrow(m)
[21]            if (L ≠ null)
[22]                    breaks = {m | (n,m)∈ExitEdges(L) && m∉returns && m∉throws}
[23]                    foreach (m ∈ breaks)
[24]                            ProcessBreak(seq, n, m)
[25]            rest =  {m | (n,m) ∈ G && m ∉ breaks && m ∉ returns && m ∉ throws}
[26]            if rest = ∅
[27]                    n = null
[28]            if rest = {m}
[29]                    n = m
[30]            if ((rest = {m1,m2}) ∧ (m1 = NasledovnikVetvy(n) ∨ m2 =
        NasledovnikVetvy(n)))
[31]                    ProcessOpt(seq, n, rest)
[32]            else if rest = {m1,…mk}, k > 1
[33]                    ProcessAlt(seq, n, rest)
[34]                    n = NasledovnikVetvy(n)
[35]            if n = start
[36]                    n = null
```

Algoritmus 4.4 - Generovanie dát sekvenčného diagramu (1. časť)

```
void ProcessLoop(seq, L)
[37]    vytvor nový loop fragment l
[38]    seq.Add(l)
[39]    ProcessCollection(seq.Childs, L.hlava, null)

void ProcessBreak(seq, n, m)
[40]    vytvor nový break fragment b
[41]    seq.Add(b)
[42]    L = BreaksFrom(n,m)
[43]    ProcessCollection(b.Childs, m, NasledovnikCyklu(L))

void ProcessOpt(seq, n, rest)
[44]    vytvor nový opt fragment o
[45]    seq.Add(o)
[46]    foreach (m_i ∈ rest)
[47]            if (m_i ≠ NasledovnikVetvy(n))
[48]                    ProcessCollection(seq_2,m_i, NasledovnikVetvy(n))

void ProcessAlt(seq, n, rest)
[49]    vytvor nový alt fragment a
[50]    seq.Add(a)
[51]    foreach (m_i ∈ rest)
[52]            pridaj do fragmentu a alternatívu a_i
[53]            ProcessCollection(a_i.Childs, m_i, NasledovnikVetvy(n))

void ProcessReturn(seq, m)
[54]    vytvor nový return fragment r
[55]    seq.Add(r)
[56]    ProcessCollection(r.Childs, m, null)

void ProcessThrow(seq, m)
[57]    vytvor nový throw fragment t
[58]    seq.Add(t)
[59]    ProcessCollection(r.Childs, m, null)
```

Algoritmus 4.5 – Generovanie dát sekvenčného diagramu (2. časť)

Vygenerované dáta pre GRT z použitého príkladu obsahujú niektoré redundantné informácie, ako napríklad prázdny fragment typu break alebo volania správ, ktoré sú súčasťou strážcov. Tieto redundantné informácie budú odstránené v ďalšej, optimalizačnej, fáze algoritmu.

Pre použitý príklad, ktorého GRT znázorňuje diagram aktivít na Obr. 4.3 štartuje metóda *ProcessCollection* so vstupným (1) a výstupným vrcholom (15) (riadok 7). V prvej iterácii detekuje (riadok 15) predčasnú „exit“ hranu (2, 3) a následne zavolá metódu *ProcessReturn* (riadok 16). Metóda ProcessReturn pridá do kolekcie nový fragment typu return a rekurzívne zavolá metódu ProcessCollection. ProcessCollection ďalej spracováva tento fragment. Na riadku 14 pridá správu Console.WriteLine() z vrcholu 3 do fragmentu typu return. Pokračovaním vrcholu 3 je exit blok (riadok 29). Nakoľko exit blok neobsahuje žiadne relevantné dáta a nemá žiadneho nasledovníka (riadok 29), v ďalšej iterácii sa spracovanie fragmentu typu return ukončí (riadok 4).

Na riadku 20 algoritmus začne spracovanie fragmentu typu throw, ktorý obsahuje interakcie z vrcholu 5. Riadkom 31 začne spracovanie fragmentu typu opt, ktorý začína vo vrchole 6.

Po dosiahnutí vrcholu 8 začne v ďalšej iterácii ProcessSequence na riadku 7 spracovanie fragmentu typu loop. Algoritmus zavolá metódu *ProcessLoop*, ktorá vytvorí nový fragment a následne rekurzívne zavolá ProcessSequence pre spracovanie obsahu fragmentu cyklu L. Pri tomto volaní algoritmus detekuje výstupnú hranu z vetvy *!reader.EOF* do vrcholu 14 (riadok 22), pre ktorú zavolá metódu *ProcessBreak* (riadok 24) s n = vetva !reader.EOF a m = 14. Metóda *ProcessBreak* vytvorí nový fragment typu break. Následne zavolá ProcessSequence so start = m = 14 a stop = NasledovnikCyklu(L) = 14. Vďaka nesplneniu podmienky na riadku 6 fragment typu break zostane prázdny. Volanie sa vráti o úroveň späť. Po dosiahnutí vrcholu 9 začne spracovanie fragmentu typu alt (riadok 33), v ktorom sa vytvoria dve alternatívy. Prvá alternatíva bude po spracovaní vrcholu 11 obsahovať fragment typu break. Konečný výstup algoritmu znázorňuje Obr. 4.8.

```
Seq.Dsl.FragmentShapeSerializer.ReadChild(Microsoft.VisualStudio.Modeling.SerializationContext,Seq.Dsl.Sequ
enceDiagram,System.Xml.XmlReader)
  Seq.Core.StaticAnalysis.ReturnFragment [reader == null]
    System.Console.WriteLine(System.String)
  System.Xml.XmlReader.get_HasValue
  Seq.Core.StaticAnalysis.ThrowFragment [!reader.HasValue]
    System.ArgumentException.#ctor(System.String)
  Microsoft.VisualStudio.Modeling.ModelElement.get_IsActive
  Seq.Core.StaticAnalysis.OptFragment [if (!element.IsActive)]
    Microsoft.VisualStudio.Modeling.ModelElement.GetDomainClass
  Seq.Core.StaticAnalysis.LoopFragment [while (!reader.EOF)]
    System.Xml.XmlReader.get_EOF
    Seq.Core.StaticAnalysis.BreakFragment
    System.Xml.XmlReader.get_IsEmptyElement
    Seq.Core.StaticAnalysis.AltFragment [if (reader.IsEmptyElement) //alt]
      Seq.Core.StaticAnalysis.AltFragment+Alternative [if (reader.IsEmptyElement) //alt]
        Microsoft.VisualStudio.Modeling.SerializationUtilities.SkipToFirstChild(System.Xml.XmlReader)
        Microsoft.VisualStudio.Modeling.SerializationUtilities.Skip(System.Xml.XmlReader)
      Seq.Core.StaticAnalysis.AltFragment+Alternative
        Microsoft.VisualStudio.Modeling.SerializationUtilities.Skip(System.Xml.XmlReader)
        System.Xml.XmlReader.get_Name
        System.String.op_Inequality(System.String,System.String)
        Seq.Core.StaticAnalysis.BreakFragment [reader.Name != "components"]
          System.Xml.XmlReader.Read
  System.Xml.XmlReader.ReadEndElement
```

Obr. 4.8 - Výstup algoritmu 4.4 pre použitý príklad

```
Seq.Dsl.FragmentShapeSerializer.ReadChild(Microsoft.VisualStudio.Modeling.SerializationContext,Seq.Dsl.Seque
nceDiagram,System.Xml.XmlReader)
  Seq.Core.StaticAnalysis.ReturnFragment [reader == null]
    System.Console.WriteLine(System.String)
  System.Xml.XmlReader.get_HasValue
  Seq.Core.StaticAnalysis.ThrowFragment [!reader.HasValue]
    System.ArgumentException.#ctor(System.String)
  Seq.Core.StaticAnalysis.OptFragment [if (!element.IsActive) //opt]
    Microsoft.VisualStudio.Modeling.ModelElement.GetDomainClass
  Seq.Core.StaticAnalysis.LoopFragment [while (!reader.EOF) //loop]
    Seq.Core.StaticAnalysis.AltFragment [if (reader.IsEmptyElement) //alt]
      Seq.Core.StaticAnalysis.AltFragment+Alternative [if (reader.IsEmptyElement) //alt]
        Microsoft.VisualStudio.Modeling.SerializationUtilities.Skip(System.Xml.XmlReader)
        Seq.Core.StaticAnalysis.BreakFragment [reader.Name != "components"]
          System.Xml.XmlReader.Read
      Seq.Core.StaticAnalysis.AltFragment+Alternative [else]
        Microsoft.VisualStudio.Modeling.SerializationUtilities.SkipToFirstChild(System.Xml.XmlReader)
        Microsoft.VisualStudio.Modeling.SerializationUtilities.Skip(System.Xml.XmlReader)
  System.Xml.XmlReader.ReadEndElement
```

Obr. 4.9 - Výstup algoritmu 4.4 po prebehnutí optimalizačnej fázy

4.5 Optimalizácia

Účelom optimalizačného procesu je odstrániť redundantné informácie, ktoré vznikli počas výstupu Algoritmu 4.4, a tým zlepšiť jeho čitateľnosť. Optimalizácia používa nasledujúce kroky:

- Všetky prázdne fragmenty sú odstránené.
- Ak je fragment typu alt a obsahuje prázdnu alternatívu, táto alternatíva sa z fragmentu odstráni. Ak po odstránení tejto alternatívy, obsahuje fragment už iba jednu alternatívu, je táto alternatíva transformovaná do fragmentu typu opt.
- Fragment typu break je odstránený, ak je
    1. prázdny a
    2. je súčasťou fragmentu typu *L* typu loop, pričom je jeho prvým alebo posledným fragmentom a
    3. po prerušení fragmentu *L* sa tok vráti do fragmentu, ktorý je okolitým fragmentom L, t.j. break neskáče cez viac než jednu úroveň.

  Výstup Algoritmu 4.4 po prebehnutí optimalizačnej fázy znázorňuje Obr. 4.9.

4.5.1 Čistenie fragmentov

Strážca môže podmieňovať vykonanie fragmentu i volaním metódy. Takéto volanie je v medzikóde kompilované ešte pred volaním kódu z fragmentu. Nakoľko náš algoritmus postupuje sekvenčne, nie je schopný rozlíšiť, či nájdené volanie metódy je alebo nie je súčasťou strážcu. Pre takéto volanie preto vytvorí správu. Následne táto správa bude v diagrame duplicitne – ako správa a ako súčasť strážcu. Z tohto dôvodu je potrebné v optimalizačnej fáze takéto správy rozlíšiť a z výsledného diagramu odstrániť.

5 Editácia diagramov

Súčasťou práce je i editor sekvenčných diagramov. Táto kapitola popisuje prístup použitý pri návrhu tohto editora.

5.1 Porovnanie existujúcich editorov

V súčasnosti je dostupný veľký počet nástrojov umožňujúcich editovať UML diagramy. Detailné zrovnanie väčšiny dostupných UML editorov je možné nájsť v práci [10]. Táto kapitola stručne popisuje editory v citovanej práci neuvedené so zameraním na funkcionalitu reverzného inžinierstva sekvenčných diagramov.

5.1.1 Enterprise Architect

Nástroj Enterprise Architect[1] patrí v súčasnosti špičku medzi nástrojmi určenými pre editáciu UML diagramov. Predstavuje komplexný nástroj podporujúci takmer všetky vlastností UML. Podporuje všetky typy UML 2 diagramov, MDA architektúru ako aj export do XMI 2.1. Nástroj je v súčasnosti jedným z najviac rozšírených UML editorov.

Nástroj podporuje plné round-trip inžinierstvo diagramov tried pre veľké množstvo programovacích jazykov. V prípade sekvenčných diagramov umožňuje ich reverzné inžinierstvo. Pre generovanie používa princíp založený na run-time analýze popísaný v kapitole 3.1. Používa vlastné prostredie, do ktorého užívateľ naimportuje zdrojový kód programu. Z tohto prostredia spúšťa vlastný profiler, ktorý zachytáva jednotlivé volania. Na základe zachytených volaní potom generuje výsledný diagram. Z existujúcich UML fragmentov podporuje jedine fragmenty typu loop. N-krát po sebe idúce volanie metódy zaznamená ako fragment typu loop(n).

5.1.2 Microsoft Visio

Microsoft Visio[2] podporuje okrem UML diagramov veľké množstvo ostatných typov diagramov – napr. obchodné procesy, organizačné diagramy, diagramy potrubného vedenia a prístrojového vybavenia a mnohé ďalšie. V súčasnej verzii 2007 neobsahuje podporu UML 2.0. Ďalšou nevýhodou je nemožnosť XMI exportu a importu. Naopak, nástroj je veľmi dobre integrovaný s Microsoft Word a Microsoft Visual Studio. Visio nepodporuje reverzné a ani dopredné inžinierstvo žiadnych diagramov.

[1] Enterprise Architect – www.sparxsystems.com.au
[2] Microsoft Visio - http://www.microsoft.com/cze/office/programs/visio/highlights.mspx

5.1.3 Visual Studio Team System "Rosario"

Microsoft Visual Studio Team System "Rosario"[1] - nástroj v súčasnosti v štádiu testovacej verzie - je primárne určený pre tímový vývoj. Obsahuje širokú škálu prostriedkov pre tímový vývoj od projektového riadenia, cez správu databázovej schémy, analýzu kódu až po ladenie a automatické testovanie aplikácie. Verzia pre architektov umožňuje reverzné generovanie a editáciu sekvenčných diagramov. V súčasnej verzii je možné generovať diagram pre metódy. Výsledný diagram obsahuje volania správ, neobsahuje však žiadne fragmenty.

5.2 Prehľad možností implementácie editora

Pri výbere technológie na implementáciu editora sme zvažovali niekoľko reálne použiteľných technológií.

Windows Presentation Foundation (WPF) [2] je grafický systém .NET frameworku 3.0. Poskytuje konzistentný programové model pre vývoj aplikácií a jasne oddeľuje užívateľské rozhranie a biznis logiku aplikácie. K zápisu užívateľského rozhrania používa značkovací jazyk XAML (Extensible Application Markup Language). XAML sa líši od ostatných značkovacích jazykoch tým, že jeho elementy predstavujú priamo inštancie spravovaných objektov. Zahŕňa v sebe i množstvo špecializovaných techník, ako napríklad komplexnú podporu pre viazanie dát (Data Binding), vektorovú grafiku, animácie či 3D technológiu. Webovou nadstavbou WPF je Microsoft Silverlight[3] umožňujúci vytvárať webové aplikácie podobné aplikáciam typu Flash s rovnakým kódom ako Windows aplikácie vo WPF. Táto vlastnosť umožňuje mať webovú i Windows aplikáciu s identickým kódom. V našej implementácii by takto existoval jeden kód pre editor diagramov, ktorý by bol použitý pre Windows i webovú aplikáciu. V prípade použitia WPF by však bolo nutné vytvárať editor diagramov úplne od začiatku. Následne by bolo nutné od začiatku implementovať kód zabezpečujúci serializáciu diagramu, podporu undo/redo, validáciu, generáciu artefaktov, zobrazenie panelov nástrojov atď..

Nástroj *Microsoft Visio* spomenutý v kapitole 5.1.2 poskytuje API umožňujúce implementovať vlastnú funkčnosť do tohto nástroja. Týmto spôsobom je možné doprogramovať do Visia podporu UML 2 sekvenčných diagramov (súčasná verzia nepodporuje). Nevýhodou tohto riešenia je nutnosť inštalácie Visia pre prácu s editorom, ako

[1] Visual Studio Team System "Rosario" - http://msdn.microsoft.com/en-us/vstudio/bb936702.aspx
[2] WPF - http://msdn.microsoft.com/en-us/netframework/aa663326.aspx
[3] Microsoft Silverlight - http://silverlight.net/

aj nutnosť doprogramovania ďalších funkcionalít (serializácia, podpora undo/redo, validácia, dátová reprezentácia).

DSL Tools je API rozširujúce prostredie Visual Studia 2005 o podporu doménového vývoja popísaného v kapitole 2.2. DSL Tools umožňuje vytvárať grafické doménové jazyky špecifické pre daný problém. Pridáva podporu pre validáciu vytvoreného jazyka, serializáciu, podporu undo/redo a generáciu artefaktov jazyka. Navyše je plne integrované do prostredia Visual Studia 2005. Jedinou nevýhodou je prítomnosť Visual Studia pre použitie API i vytvoreného jazyka.

Zo spomenutých technológií bola pre editor sekvenčných diagramov po zvážení pozitív a negatív zvolená technológia DSL Tools. Pre sekvenčný diagram sme vytvorili doménový jazyk sekvenčných diagramov popísaný ďalej v kapitole 6.4.1.

5.3 Grafická reprezentácia diagramu

Dáta získané z kapitoly 4 tvoria vstup pre generovanie vizuálnej reprezentácie sekvenčného diagramu. Diagram reprezentujú tvary popísané v kapitole 2.1.1.

Jednotlivé správy medzi objektmi sú transformované do volaní správ medzi lifeline. Každý volaný objekt znázorňuje práve jedna lifeline. Názov triedy je uvedený v hlavičke lifeline. V prípade sekvenčného diagramu metódy vychádzajú všetky správy z jednej lifeline, ktorá predstavuje inštanciu triedy, ku ktorej metóda patrí. Napríklad pre volanie metódy System.Xml.XmlReader.Read() z Obr. 4.9 je vytvorená správa Read() smerujúca od lifeline Seq.Dsl.FragmentShapeSerializer do lifeline System.Xml.XmlReader.

Ak je metóda volaná na rovnakom objekte, je pre ňu vytvorená samo-správa. Ďalej sú do diagramu pridávané jednotlivé fragmenty. Fragmenty obsahujú spolu s typom fragmentu i jeho prípadného strážcu. V prípade fragmentu typu alt sú jeho alternatívy oddelené prerušovanou čiarou.

Obr. 4.9 zobrazuje výstup dát z kapitoly 4 pre použitý príklad z Obr. 4.1. Grafickú reprezentáciu týchto dát, resp. výsledný sekvenčný diagram vytvorený implementáciou editora znázorňuje Obr. 4.2.

5.4 Rozmiestňovanie tvarov

Pri generovaní diagramu na základe dát z kapitoly 4 sa pri pridaní každého nového tvaru volá rozvrhovací algoritmus, ktorý zabezpečuje správne rozloženie tvarov v diagrame. Rozvrhovací algoritmus tvoria dve časti. Prvá časť zabezpečuje správne polohovanie správ

a aktivácií a volá sa pri pridaní novej správy alebo aktivácie. Druhá časť zabezpečuje správne umiestnenie tvarov v rámci fragmentu.

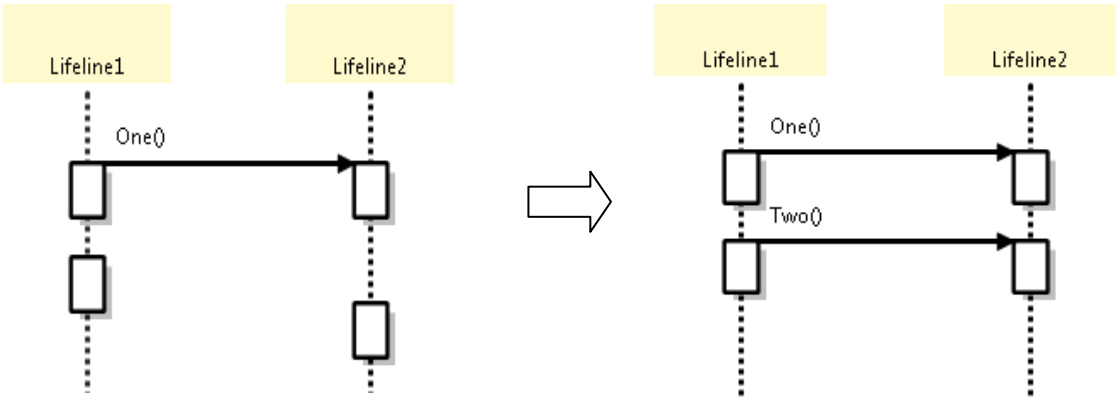
5.4.1 Rozmiestňovanie správ a aktivácií

Sekvenčný diagram predstavuje časovú postupnosť jednotlivých volaní. Túto postupnosť znázorňujú správy a aktivácie v diagrame. Čím je správa umiestnená v diagrame nižšie, tým neskôr dochádza k jej vykonaniu. Implementácia editora diagramov zabezpečuje po každej novej interakcii (pridanie tvaru, odobranie tvaru, atď.) správny tvar sekvenčného diagramu spočívajúci v korektnom umiestnení správ a aktivácií. Aktivácia predstavuje časovú postupnosť volaní v exekučnom procese na lifeline, resp. objekte kde je umiestnená. Tento proces vznikol ako reakcia na prijatú správu v lifeline, resp. ako začiatok volania metódy objektu na nej. Z tohto exekučného procesu môžu vychádzať ďalšie správy spôsobujúce vznik nových aktivácií. Koniec exekučného procesu, resp. aktivácie musí byť preto rovnaký ako koniec cieľovej aktivácie poslednej odoslanej správy v rámci tohto procesu.

Editor ďalej implementuje procesy, ktoré obstarávajú rozloženie tvarov v rámci diagramu, správne rozvrhovanie už existujúcich tvarov a tým zaručujú sémantickú korektnosť diagramu. V prípade správ a aktivácii procesy uskutočňujú v závislosti podľa potreby *Zarovnanie* a *Predĺženie*.

Zarovnanie

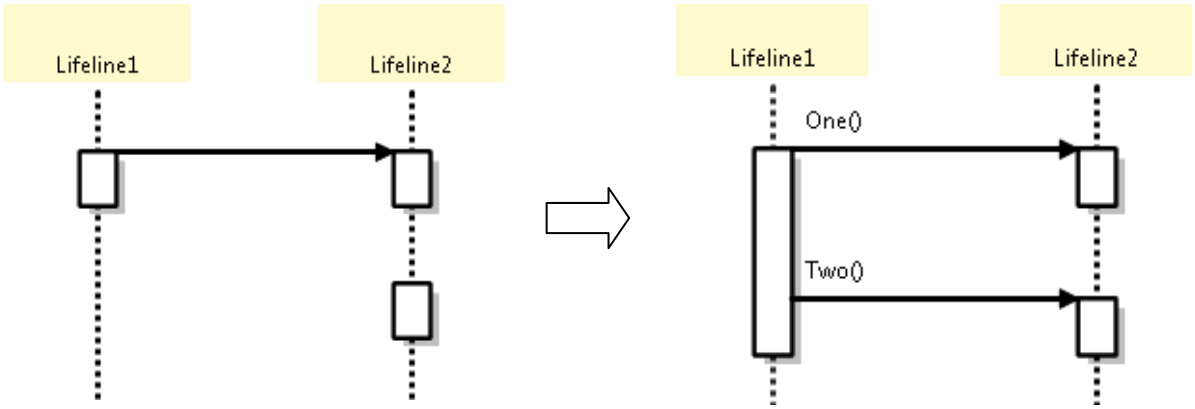
Proces zarovnania zabezpečuje správne umiestnenie nových aktivácií po ich prepojení správou. Správa vzniknutá medzi dvoma prázdnymi aktiváciami predstavuje časovú postupnosť, ktorá vzniká a zaniká súčasne na oboch aktiváciách, preto aj tvary príslušných aktivácií musia byť rovnako veľké a začínajúce na rovnakej pozícii. Príklad použitia zachycuje Obr. 5.1.



Obr. 5.1 - Proces zarovnávania správ a aktivácií

Predĺženie

Tento proces zabezpečuje, aby zdrojová aktivácia bola vždy na úrovni poslednej cieľovej aktivácie z nej vychádzajúcej, nakoľko exekučný proces zdrojovej aktivácie končí s ukončením exekučného procesu aktivácie z nej vychádzajúcej. Príklad použitia procesu znázorňuje Obr. 5.2



Obr. 5.2 - Proces predĺženia správ a aktivácií

5.4.2 Rozmiestňovanie fragmentov

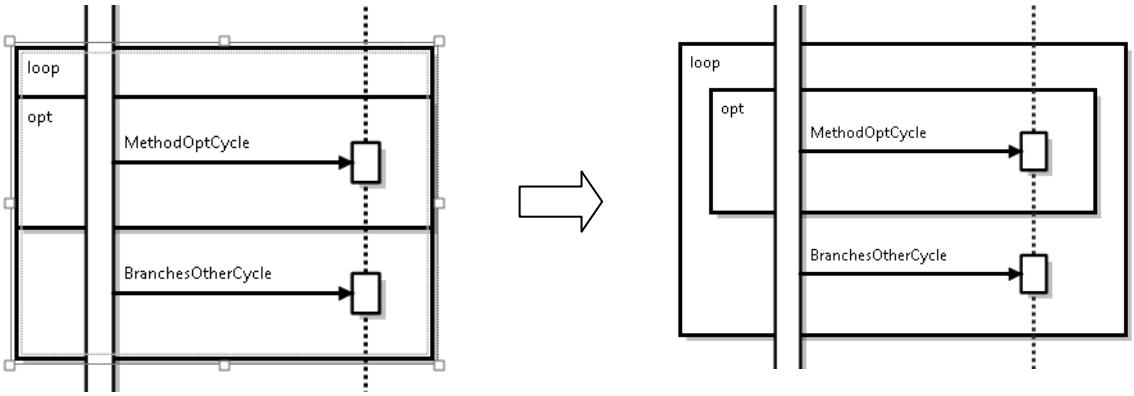
Algoritmus 5.1 obsluhuje správne umiestnenie tvarov v rámci fragmentu. Po pridaní nového tvaru do existujúceho fragmentu je nutné prispôsobiť rozmery fragmentu tak, aby všetky vnorené tvary boli umiestnené vnútri fragmentu a navzájom vzdialené medzi sebou, ako aj od rodičovského fragmentu, podľa definovaných vzdialeností. Polohovací algoritmus fragmentu volá implementácia vždy pri pridaní nového tvaru do existujúceho fragmentu.

Algoritmus 5.1: Rozmiestňovanie vnorených tvarov fragmentu
Vstup: Inštancia triedy Shape reprezentujúca vnorený tvar, ktorý má byť pridaný do fragmentu
Výstup: Pridanie vnoreného tvaru do fragmentu; úprava veľkosti fragmentu
Postup:

```
void AddChild(Shape child)
[1]    childs.Add(child)
[2]    CheckLayout(child is FragmentShape)
void CheckLayout(bool needsResizing)
[3]      Shape pivotChild = childs.Last;
[4]      if (pivotChild is Fragment)
[5]              if (childs.Count == 1)
[6]                              if (needsResizing)
[7]                                      pivotChild.SetWidthAndX(this.GetWidthAndX())
[8]                              pivotChild.SetY(this.GetY() + fragmentYOffset)
[9]                      else
[10]                             pivotChild.AlignByShape(childs.GetPrevious(pivotChild))
[11]     else //mesage
[12]                     if (childs.Count == 1)
[13]                             pivotChild.SetY(this.GetY + messageYOffset)
[14]                     else
[15]                             pivotChild.AlignByShape(childs.GetPrevious(pivotChild))
[16]     EnsureShape()
[17]     if (parent != null)
[18]                     Parent.CheckLayout(needsResizing)
```

Algoritmus 5.1 - Rozmiestňovanie vnorených tvarov fragmentu

Po pridaní nového vnoreného tvaru do fragmentu (riadok 1) sa na riadku 2 volá metóda CheckLayout. V prípade, ak je novo pridaný vnorený tvar fragment a je to zároveň prvý vnorený fragment (5), je potrebné zmeniť šírku fragmentu tak, aby rodičovský fragment pokrýval vnorený fragment. V tomto prípade sa na riadku 7 nastavia rozmery nového vnoreného fragmentu na rozmery rodičovského fragmentu. Metóda EnsureShape() (volaná na riadku 16) ďalej zmení rozmery rodičovského fragmentu tak, aby bol v preddefinovanej vzdialenosti od svojich vnorených fragmentov, a tým rodičovský fragment zväčší. Tento krok znázorňuje Obr. 5.3.



Obr. 5.3 - Použitie rozmiestňovacieho algoritmu

Ak fragment už nejaký vnorený fragment obsahuje, nie je potrebné fragment ďalej zväčšovať, nakoľko sa to uskutočnilo v jednom z predchádzajúcich volaní metódy CheckLayout(). V tom prípade stačí nastaviť Y-ovú súradnicu nového fragmentu do preddefinovanej vzdialenosti od predposledného vnoreného tvaru fragmentu (riadok 10).

Ak je novo pridaný vnorený tvar správa, algoritmus pokračuje riadkom 11. Ak rodič neobsahuje ešte žiadne vnorené tvary, nastaví sa poloha správy podľa polohy rodiča, v opačnom prípade podľa predposledného vnoreného tvaru.

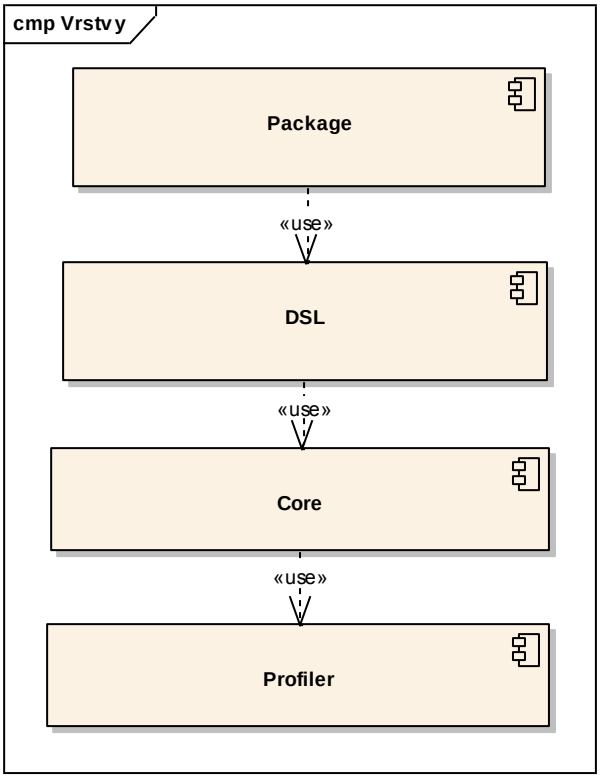
Volanie metódy EnsureShape() (riadok 16) zabezpečí, aby všetky vnorené tvary fragmentu boli umiestnené vo vnútri fragmentu, a to v preddefinovanej vzdialenosti od fragmentu. Metóda vytvorí obdĺžnik, ktorého rozmery sú zjednotením rozmerov všetkých vnorených tvarov fragmentu. Následne sa rozmery obdĺžnika upravia o preddefinovanú vzdialenosť a nastavia sa ako nové rozmery fragmentu.

6 Implementácia

6.1 Architektúra

K implementácii bol použitý programovací jazyk C# a prostredie platformy .NET. Samotná aplikácia je implementovaná do niekoľkých vrstiev. Prezentačnú vrstvu aplikácie – editor diagramov - predstavuje zásuvný modul (plugin) do vývojového prostredia Visual Studio 2005. Pre použitie implementácie je preto potrebné mať nainštalovaný tento nástroj s načítanými zdrojovými kódmi aplikácie, pre ktorú chceme sekvenčné diagramy reverzne generovať.

Jednotlivé vrstvy sú hierarchicky prepojené a navzájom nezávislé, umožňujúce v prípade potreby nahradiť jednu vrstvu inou. Takto je možné napríklad nahradiť prezentačnú vrstvu viazanú na prostredie Visual Studia inou, na tomto prostredí nezávislou, vrstvou umožňujúcou editovať diagramy napríklad v prostredí internetového prehliadača. Štruktúru použitých vrstiev znázorňuje Obr. 6.1. Nasledujúce odstavce popisujú jednotlivé vrstvy.



Obr. 6.1 - Vrstvy implementácie

6.2 Vrstva Profiler

Vrstva Profiler zachytáva jednotlivé volania metód počas behu aplikácie a na základe týchto volaní generuje výstup, ktorý je ďalej podkladom pre Core vrstvu.

Vrstva je implementovaná ako COM objekt v jazyku C++. Implementácia vychádza z článku [11]. Vrstva je spúšťaná z prezentačnej vrstvy pri spustení aplikácie, pre ktorú má byť generovaný sekvenčný diagram. Tesne pred spustením vrstvy Profiler, implementácia nastaví premenné prostredia, na základe ktorých jadro .NET frameworku načíta vrstvu Profiler a pripojí ju k bežiacej aplikácii.

Implementácia vrstvy zachytáva všetky volania v rámci aplikácie a z týchto volaní generuje textový výstup. Príklad výstupu znázorňuje Obr. 6.2. Každý riadok výstupu predstavuje jedno volanie metódy. Formát riadku začína odsadením, ktoré určuje aktuálnu hĺbku zásobníka, pri ktorej volanie začína. Ďalej nasleduje plný názov volanej metódy a identifikátor metódy.

```
DoFactory.GangOfFour.Facade.Structural.MainApp::Main, id=1060912
  DoFactory.GangOfFour.Facade.Structural.Facade::.ctor, id=1061104
    DoFactory.GangOfFour.Facade.Structural.SubSystemOne::.ctor, id=1061328
    DoFactory.GangOfFour.Facade.Structural.SubSystemTwo::.ctor, id=1061456
    DoFactory.GangOfFour.Facade.Structural.SubSystemThree::.ctor, id=1061584
    DoFactory.GangOfFour.Facade.Structural.SubSystemFour::.ctor, id=1061712
  DoFactory.GangOfFour.Facade.Structural.Facade::MethodA, id=1061112
```

Obr. 6.2 - Príklad výstupu Profilera

6.3 Vrstva Core

Vrstva Core obsahuje triedy obsluhujúce aplikačnú logiku aplikácie. Obsahuje kód, ktorý uskutočňuje generovanie metadát sekvenčného diagramu, ktoré sú vstupom pre prezentačnú vrstvu. Vykonáva statickú analýzu metódy a na jej základe generuje pomocou algoritmu popísaného v kapitole 4.4 dáta pre sekvenčný diagram metódy. Ďalej spracováva výstup z vrstvy Profiler, z ktorého generuje dáta pre sekvenčný diagram aplikácie. Príklad výstupu z vrstvy Core znázorňuje Obr. 4.9

6.3.1 Statická analýza medzikódu

Vrstva Core uskutočňuje statickú analýzu medzikódu. Výstupom tejto analýzy je graf riadenia toku, ktorý je vstupom pre algoritmus uskutočňujúci reverzné inžinierstvo popísaný v kapitole 4. V súčasnosti existuje na platforme .NET niekoľko frameworkov, ktoré podporujú statickú analýzu medzikódu.

Open source projekt *Cecil*[1] v súčasnej verzii neposkytuje príliš veľa možností pre statickú analýzu kódu. Ponúka rozšírenú nadstavbu nad vstavaným mechanizmom platformy .NET pre statickú analýzu – reflekciou.

Phoenix[2] je výskumný projekt pre statickú analýzu kódu. Okrem podpory medzikódu dokáže manipulovať aj s binárnymi súbormi. Aktúalna CTP (Community Technology Preview) verzia dokáže generovať napr. cykly, grafy riadenia toku alebo počítať dominanciu. Napriek jeho komplexným možnostiam nie je ešte aktuálna verzia dostatočne zrelá na reálne použitie.

CCI je framework spoločnosti Microsoft. Predstavuje súčasť nástroja pre meranie kvality kódu *FxCop*[3]. Framework podporuje čítanie, zapisovanie a dekompiláciu súborov v medzikóde. Poskytuje komplexné programové API pre prístup k metadátam.

Zo spomenutých frameworkov bol po zvážení pozitív/negatív zvolený CCI framework. Naša implementácia ho používa pre konštrukciu grafu riadenia toku a pre manipuláciu s medzikódom metódy.

6.4 Vrstva Dsl

Vrstva DSL predstavuje prezentačnú vrstvu. Zahŕňa kód pre logiku editora sekvenčných diagramov. Poskytuje implementáciu tvarov, ďalej väzby a algoritmy pre rozloženie tvarov v rámci diagramu. Implementuje serializátor/deserializátor pre čítanie a zápis inštancií diagramov z/do súborov.

6.4.1 Doménový model sekvenčných diagramov

Pre sekvenčný diagram bol pomocou technológie DSL Tools vytvorený doménový model popisujúci jeho štruktúru. Model doménového jazyka sekvenčných diagramov znázorňujú Obr. B.1 a Obr. Obr. B.2 v prílohe. Jednotlivé elementy diagramu reprezentujú ich ekvivalentné domény. Doména je základná stavebná jednotka doménového jazyka. Z domén sa skladajú konštrukty daného jazyka. K doménam sú priradené tvary, ktoré predstavujú ich grafickú reprezentáciu. Tvar domény špecifikuje napríklad geometriu tvaru (obdĺžnik, kruh), výplň, tieň, východzie rozmery atď. Jednotlivé domény sú prepojené väzbami, ktoré určujú vzťahy medzi nimi. Väzby môžu byť buď vložené alebo referenčné s rôznou násobnosťou

[1] Cecil - http://www.mono-project.com/Cecil

[2] Phoenix - http://research.microsoft.com/Phoenix/

[3] FxCop - http://en.wikipedia.org/wiki/FxCop

(1..n, 0..n atď.). Z vytvoreného modelu API DSL Tools umožňuje vygenerovať kostru programovej reprezentácie doménového jazyka.

Príkladom domény v doménovom modeli sekvenčného diagramu je doména `Activation`, ktorá reprezentuje element aktivácie sekvenčného diagramu. K tejto doméne je priradený tvar `ActivationShape` definujúci grafickú reprezentáciu domény. Doména `Activation` je súčasťou dvoch väzieb. Prvá, vnorená väzba, definuje doménu `Activation` ako vnorenú súčasť jej nadradenej domény, čiže `Lifeline`. Druhá, referenčná väzba, predstavuje referenčný vzťah medzi doménami `Activation` - správu.

6.4.2 Modifikácie modelu

Vygenerovaný kód doménového modelu je možné vďaka mechanizmu *partial classes*[1] ďalej upravovať a prispôsobovať tak doménový jazyk.

Príkladom modifikácie je prispôsobenie geometrie tvaru `LifelineShape`, ktorý znázorňuje doménu `Lifeline`. K tvaru bol takto pridaný obdĺžnik zobrazujúci názov lifeline.

Ďalším príkladom modifikácie je vstavaný prepojovací (routing) engine, ktorý obsluhuje prepojenie objektov. V prípade sekvenčného diagramu engine prepája aktivácie a vytvára medzi nimi správy. Engine bol upravený tak, aby po prepojení aktivácií zabezpečil výslednej správe správny tvar a umiestnil jej štart a koniec na správne miesta v jednotlivých aktiváciách.

6.4.3 Pravidlá

Pravidlá v DSL Tools poskytujú robustnú metódu pre implementáciu správania závislého na zmenách v modeli. Pravidlo môže byť použité pre vyvolanie výnimky, zakázanie vyvolanej zmeny alebo na predanie zmeny iným častiam modelu. Pravidlá môžu byť vyvolané, ak sa zmení vlastnosť domény, keď je do modelu pridaná alebo zmazaná inštancia a za ďalších iných podmienok.

Implementácia editora používa niekoľko pravidiel pre dosiahnutie správnej funkčnosti editora sekvenčných diagramov. Napríklad pravidlo `LifelineShapeChangeRule` zaisťuje pri presune Lifeline presun všetkých aktivácií s ňou asociovaných. Pravidlo `ActivationShapeBoundsRule` zasa zaisťuje, aby sa spoločne s aktiváciou posúvali aj správy, ktoré z nej vchádzajú alebo vychádzajú.

[1] Partial classes - http://msdn.microsoft.com/en-us/library/wa80x488(VS.80).aspx

6.4.4 Ukladanie diagramov

Serializáciu a deserializáciu diagramu zabezpečuje použité DSL Tools API. Výsledný diagram je serializovaný do dvoch natívnych xml súborov – prvý súbor obsahuje elementy diagramu. V druhom súbore s príponou .diagram sú umiestnené informácie o rozvrhnutí elementov v diagrame. Diagramy teda nie sú ukladané do XMI formátu, ale do natívneho formátu DSL Tools.

6.5 Vrstva Package

Vrstva Package implementuje zásuvný modul do prostredia Visual Studia 2005. Obsluhuje komunikáciu medzi týmto nástrojom a ostatnými vrstvami aplikácie. Poskytuje kód, ktorý umožňuje prácu s vrstvou DSL v prostredí Visual Studia. Rozširuje prostredie nástroja o dizajnér diagramov. Kostra pluginu bola vygenerovaná pomocou DSL Tools.

Implementáciou nástroja ako pluginu do Visual Studia došlo k prepojeniu vývojového prostredia s prostredím diagramov. Implementácia napríklad využíva vstavaný panel nástrojov Visual Studia - ClassView, ktorý zobrazuje zoznam všetkých tried a ich metód v projekte. Pomocou tohto nástroja si užívateľ vyberá zdrojovú metódu pre vygenerovanie sekvenčného diagramu pomocou reverzného inžinierstvo. Ďalším príkladom integrácie je výstupné okno informujúce o priebehu generovania diagramu.

6.6 Export

Vygenerované diagramy je možné exportovať do formátu XMI 2.1. Export do starších verzií XMI formátu nie je podporovaný. S použitým prístupom je vytvorenie exportu do starších verzií rutinné.

6.6.1 Porovnanie možnosti exportu

Existuje niekoľko možností ako exportovať sekvenčné diagramy do formátu XMI. Sekvenčné diagramy vytvorené pomocou našej implementácie sú perzistované do natívneho XML formátu pomocou serializéra, ktorý poskytuje DSL Tools.

Jednou z možných metód je transformácia perzistovanej reprezentácie diagramu do XMI formátu s využitím jazyka pre transformáciu XML - *XSLT*[1]. Nevýhodou tohto prístupu je prílišná veľkosť a netriviálnosť výsledného XSLT skriptu. Generovanie XMI z DSL

[1] XSLT – www.xslt.org

formátu vyžaduje použitie rozličných funkcií, napr. pre prevod znakov, parsovanie reťazca atď., čo zväčšuje veľkosť výsledného XSLT.

Ďalšou možnosťou je implementácia kódu, ktorý prechádza všetky prvky diagramu a následne podľa ich vlastností generuje výsledný XMI súbor. Nevýhodou tohto prístupu je dlhý kód s veľkým počtom formátovacích blokov. Ďalšou nevýhodou je malá flexibilita takéhoto prístupu – pri úprave XMI výstupu je nutné upraviť a prekompilovať celý kód.

Poslednou možnosťou je použitie *šablónového mechanizmu*, ktorý je súčasťou DSL Tools. XMI formát je možné popísať transformačnú šablónu, ktorá obsahuje štruktúru XMI súboru. Z tejto šablóny sa následne generuje XMI súbor. Šablóna obsahuje dva typy blokov – pevný, ktorý sa vo výstupnom XMI súbore nemení a dynamický, ktorý sa vyhodnocuje pri spustení šablóny a na základe toho produkuje výstup. Dynamické bloky obsahujú popis, akým majú byť jednotlivé elementy sekvenčného diagramu transformované do ich XMI reprezentácie. Dynamické i statické bloky je možné vzájomne kombinovať.

Po zhodnotení kladov a záporov sme použili prístup pomocou šablónového mechanizmu, predovšetkým kvôli jeho efektivite a flexibilite.

6.6.2 Proces exportu

Výsledný XMI súbor vzniká transformáciou elementov sekvenčného diagramu do ich xmi tvaru. Obr. 6.3 znázorňuje xml fragment predstavujúci export správy `Skip()` z príkladu na Obr. 4.2, vedúcej z lifeline `Dsl::FragmentShapeSerializer` do lifeline `Modeling::SerializationUtilities`. Elementy `<lifeline/>` definujú jednotlivé lifeline, z ktorých správy smerujú. Elementy `<fragment/>` špecifikujú aktivácie správy. Element `<message/>` popisuje samotnú správu. Jednotlivé atribúty elementu `<message/>` špecifikujú jej vlastnosti. Napríklad atribút `recieveEvent` označuje fragment, do ktorého správa smeruje, atribút `sendEvent` fragment, z ktorého správa vychádza. Elementy sú identifikované pomocou ich jednoznačného identifikátora umiestneného v atribúte `xmi:id`.

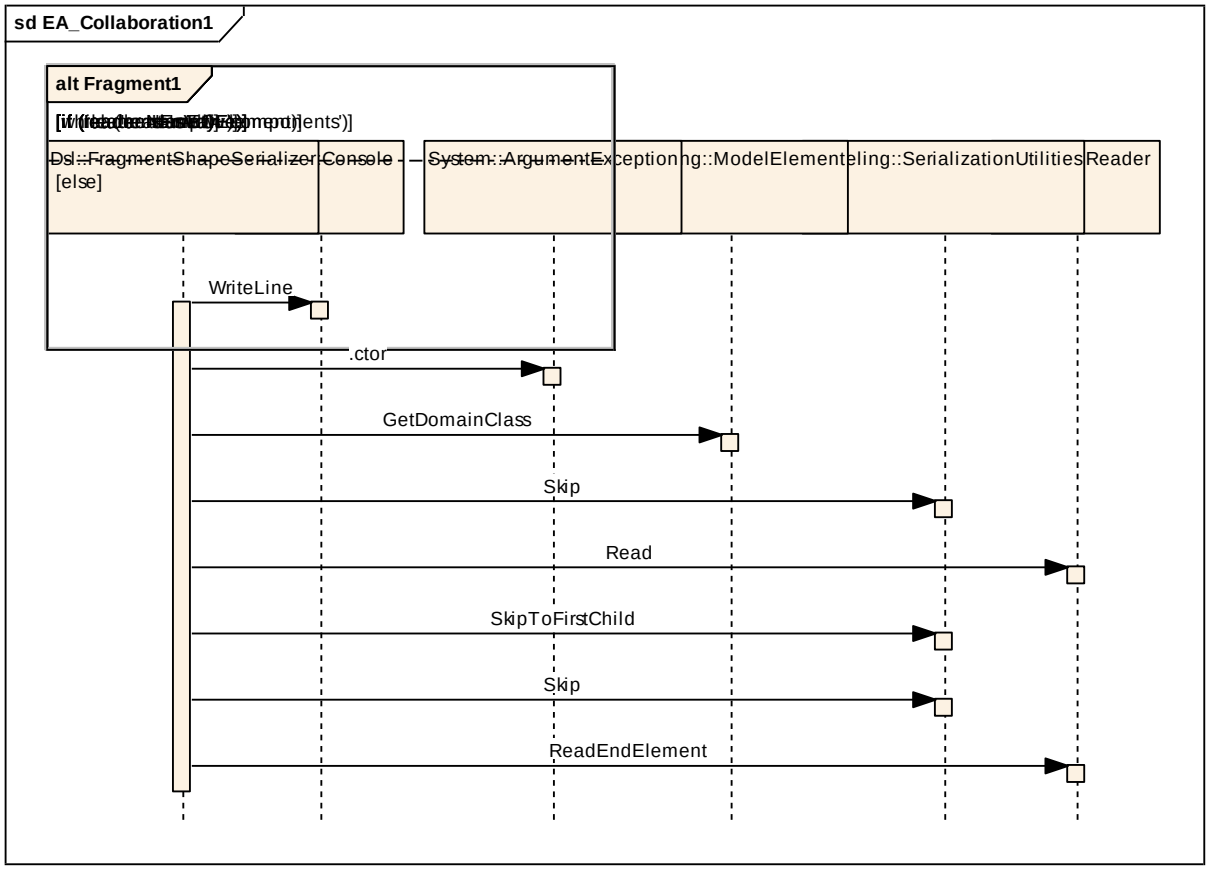
Norma formátu XMI nešpecifikuje spôsob exportu informácií o umiestnení elementov SD (viď 2.1.2). Štruktúra týchto informácií je ponechaná na konkrétnej implementácií výrobcu a je ju možné umiestniť do obsahu elementu `<xmi:Extension/>`. Ďalšou nevýhodou súčasnej verzie XMI normy je nemožnosť definovať, aké správy obsahujú jednotlivé fragmenty.

Naša implementácia umožňuje dve možnosti XMI exportu. V prvom prípade je SD exportovaný striktne podľa XMI štandardu bez akýchkoľvek rozšírení. Druhá možnosť

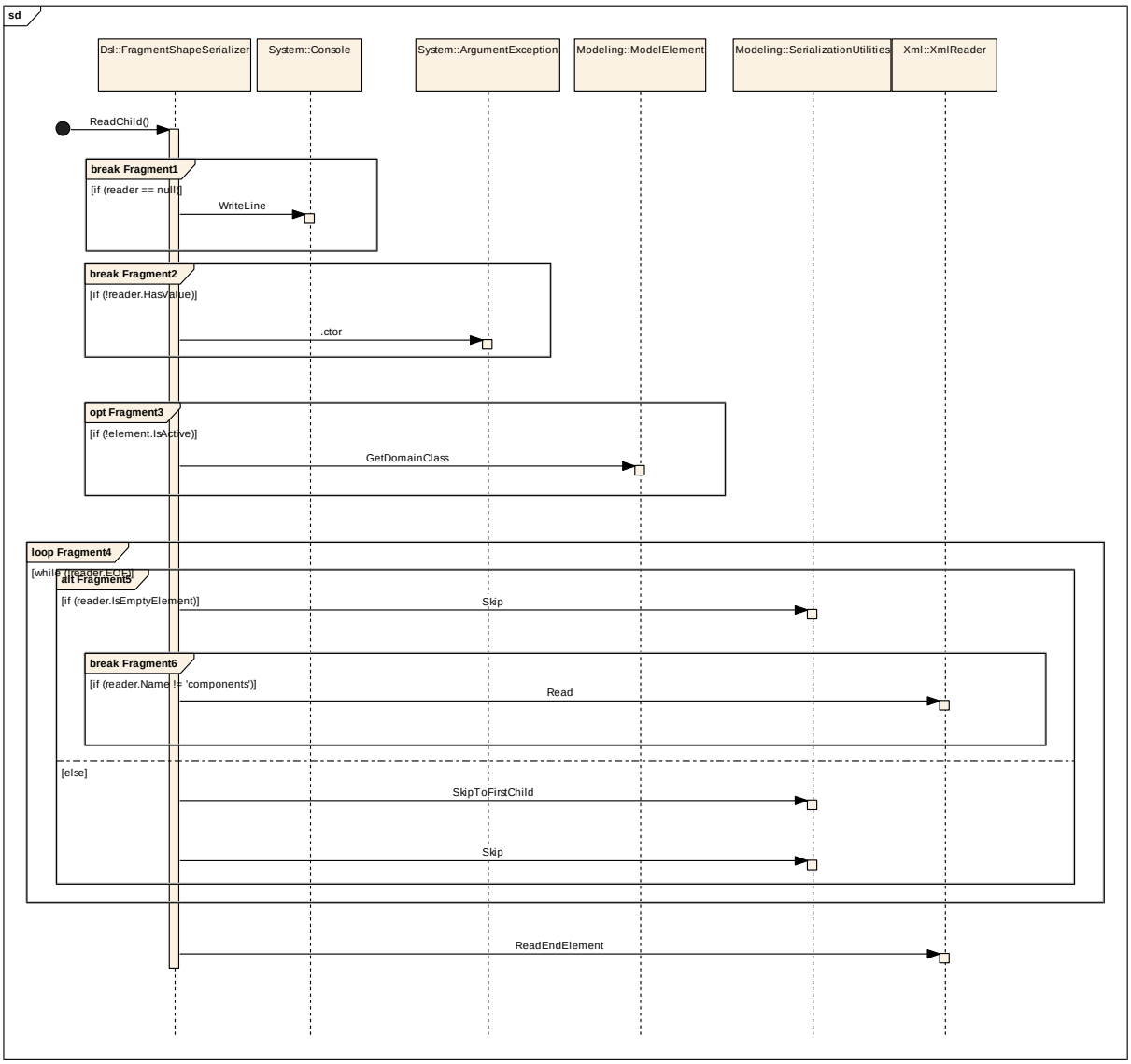
pridáva k XMI súboru ďalšie elementy vložené do elementu <xmi:Extension/>, ktoré obsahujú podrobné informácie o umiestnení a tvaroch jednotlivých elementov SD. Syntax týchto rozšírení je zhodná z XMI výstupom nástroja Enterprise Architect, vďaka čomu je možné pomerne presne naimportovať SD do tohto nástroja pre ďalšie úpravy. Import výstupu exportu podľa XMI štandardu z príkladu použitého v tejto práci znázorňuje Obr. 6.4. Nakoľko XMI export neobsahuje žiadne informácie o umiestnení elementov, jednotlivé lifeline a fragmenty sa prekrývajú. Obr. 6.5 zobrazuje import exportu z použitého príkladu s využitím rozšírených informácií vo formáte nástroja Enterprise Architect. Jednotlivé elementy sú presne umiestnené a tvar SD je veľmi podobný tvaru diagramu z Obr. 4.2.

```
<lifeline xmi:type="uml:Lifeline" xmi:id="8e8a6e46-9dac-4233-9de7-
7bd70ab19c58" name="Dsl::FragmentShapeSerializer" visibility="vis_public"
represents="f305c516-a5b3-4ce5-be3c-e540afcdcc8e"/>
<lifeline xmi:type="uml:Lifeline" xmi:id="5a3608cf-f0a6-4145-b990-
e2f0ed204a75" name="Modeling::SerializationUtilities"
visibility="vis_public" represents="4fcd526f-6537-4716-ad28-
8c60af3b85d3"/>

<fragment xmi:type="uml:OccurrenceSpecification" xmi:id="fe7cbc77-cf5c-
4036-ad59-f6903e39725d" covered="8e8a6e46-9dac-4233-9de7-7bd70ab19c58"/>
<fragment xmi:type="uml:OccurrenceSpecification" xmi:id="339d8e12-bd97-
4169-a557-aa4371643bde" covered="5a3608cf-f0a6-4145-b990-e2f0ed204a75"/>

<message xmi:type="uml:Message" xmi:id="c1f9c4cb-9b2a-4056-99f5-
e270ddcd8378" name="Skip" messageKind="complete" messageSort="synchCall"
sendEvent="fe7cbc77-cf5c-4036-ad59-f6903e39725d" receiveEvent="339d8e12-
bd97-4169-a557-aa4371643bde" />
```

Obr. 6.3 – Ukážka XMI fragmentu



Obr. 6.4 - Import do nástroja Enterprise Architect 6.5 podľa XMI štandardu



Obr. 6.5 - Import do nástroja Enterprise Architect 6.5 s využitím jeho rozšírení XMI formátu

7 Záver

7.1 Zhodnotenie splnenia cieľov

Cieľom tejto práce bolo vytvorenie komplexného riešenia pre reverzné inžinierstvo sekvenčných diagramov postaveného na platforme .NET. Toto riešenie zahŕňa proces generovania dát diagramu, ich vizuálnu reprezentáciu a možnosť exportu diagramov pre úpravu v externých nástrojoch. Zároveň prináša implementáciu uvedených postupov. Hoci je riešenie optimalizované pre platformu .NET, prezentované prístupy je možné preniesť i na iné platformy.

Pre generovanie diagramu aplikácie implementácia práce využíva run-time dynamickú analýzu, pri ktorej zachytáva tok volaní v rámci aktuálne bežiacej inštancie aplikácie. Na základe tejto postupnosti následne generuje výsledný diagram aplikácie.

Komplexnejší prístup je použitý pri generovaní diagramu metódy. Prezentovaný algoritmus analyzuje *graf riadenia toku* (GRT) metódy. Tento graf sa generuje zo skompilovaného tvaru metódy – medzikódu. Algoritmus prechádza graf riadenia toku metódy. Na základe *post-dominancie* vrcholov a analýzy možných skokov a vetiev v grafe detekuje príslušné UML fragmenty. Takto zachycuje opakované (fragment typu *loop*), alternatívne (fragment typu *alt*), voliteľné (fragment typu *opt*) a opúšťajúce (fragment typu *break*) správanie toku metódy. Pre presnejšie znázornenie toku metódy práca ďalej pridáva nové UML fragmenty umožňujúce zachytiť predčasné ukončenie vykonávania metódy. Pri detekcii predčasného ukončenia používame termín *riadiacej závislosti*. Vygenerované fragmenty sú doplnené o prípadnú *strážnu podmienku*, ktorá podmieňuje vykonanie toku vo fragmente. Vygenerované dáta diagramu následne prechádzajú procesom, ktorý ich prevádza do prehľadnejšej formy sériou optimalizačných transformácií.

Dáta sekvenčného diagramu získané reverzným procesom sú následne transformované do jeho vizuálnej reprezentácie. Pre zobrazenie tejto reprezentácie bolo zvolené vývojové prostredie Visual Studio 2005. Vďaka vytvorenia vlastného editora je možné zobraziť i rozšírené fragmenty. Editor bol implementovaný s využitím existujúceho API *DSL Tools*, ktoré slúži pre vytváranie *grafických doménových jazykov*. Pre sekvenčný diagram bol vytvorený grafický doménový jazyk, ktorý ho popisuje. Znázornenie sekvenčného diagramu ako doménového jazyka umožňuje v prípadných ďalších verziách UML jednoduché rozšírenie implementácie o nové konštrukty editáciou doménového jazyka diagramu. Vytvorenie doménového jazyka sekvenčných diagramov pridáva k editácii grafického znázornenia

diagramu ďalšie vlastnosti dostupné doménovým jazykom napísaným pomocou DSL Tools API, akými sú napríklad možnosť použitia šablónovej transformácie a validácie. Šablónová transformácia sa využíva pri exporte diagramu do štandardizovaného formátu *XMI*. Pri tomto procese šablónový engine DSL Tools prechádza jednotlivé elementy sekvenčného diagramu a transformuje ich dáta do xml elementov, ktorých tvar zodpovedá XMI špecifikácii.

7.2 Možné rozšírenia

Prezentovaný prístup prináša niekoľko zaujímavých možností rozšírenia. V prípade generovania diagramu metódy by bolo zaujímavé rozšírenie pridanie viacúrovňového zobrazenia volaní. Výsledný diagram metódy by obsahoval aj postupnosti volaní vnorených metód a tie zasa postupnosti volaní ich vnorených metód atď., až do zvolenej úrovne. Tieto vnorené volania by mohli byť zobrazené v jednom diagrame alebo odkazované pomocou interaktívneho *ref* fragmentu (popísaný v 2.1.1). Ďalším zaujímavým rozšírením by bolo doplnenie výsledného diagramu o niektoré metriky kódu. Napríklad pomocou cyklomatickej zložitosti[1], ktorá udáva počet lineárne nezávislých ciest v zdrojovom kóde, by takto bolo možné určiť, ako často prechádza tok jednotlivými vetvami.

Komplexnejšie rozšírenie implementácie je možné pridaním dopredného inžinierstva. To by umožňovalo generovanie programových artefaktov z diagramu. Takto by vznikla podpora plného round-trip inžinierstva diagramov. Užívateľ by si mohol reverzne vygenerovať diagram metódy, následne napríklad presunúť fragment typu loop na iné miesto diagramu a z tejto úpravy by vygeneroval aktualizovaný kód metódy.

Návrhy uvedené v tejto kapitole môžu byť inšpiráciou pre ďalších autorov zaoberajúcich sa problematikou sekvenčných diagramov.

[1] Cyclomatic complexity - http://en.wikipedia.org/wiki/Cyclomatic_complexity

Referencie

[1] Larman, C. *Applying UML and Patterns.* Prentice Hall, 2001.

[2] Sarma, M., Kundu, D. a Mall, R. *Automatic Test Case Generation from UML Sequence Diagram.* s.l. : Advanced Computing and Communications Conference, 2007.

[3] OMG. Unified Modelling Language Infrastructure, v2.1.2. *http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.* OMG, 2007

[4] OMG. Unified Modeling Language Superstructure, v2.1.2. *http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.* OMG, 2007

[5] Cook, S., Jones G., Kent S., Wills A.C.. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.

[6] Delamare, R., Baudry, B., Traon, Y. L.. *Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces.* IEEE Computer Society, 2006

[7] Briand, L.C., Labiche Y,, Miao Y.. *Towards the Reverse Engineering of UML Sequence Diagrams.* s.l. : IEEE Computer Society, 2003.

[8] Rountev A., Volgin O., Reddoch M.. *Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams.* ACM New York, 2006

[9] Aho, V. A., Lam, S.M., Sethi R., Ulllman, J.D. *Compilers - Principles, Techniques and Tools Second Edition.* Addison Wesley, 2007

[10] Bareš, Libor. *Editor diagramu tříd UML - Bakalářská práce.* Praha : ČVUT, Fakulta elektrotechnická, 2007.

[11] Hacket, Scottt. *Creating a Custom .NET Profiler.* CodeProject.com, 2006 http://www.codeproject.com/KB/dotnet/dotnetprofiler.aspx.

A Obsah DVD

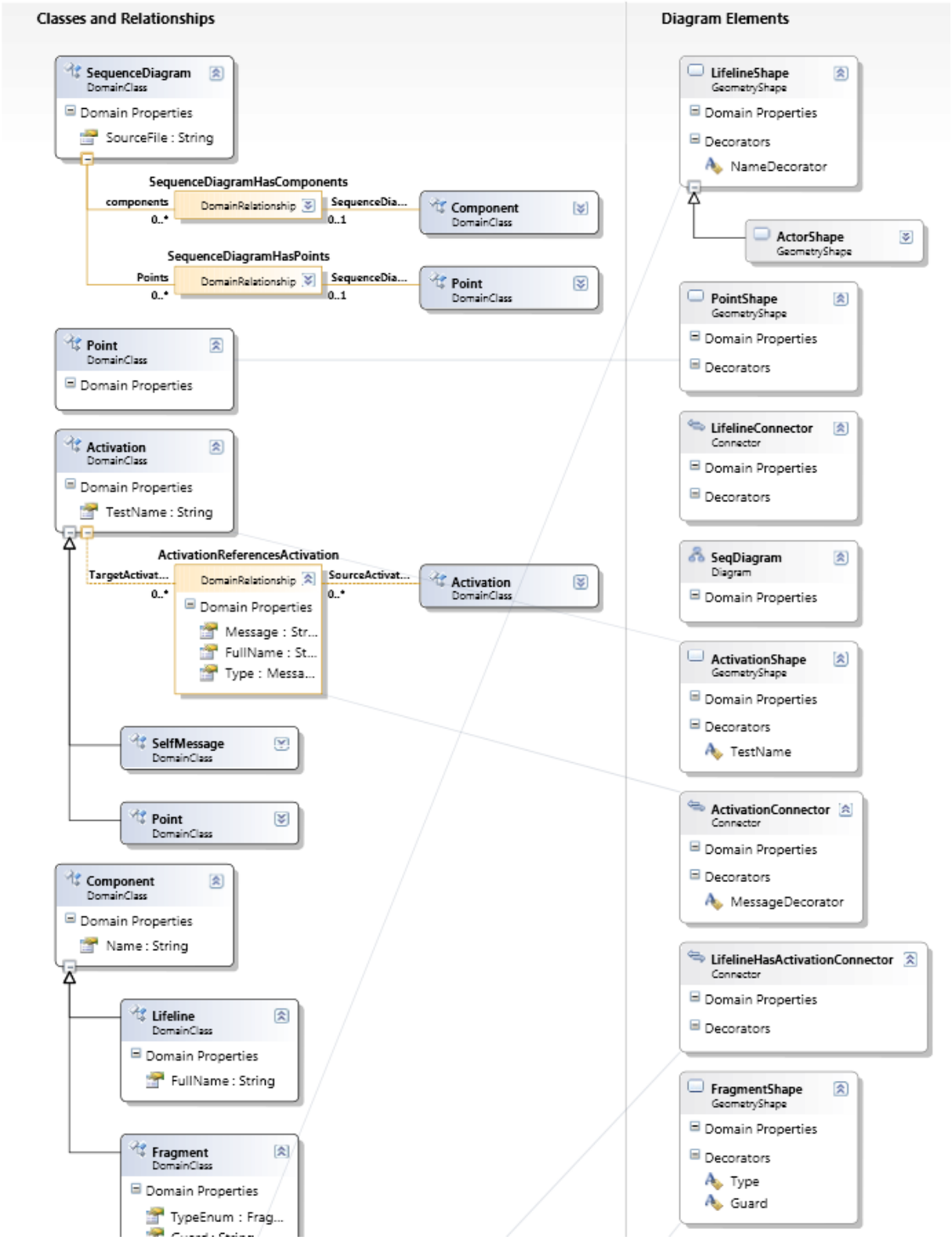
Priložené DVD obsahuje v adresároch

- Doc
  - Uzivatelska-dokumentacia.pdf – užívateľská dokumentácia implementácie
- Examples
  - Outputs – príklady vygenerovaných diagramov. Každý podadresár obsahuje vygenerovaný diagram a zdrojový kód, z ktorého bol diagram vygenerovaný.
  - SeqSample – zdrojové súbory použité pre generovanie príkladov diagramov.
- Src – zdrojové kódy implementácie. Implementácia bola vytvorená s použitím nástroja Visual Studio 2005 ako jedno solution[1]. Solution je ďalej členené do jednotlivých projektov, z ktorých niektoré zodpovedajú vrstvám implementácie popísaným v kapitole 6. Všetky zdrojové kódy sú okomentované XML komentármi.
  - Dsl – vrstva DSL popísaná v kapitole 6.4
  - DslPackage – vrstva implementujúca zásuvný modul (kapitola 6.5)
  - Libs – externé knižnice
  - Profiler – implementácia profilera (kapitola 6.2)
  - Seq.Core – jadro – aplikačná vrstva aplikácie (kapitola 6.3)
  - SeqSetup – zdrojové kódy inštalátora aplikácie
  - Tests – unit a systémové testy aplikácie
- Install – inštalačné súbory
  - Enterprise Architect 6.5 Trial – skúšobná verzia nástroja popísaného v kapitole 5.1.1
  - Seq – inštalačné súbory implementácie. Inštalácia začne spustením súboru setup.exe.
- Thesis
  - Thesis.pdf – text diplomovej práce

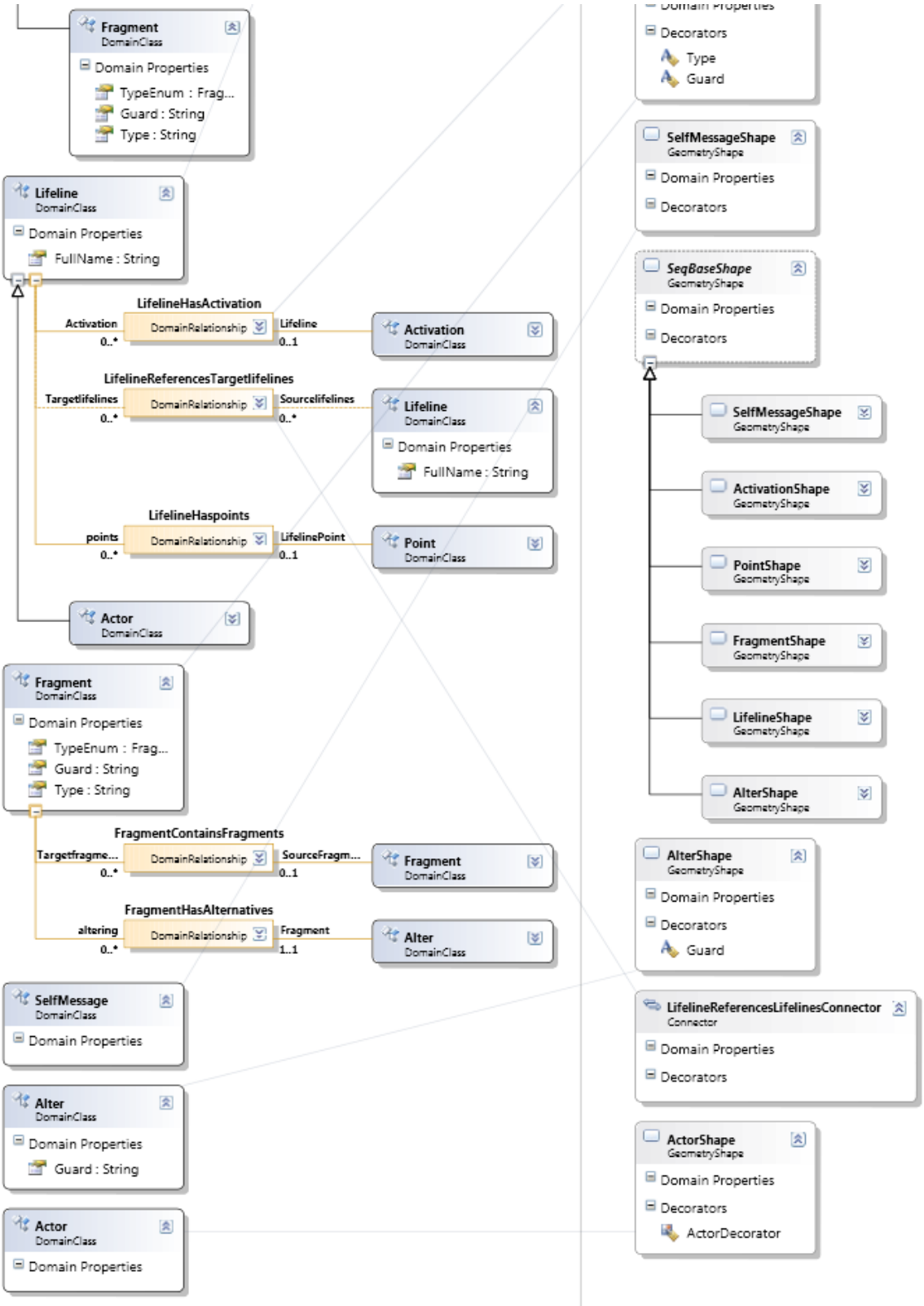
[1] Visual Studio solution – súbor obsahujúci informácie o všetkých projektoch a ďalších súboroch (zdrojové kódy, HTML stránky) použitých pri vývoji aplikácie.

B DSL model sekvenčných diagramov

Obrázok B.1 a Obrázok B.2 znázorňujú doménový model sekvenčných diagramov použitý v tejto práci. Model popisuje kapitola 6.4.1.



Obr. B.1 - DSL model sekvenčného diagramu (1. časť)



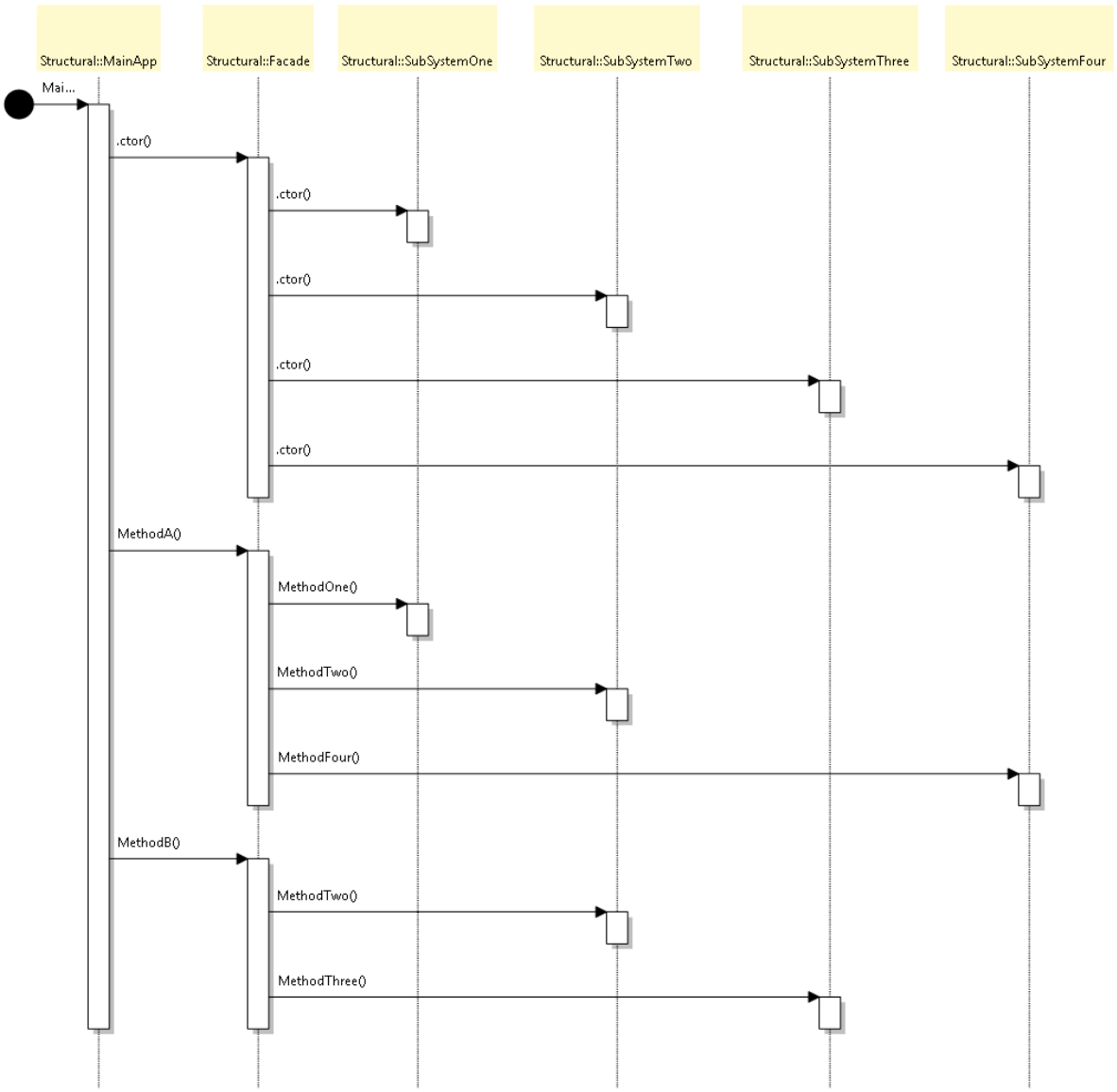
Obr. B.2 - DSL model sekvenčného diagramu (2. časť)

C Príklad sekvenčného diagramu aplikácie

Príklad vygenerovaného sekvenčného diagramu aplikácie, ktorej kód v jazyku C# je na Obr. C.1 znázorňuje Obr. C2. Sekvenčný diagram bol vygenerovaný implementáciou práce s využitím prístupu založenom na run-time dynamickej analýze, ktorý popisuje kapitola 3.1. Kód aplikácie reprezentuje implementáciu návrhového vzoru Facade a je prevzatý zo stránky http://www.dofactory.com/Patterns/PatternFacade.aspx.

```
// Facade pattern -- Structural example
using System;

namespace DoFactory.GangOfFour.Facade.Structural



    // Mainapp test application
    class MainApp
    {
        public static void Main()
        {
            Facade facade = new Facade();

            facade.MethodA();
            facade.MethodB();

            // Wait for user
            Console.Read();
        }
    }

    // "Subsystem ClassA"
    class SubSystemOne
    {
        public void MethodOne()
        {
            Console.WriteLine(" SubSystemOne Method");
        }
    }

    // Subsystem ClassB"
    class SubSystemTwo
    {
        public void MethodTwo()
        {
            Console.WriteLine(" SubSystemTwo Method");
        }
    }

    // Subsystem ClassC"
    class SubSystemThree
    {
        public void MethodThree()
        {
            Console.WriteLine(" SubSystemThree Method");
        }
    }

    // Subsystem ClassD"
    class SubSystemFour
    {
        public void MethodFour()
        {
            Console.WriteLine(" SubSystemFour Method");
        }
    }

    // "Facade"
    class Facade
    {
        SubSystemOne one;
        SubSystemTwo two;
        SubSystemThree three;
        SubSystemFour four;

        public Facade()
        {
            one = new SubSystemOne();
            two = new SubSystemTwo();
            three = new SubSystemThree();
            four = new SubSystemFour();
        }

        public void MethodA()
        {
            Console.WriteLine("\nMethodA() ---- ");
            one.MethodOne();
            two.MethodTwo();
            four.MethodFour();
        }

        public void MethodB()
        {
            Console.WriteLine("\nMethodB() ---- ");
            two.MethodTwo();
            three.MethodThree();
        }
    }
}
```

Obr. C.1 - Zdrojový kód aplikácie (implementácia návrhového vzoru Façade)



Obr. C.2 - Vygenerovaný sekvenčný diagram pre aplikáciu z Obr. C.1