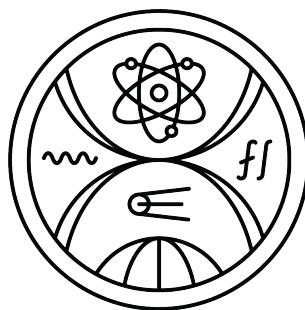


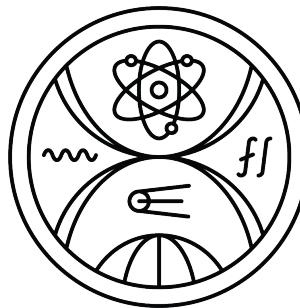
UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



GENEROVANIE ZDROJOVÉHO KÓDU Z  
DYNAMICKÉHO MODELU  
DIPLOMOVÁ PRÁCA

2024  
Bc. MATEJ ČIERNIK

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



GENEROVANIE ZDROJOVÉHO KÓDU Z  
DYNAMICKÉHO MODELU

DIPLOMOVÁ PRÁCA

Študijný program: Aplikovaná informatika  
Študijný odbor: Informatika  
Školiace pracovisko: Katedra aplikovanej informatiky  
Školiteľ: doc. Ing. Ivan Polášek, PhD.

Bratislava, 2024  
Bc. Matej Čiernik



## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Bc. Matej Čiernik  
**Študijný program:** aplikovaná informatika (Jednoodborové štúdium,  
magisterský II. st., denná forma)  
**Študijný odbor:** informatika  
**Typ záverečnej práce:** diplomová  
**Jazyk záverečnej práce:** slovenský  
**Sekundárny jazyk:** anglický

**Názov:** Generovanie zdrojového kódu z dynamického modelu  
*Generating source code from a dynamic model*

**Anotácia:** Zložitosť rozsiahlych softvérových systémov nás núti k výskumu a pokusom zaviesť do oblasti softvérového inžinierstva nové metódy vizualizácie a modelovania, ktoré by podporili ľahšie porozumenie, rozširovanie a znovupoužitie funkcionality a softvérových znalostí v zdrojovom kóde.

**Ciel:** Analyzujte vybrané metódy modelovania v softvérovom inžinierstve (napríklad Executable UML a Object Action Language), interaktívnej grafiky v Unity, ako aj existujúci prototyp animácie dynamiky UML modelu.  
Cieľom práce bude obohatiť existujúci prototyp modelovania softvérovej architektúry a jej funkcionality o možnosť generovať zdrojový kód v jazyku OAL alebo Python z dynamického modelu.

**Literatúra:** JOUAULT, Frédéric, et al. Designing, animating, and verifying partial UML Models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. 2020. p. 211-217.

GREGOROVIČ, Lukáš; POLASEK, Ivan; SOBOTA, Branislav. Software model creation with multidimensional UML. In: Information and Communication Technology-EurAsia Conference. Springer, Cham, 2015. p. 343-352.

**Vedúci:** doc. Ing. Ivan Polášek, PhD.  
**Katedra:** FMFI.KAI - Katedra aplikovanej informatiky  
**Vedúci katedry:** doc. RNDr. Tatiana Jajcayová, PhD.

**Dátum zadania:** 04.11.2022

**Dátum schválenia:** 06.11.2022

prof. RNDr. Roman Ďuríkovič, PhD.  
garant študijného programu

.....  
študent

.....  
vedúci práce

**Pod'akovanie:** Týmto by som chcel pod'akovať svojmu školiteľovi, doc. Ing. Ivanovi Poláškovi, PhD. za všetky rady a odborné usmernenia, ktoré mi adresoval. Svoju vd'aku za veľkú podporu počas celého štúdia by som rád adresoval aj mojej rodine.

# Abstrakt

Hlavný cieľ diplomovej práce spočíval vo vytvorení funkcionality umožňujúcej generovanie zdrojového kódu z dynamického modelu tak, aby bol vygenerovaný kód použiteľný v existujúcom prototype modelovania softvérovej architektúry. Dynamické modely, vyjadrené sekvenčnými diagramami, bolo našim riešením umožnené transformovať do kódu v špeciálnom abstraktnom jazyku OAL, pričom takto generovaný kód je zároveň aplikovateľný v animáciach dynamiky UML modelov v spomenutom prototype. V práci sú uvedené teoretické aspekty súvisiace s problematikou modelovania softvéru, ako aj nami využité technológie. Práca obsahuje podrobný návrh implementovaného riešenia a analýzu vytvoreného postupu transformácie sekvenčných diagramov do zdrojového kódu. V poslednej kapitole je uvedená evaluácia finálneho riešenia, ktorou bola zhodnotená relevancia a praktická využiteľnosť našich výsledkov.

**Kľúčové slová:** generovanie zdrojového kódu, sekvenčný diagram, OAL, MDD

# **Abstract**

The main objective of the thesis was to create functionality that allows the generation of source code from a dynamic model in a way that the generated code is usable in an existing prototype for modeling a software architecture. Dynamic models, expressed by sequence diagrams, were enabled by our solution to be transformed into code in a special abstract language OAL, while the code generated in this way is also applicable in the animations of the dynamics of UML models in the aforementioned prototype. In this thesis, the theoretical aspects related to the software modelling problem are presented, as well as the technologies we used. The thesis contains a detailed design of the implemented solution and an analysis of the developed procedure for transforming sequence diagrams into source code. In the last chapter, an evaluation of the final solution is presented to assess the relevance and practical applicability of our results.

**Keywords:** source code generation, sequence diagram, OAL, MDD

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Úvod do problematiky modelom riadeného vývoja</b>	<b>3</b>
1.1 Modelom riadený vývoj . . . . .	3
1.2 xtUML . . . . .	4
1.3 Jazyk OAL . . . . .	5
1.4 Lexikálna a syntaktická analýza . . . . .	6
1.4.1 Lexikálna analýza . . . . .	6
1.4.2 Syntaktická analýza . . . . .	7
1.4.3 Formálne gramatiky . . . . .	8
1.5 ANTLR . . . . .	9
1.5.1 Spôsoby prechádzania vrcholov parsovacích stromov . . . . .	10
1.6 Prehľad spôsobov transformovania dynamických modelov do zdrojového kódu . . . . .	12
<b>2 Návrh riešenia generovania zdrojového kódu</b>	<b>14</b>
2.1 Proces konverzie sekvenčných diagramov do zdrojového kódu . . . . .	14
2.1.1 Vytvorenie sekvenčného diagramu . . . . .	14
2.1.2 Parsovanie sekvenčných diagramov . . . . .	16
2.1.3 Konverzia sekvenčných diagramov do zdrojového kódu . . . . .	23
2.1.4 Diagram aktivít procesu generovania OAL kódu . . . . .	30
2.2 Nástroj AnimArch . . . . .	33
2.2.1 Načítanie a vytvorenie diagramu tried . . . . .	34
2.2.2 Spustenie animácie . . . . .	34
2.2.3 Vytvorenie animácie . . . . .	36
2.2.4 OAL kód v nástroji AnimArch . . . . .	36
<b>3 Náš prístup generovania zdrojového kódu</b>	<b>41</b>
3.1 Prístupy konverzie sekvenčných diagramov do OAL kódu . . . . .	41
3.1.1 Konvertovanie sekvenčného diagramu s jednou správou . . . . .	41

3.1.2	Konvertovanie sekvenčných diagramov s viacerými správami a lifelinami . . . . .	42
3.1.3	Konvertovanie sekvenčných diagramov so správami z iných tried	43
3.1.4	Konvertovanie sekvenčných diagramov s vlastnými správami . .	44
3.1.5	Konvertovanie sekvenčných diagramov s fragmentom typu OPT	46
3.1.6	Konvertovanie sekvenčných diagramov s fragmentom typu ALT	47
3.1.7	Konvertovanie sekvenčných diagramov s fragmentom typu LOOP	48
3.1.8	Konvertovanie sekvenčných diagramov s fragmentom typu PAR	49
3.1.9	Konvertovanie sekvenčných diagramov s prázdnym operandom vo fragmente . . . . .	50
3.1.10	Konvertovanie komplexných sekvenčných diagramov . . . . .	51
3.2	Potreba rekurzie pri generovaní OAL kódu . . . . .	52
3.3	Identifikácia nekorektných sekvenčných diagramov . . . . .	53
<b>4</b>	<b>Evaluácia</b>	<b>58</b>
4.1	Postup evaluácie generovania zdrojového kódu . . . . .	59
4.2	Evaluácia vygenerovaného zdrojového kódu . . . . .	60
4.2.1	Prvý prípad evaluácie (kontinuita správ)	60
4.2.2	Druhý prípad evaluácie (postupnosť správ)	62
4.2.3	Tretí prípad evaluácie (komplexnejší sekvenčný diagram)	63
4.2.4	Štvrtý prípad evaluácie (Observer)	65
4.2.5	Piaty prípad evaluácie (Abstract factory)	68
4.2.6	Šiesty prípad evaluácie (Abstract factory s paralelizáciou)	71
4.3	Relevantnosť našej metódy v prototype AnimArch . . . . .	74
4.4	Zhrnutie výsledkov evaluácie . . . . .	75
<b>Záver</b>		<b>76</b>
<b>Literatúra</b>		<b>78</b>
<b>Príloha A</b>		<b>80</b>

# Zoznam obrázkov

1.1	Ukážka kódu v OAL . . . . .	6
1.2	Ukážka procesu lexikálnej a syntaktickej analýzy . . . . .	7
1.3	Prechádzanie parsovacieho stromu listenerom . . . . .	11
1.4	Prechádzanie parsovacieho stromu visitorom . . . . .	11
1.5	Transformačný prístup predstavený v publikácii . . . . .	13
2.1	Ukážka prototypu SQD_Tunder . . . . .	16
2.2	Pravidlá lexera . . . . .	23
2.3	Pravidlá parsera - 1. časť . . . . .	23
2.4	Pravidlá parsera - 2. časť . . . . .	24
2.5	Pravidlá parsera - 3. časť . . . . .	24
2.6	Diagram tried s objektmi sekvenčného diagramu . . . . .	25
2.7	Diagram tried predprípravy na konverziu . . . . .	26
2.8	Diagram tried konverzie do OAL kódu . . . . .	27
2.9	Ukážka obsahu animačného súboru . . . . .	29
2.10	Diagram tried vzoru Builder v našom riešení . . . . .	29
2.11	Diagram aktivít procesu generovania OAL kódu . . . . .	32
2.12	Tok údajov v nástroji AnimArch . . . . .	34
2.13	Ukážka nástroja AnimArch . . . . .	35
2.14	Ukážka vykonávania animácie v nástroji AnimArch . . . . .	35
2.15	Ukážka vytvárania animácie v nástroji AnimArch . . . . .	36
2.16	Časť pravidiel gramatiky v nástroji AnimArch . . . . .	40
3.1	Sekvenčný diagram s jednou správou . . . . .	42
3.2	Sekvenčný diagram s viacerými správami a lifelinami . . . . .	42
3.3	Sekvenčný diagram so správou z inej lifeline . . . . .	44
3.4	Sekvenčný diagram s vlastnou správou . . . . .	45
3.5	Sekvenčný diagram s fragmentom typu OPT . . . . .	47
3.6	Sekvenčný diagram s fragmentom typu ALT . . . . .	48
3.7	Sekvenčný diagram s fragmentom typu LOOP . . . . .	49
3.8	Sekvenčný diagram s fragmentom typu PAR . . . . .	50

3.9	Sekvenčný diagram s fragmentom s prázdnym operandom . . . . .	51
3.10	Komplexný sekvenčný diagram . . . . .	52
3.11	Sekvenčný diagram nespĺňajúci pravidlo 1 a 2 . . . . .	54
3.12	Sekvenčný diagram nespĺňajúci pravidlo 3 . . . . .	54
3.13	Sekvenčný diagram nespĺňajúci pravidlo 4 . . . . .	56
3.14	Sekvenčný diagram nespĺňajúci pravidlo 5 . . . . .	57
3.15	Sekvenčný diagram nespĺňajúci pravidlo 6 . . . . .	57
4.1	Sekvenčný diagram prvého prípadu evaluácie . . . . .	61
4.2	Ukážka očakávaného kódu 1. prípadu evaluácie . . . . .	61
4.3	Ukážka generovaného kódu 1. prípadu evaluácie . . . . .	61
4.4	Sekvenčný diagram druhého prípadu evaluácie . . . . .	62
4.5	Ukážka očakávaného kódu 2. prípadu evaluácie . . . . .	62
4.6	Ukážka generovaného kódu 2. prípadu evaluácie . . . . .	62
4.7	Sekvenčný diagram tretieho prípadu evaluácie . . . . .	63
4.8	Ukážka očakávaného kódu 3. prípadu evaluácie . . . . .	64
4.9	Ukážka generovaného kódu 3. prípadu evaluácie . . . . .	64
4.10	Ukážka očakávaného kódu 4. prípadu evaluácie . . . . .	66
4.11	Ukážka generovaného kódu 4. prípadu evaluácie . . . . .	66
4.12	Sekvenčný diagram štvrtého prípadu evaluácie . . . . .	67
4.13	Sekvenčný diagram piateho prípadu evaluácie . . . . .	69
4.14	Ukážka očakávaného kódu 5. prípadu evaluácie . . . . .	70
4.15	Ukážka generovaného kódu 5. prípadu evaluácie . . . . .	70
4.16	Sekvenčný diagram šiesteho prípadu evaluácie . . . . .	72
4.17	Ukážka očakávaného kódu 6. prípadu evaluácie . . . . .	73
4.18	Ukážka generovaného kódu 6. prípadu evaluácie . . . . .	73
A.1	Kompletný diagram tried nášho riešenia . . . . .	81

# Zoznam tabuľiek

4.1	Porovnanie počtu elementov kódu v 1. prípade evaluácie . . . . .	61
4.2	Porovnanie počtu elementov kódu v 2. prípade evaluácie . . . . .	63
4.3	Porovnanie počtu elementov kódu v 3. prípade evaluácie . . . . .	65
4.4	Porovnanie počtu elementov kódu v 4. prípade evaluácie . . . . .	68
4.5	Porovnanie počtu elementov kódu v 5. prípade evaluácie . . . . .	71
4.6	Porovnanie počtu elementov kódu v 6. prípade evaluácie . . . . .	73

# Úvod

V oblasti vývoja softvérových produktov sa už desaťročia vytvárajú rôzne technológie a metódy, ktorých cieľom je uľahčovať a neustále zlepšovať špecifický proces, akým je vývoj softvéru. Jednou z metodík používaných pri tvorbe softvéru je modelom riadený vývoj (Model-driven development). Jedným z kľúčových prvkov tejto metodiky je používanie modelov počas vývojového cyklu softvéru a automatické generovanie kódu na základe týchto modelov. Prípad, v akom sa napríklad môže aplikovať takýto prístup, je používanie technológie, ktorá z dynamických modelov návrhu softvéru generuje zdrojový kód. Práve generovanie zdrojového kódu z dynamického modelu predstavuje základ našej práce.

Aby sme však konkretizovali hlavný cieľ práce, našou úlohou je implementovať generovanie zdrojového kódu na základe dynamického modelu a takoto funkcionality obohatiť existujúci prototyp modelovania softvérovej architektúry. Presnejšie je však našou úlohou transformovať dynamické modely, vyjadrené pomocou UML sekvenčných diagramov a vytvorené iným, už implementovaným prototypom, do zdrojového kódu. Tento kód má byť generovaný tak, aby bol zároveň využiteľný v programe AnimArch.

Prototyp AnimArch umožňuje animovať UML modely, predovšetkým procesy v diagramoch tried. Aplikovanie cieľa našej práce konkrétnie spočíva v generovaní zdrojového kódu tak, aby bol takto vygenerovaný kód použiteľný v animáciách programu AnimArch. Ako neskôr uvádzame v práci, kód sme sa rozhodli generovať v špeciálnom jazyku OAL, nakoľko tento jazyk je používaný v už implementovaných animáciách prototypu AnimArch. Výhoda využitia tohto jazyka spočíva aj v jeho vysokom stupni abstrakcie.

V nasledujúcich kapitolách postupne predstavujeme riešenia cieľov našej práce. V prvej kapitole popisujeme úzko súvisiacu a vyššie spomínanú metodiku modelom riadeného vývoja. Okrem nej v prvej kapitole spomíname ostatné nami používané technológie a s nimi súvisiace teoretické aspekty. V rámci tejto kapitoly analyzujeme aj predchádzajúce publikácie, ktoré mali podobný cieľ, aký má naša práca, teda publikácie zaoberejúce sa transformáciou zdrojového kódu z dynamických modelov.

Druhá kapitola obsahuje podrobny návrh implementovaného procesu generovania zdrojového kódu z dynamických modelov. Systematicky v nej prezentujeme časti nášho riešenia, spolu s prislúchajúcimi modelmi, kompletnejší opis procesu generovania OAL

kódu, ako aj prototypy, s ktorými sme aktívne interagovali. Jeden z týchto prototypov je nástroj na vytváranie sekvenčných diagramov s názvom SQD\_Tunder, zatiaľ čo druhým je už spomínaný program AnimArch.

Tretia kapitola čitateľovi priblížuje postup transformácie sekvenčných diagramov do zdrojového kódu. V kapitole sa nachádza aj opis a definícia pravidiel, na základe ktorých identifikujeme nevalídne sekvenčné diagramy z pohľadu možnosti ich koreknej konverzie do zdrojového kódu.

V poslednej kapitole sa nachádza evaluácia našich výsledkov. Na základe metodiky bližšie uvedených v tejto kapitole je v rôznych prípadoch analyzovaná relevancia finálneho riešenia generovania zdrojového kódu zo sekvenčných diagramov. V závere poslednej kapitoly je celková relevantnosť nášho riešenia zhodnotená, vrátane jeho praktického využitia v prototype AnimArch.

# Kapitola 1

## Úvod do problematiky modelom riadeného vývoja

V tejto kapitole uvádzame základné pojmy, technológie a aspekty súvisiace s našou pracou. Na záver tejto kapitoly analyzujeme publikácie, ktoré mali podobný cieľ ako naša práca.

### 1.1 Modelom riadený vývoj

Komplexný proces, akým je vývoj softvéru, je výhodné uľahčovať rôznymi technológiami, či adekvátnymi metodikami. Potenciál našej práce tkvie v možnosti preniesť dynamické modely do zdrojového kódu. V skutočnosti však cieľ našej práce súvisí so známou metodikou s názvom **Modelom riadený vývoj**, ktorá sa zvykne označovať skratkou **MDD** (z angličtiny Model-driven development).

**Modelom riadený vývoj** predstavuje prístup k softvérovému inžinierstvu, ktorý zdôrazňuje vytváranie a používanie modelov systému na vysokej úrovni abstrakcie ako primárneho vstupu pre návrh, vývoj a implementáciu softvérových aplikácií. MDD metodika je založená na predpoklade, že vývoj softvéru môže byť výrazne efektívnejší a menej náchylný na chyby, ak sa programovanie na detailnej úrovni nahradí transformáciou modelov priamo do spustiteľného kódu [1].

Model v súvislosti s MDD je formálna špecifikácia niečoho konkrétneho (napríklad banky, smartfónu,...), ktorá je vytvorená tak, aby bola vhodná pre určitú formu analýzy, ako napr. generovanie testovacích scenárov. Modely v MDD by mali byť vyjadrené v jazyku, ktorý zodpovedá úrovni abstrakcie daného modelu a zároveň umožňuje jeho transformáciu do finálneho riešenia pomocou konkrétneho programovacieho jazyka [2].

Kľúčovou charakteristikou MDD je jeho prepojenie na automatizáciu. Pod automatizáciou sa v kontexte MDD myslí automatické generovanie kódu z modelov. Vďaka takému automatickému generovaniu kódu je možné dosiahnuť vyššiu produktivitu,

ako aj spoľahlivosť pri tvorbe softvéru. MDD však zároveň čelí rôznym výzvam, ako je obťažnosť pri preklade modelov na kód, alebo ubezpečenie, či vygenerovaný kód presne reprezentuje navrhnutý model. Riešenie týchto výziev zahŕňa použitie sofistikovaných nástrojov pre modelovanie a automatizované generovanie kódu, ako aj integrované testovanie a overovanie modelov [1].

MDD sa opiera o rôzne štandardy, ktoré definujú, ako by mali byť modely vytvorené a transformované. Príkladom takého štandardu je **Model-Driven Architecture** (teda v preklade „modelom riadená architektúra“, v skratke **MDA**) od organizácie **Object Management Group** (v skratke **OMG**). MDA je prístup poskytujúci usmernenia pre štruktúru špecifikácií vyjadrených prostredníctvom modelov. MDA taktiež poskytuje metodiku pre oddelenie modelov aplikácií od konkrétnych platformových implementácií. Ďalším dôležitým štandardom v súvislosti s MDD je známy štandard **Unified Modeling Language** (v preklade „jednotný modelovací jazyk“, v skratke **UML**), ktorý umožňuje modelovanie softvérových systémov na rôznych úrovniach abstrakcie a ich transformáciu do implementačných artefaktov [1, 2]. Pod štandard UML spadá aj tzv. sekvenčný diagram, ktorého koncept je pre našu prácu kľúčový.

Úspech metodiky MDD, najmä vo veľkých korporáciách, závisí od prekonania technických a kultúrnych prekážok súvisiacich s prijatím nových technológií a metodológií v praxi. Rozvoj a adaptácia štandardov, ako sú MDA a UML, sú kľúčové pre podporu širšieho prijatia MDD a prepojenia medzi modelovaním a implementáciou softvéru v praxi.

Môžeme teda konštatovať, že MDD predstavuje skutočne užitočný prístup v softvérovom inžinierstve, ktorý môže viesť k markantným zlepšeniam v efektivite a kvalite softvérových produktov. Tieto výhody však závisia od správnej implementácie a prijatia súvisiacich štandardov a metodológií v rámci vývojového procesu [1, 2].

## 1.2 xtUML

**Executable Unified Modeling Language** (v preklade „spustiteľné UML“, v skratke **xtUML** alebo **xUML**) je vysokoúrovňový jazyk, resp. metodika, ktorá rozširuje štandard UML o možnosť spustenia a verifikácie UML modelov bez priameho definovania kódu. Táto metodika umožňuje pracovať na vyššej úrovni abstrakcie, vďaka čomu je možné predísť predčasným a často nesprávnym rozhodnutiam pri implementácii. xtUML poskytuje vývojárom definovať systém pomocou modelov, ktoré sú nielen špecifikované, ale aj priamo vykonateľné a preložiteľné do kódu, čím sa zjednoduší celý proces vývoja softvéru [3].

xtUML je úzko prepojené so štandardom **MDA**, spomenutým v predošlej podkapitole. MDA konkrétnie používa UML ako jazyk pre modelovanie softvérových systémov,

pričom xtUML rozširuje možnosti UML o vykonanie a preloženie modelov [3].

Ako Mellor v [3] ďalej uvádza, jazyk xtUML pozostáva z týchto hlavných komponentov:

- **Diagram tried** – v xtUML definuje abstraktné entity v doméne a slúži ako základ pre ostatné modelovacie prvky.
- **Stavový diagram** - modeluje životný cyklus objektov danej triedy a zahrňuje definície všetkých stavov, do ktorých objekt môže prejsť.
- **Diagram aktivít** - špecifikuje súbor akcií, ktoré sa vykonávajú pri vstupe do stavu.
- **Jazyk akcií (Action language)** - jazyk určený na definovanie akcií, ktoré sa majú vykonať v danom modeli.

Ked'že v rámci našej práce do veľkej miery interagujeme so sekvenčným diagramom, dá sa povedať, že do jazyka xtUML svojím spôsobom pridávame aj sekvenčný diagram. Veľmi podstatnou súčasťou jazyka xtUML je však jazyk akcií. Medzi jazyky akcií patrí konkrétnie **jazyk akcií objektov (OAL)**, ktorý predstavuje dôležitú súčasť našej práce. Jazyk OAL bližšie popisujeme v nasledujúcej podkapitole.

### 1.3 Jazyk OAL

**Object action language** (v skratke **OAL**, v preklade „jazyk akcií objektov“) je vysokoúrovňový jazyk, nezávislý od konkrétnej platformy. OAL sa svojou syntaxou vo viacerých smeroch podobá niektorým znáym programovacím jazykom (ako napr. Python alebo Java). Zároveň je však jednoduchší ako veľká časť programovacích jazykov. Cieľom jazyka OAL je dosiahnuť to, aby bol jednoduchý, abstraktný, preložiteľný a uvedomelý modelu, do ktorého patrí. Konkrétnie názvy elementov modelov, pod ktoré spadajú triedy, správy, udalosti, parametre, či porty sú jazykom OAL analyzované. Jazyk OAL je, ako bolo spomenuté, platformovo nezávislý, a preto prostredníctvom komplilátora daného modelu je možné preložiť kód v jazyku OAL do iných jazykov [4].

V kontexte našej práce jazyk OAL využívame aj kvôli jeho už existujúcemu využitiu v rámci prototypu *AnimArch*. O prototype *AnimArch* a o používaní jazyka OAL v tomto nástroji hovoríme konkrétnejšie v kapitole 2.

Na obrázku 1.1 sa nachádza ukážka kódu v jazyku OAL. V ukážke sú postupne zobrazené vytváranie a vymazávanie inštancií konkrétnych tried, potom vytváranie a odstraňovanie relácií k daným objektom a na záver sú zobrazené unárne a binárne operácie.

```
//Creation and deletion of instance
create object instance some of SOME;
create object instance secondSome of SOME;
create object instance car of Car;
create object instance king of King;
delete object instance secondSome;

//(Un)relating instances
relate some to car across R3;
relate king to car across R4;
unrelate king from car across R4;

//Unary and binary operations
a=3*6;
b=a+58;
c=b/4;
d=b%10;
e=true;
f=(a==3);
e=not f;
g=e and f;
h=e or f;
```

Obr. 1.1: Ukážka kódu v OAL [5].

## 1.4 Lexikálna a syntaktická analýza

Jedným z hlavných cieľov našej práce je umožniť generovanie zdrojového kódu z dynamického modelu, vyjadreného sekvenčným diagramom. Proces generovania zdrojového kódu v našom prípade zahŕňa parsovanie, alebo inak povedané syntaktickú analýzu. Proces parsovania je potrebný kvôli analyzovaniu jednotlivých elementov a procesov dynamického modelu, pričom tieto elementy sú vyjadrené v štruktúre formátu JSON, čo viac priblížime v kapitole 2. Preto v tejto podkapitole približujeme parsovanie ako také. Koncept samotného parsovania vysvetlíme najskôr na popísaní lexikálnej analýzy a následne na uvedení syntaktickej analýzy.

### 1.4.1 Lexikálna analýza

**Lexikálna analýza**, nazývaná aj ako tokenizácia, je prvou fázou kompliacie alebo interpretácie programového kódu. Lexikálna analýza konkrétnie predstavuje proces konverzie sekvencie znakov zdrojového kódu na sekvenciu tokenov. Tieto tokeny predstavujú základné stavebné prvky syntaxe programovacích jazykov, resp. najmenšie významové jednotky, pričom tokeny môžu byť napr. kľúčové slová, identifikátory, či konštanty [6].

Lexikálny analyzátor, známy tiež ako lexer alebo tokenizer, skenuje zdrojový kód znak po znaku a uskutočňuje kategorizáciu každého znaku do príslušných tokenov podľa definovaných pravidiel. Lexikálny analyzátor tiež zodpovedá napr. za odstránenie komentárov či tzv. bielych znakov (v angličtine „white spaces“) [7].

Proces lexikálnej analýzy začína definíciou toho, čo v jazyku predstavuje token, pričom sa na tento účel využívajú regulárne výrazy, resp. gramatiky. Tieto definície sú

potom preložené do abstraktného výpočtového modelu na rozpoznávanie tokenov, čo je obvykle nedeterministický konečný automat (NKA), ktorý sa následne transformuje na deterministický konečný automat (DKA) [7].

Výsledkom lexikálnej analýzy je prúd tokenov, ktoré sú ďalej spracovávané syntaktickým analyzátorom. Tento prúd tokenov je teda základným vstupom pre syntaktickú analýzu, o ktorej píšeme v ďalšej podkapitole.

#### 1.4.2 Syntaktická analýza

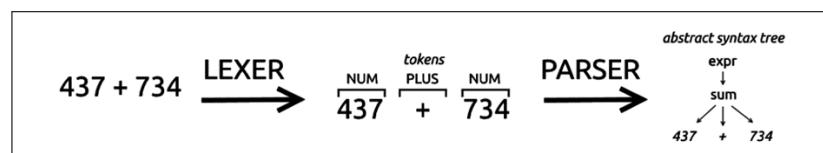
**Syntaktickú analýzu** je možné pomenovať aj pojmom parsovanie, syntaktický analyzátor, či parser. Úlohou syntaktického analyzátoru je čítať postupnosť tokenov, ktorá bola vygenerovaná lexerom a následne vytvoriť stromovú štruktúru. Táto stromová štruktúra zobrazuje gramatickú štruktúru toku jednotlivých tokenov. Stromová štruktúra je obvykle reprezentovaná ako syntaktický strom, alebo inak nazývaný aj parsovaci strom. Parsovaci strom vyjadruje logickú štruktúru daného vstupného reťazca [7].

Je dôležité poznamenať, že syntaktickou analýzou nie je možné kontrolovať sémantiku daných výrazov. Sémantika výrazov sa po syntaktickej analýze realizuje samostatným procesom – **sémantickou analýzou**. **Sémantická analýza** môže byť realizovaná rôznymi spôsobmi. Napríklad, po syntaktickej analýze je možné realizovať kontrolu výsledkov parsovania (teda či dáta, ktoré vznikli parsovaním, sú korektné) a následne je možné realizovať spracovanie dát, ktoré boli abstrahované parsovaním [6, 7].

Proces lexikálnej a syntaktickej analýzy si môžeme ukázať na výraze:

437 + 734

Výsledok procesu lexikálnej a syntaktickej analýzy tohto výrazu sa nachádza na obrázku 1.2.



Obr. 1.2: Ukážka procesu lexikálnej a syntaktickej analýzy [8].

Ako je možné z obrázku 1.2 vidieť, lexer na začiatku prechádzania vstupného reťazca identifikoval znaky „4“, „3“ a „7“ a následne znak medzery, resp. znak typu *white space*, na základe čoho určil, že tieto znaky reprezentujú číslo. Potom identifikoval znak „+“, ktorý správne označil ako znak operátora sčítania a na záver v reťazci našiel ďalšie číslo. Po lexikálnej analýze parser vytvoril parsovaci strom, pričom tento parsovaci strom správne vyjadruje štruktúru prúdu tokenov vytvorených lexerom. Parsovaci strom teda správne určuje, že sa jedná o výraz a konkrétnie, že sa jedná o operáciu sčítanie.

Lexer a parser vykonávajú analýzu na základe vopred definovanej gramatiky, ktorá môže mať rôzne podoby a môže byť realizovaná viacerými spôsobmi. Gramatiku v kontexte parsovania stručne popíšeme v nasledujúcej podkapitole.

### 1.4.3 Formálne gramatiky

**Formálne gramatiky** sú matematické modely používané na opis syntaxe prirodzených a programovacích jazykov. V kontexte parsovania, teda syntaktickej analýzy, sú gramatiky dôležité, nakoľko vďaka gramatikám sa pri procese parsovania určuje, či daný vstup spĺňa danú gramatiku alebo nie. Rozlišujú sa rôzne typy gramatík, ktoré definujú rôzne triedy jazykov v závislosti od svojej zložitosti a obmedzení pravidiel. Tieto triedy jazykov sú organizované do hierarchie zvanej *Chomského hierarchia jazykov*, ktorá zahŕňa štyri typy gramatík: neobmedzené, kontextovo závislé, bezkontextové a regulárne [9].

Pri parsovaní sa obvykle používajú bezkontextové gramatiky. Bezkontextové gramatiky sa skladajú z konečnej množiny pravidiel, kde každé pravidlo transformuje jeden neterminál na reťazec terminálov a neterminálov. Vďaka svojej relatívnej jednoduchosti a dostatočnej výkonnosti sú tieto gramatiky ideálne pre parsovacie proces [9].

Pri procese parsovania sa napr. v rámci nástroja na generovanie parserov ANTLR (opísaný v podkapitole 1.5) definuje gramatika samostatne pre lexikálnu analýzu a pre syntaktickú analýzu. Na definovanie gramatiky existujú dva rôzne prístupy, ktoré môžeme označiť aj ako parsovacie metódy. Tieto prístupy sa zvyknú označovať ako „*zhora nadol*“ a „*zdola nahor*“.

#### Prístup zhora nadol

Prístup „*zhora nadol*“ pozostáva zo všeobecnej organizácie súboru napísaného v konkrétnom jazyku alebo formáte. Tento prístup funguje najlepšie, keď už je známy formát alebo jazyk, pre ktorú je gramatika navrhovaná. Takáto stratégia je často preferovaná jednotlivcami s vyšším teoretickým zázemím alebo jednotlivcami, ktorí dávajú prednosť začínať nejaký proces s komplexným návrhom. Prístup zhora nadol sa začína definovaním pravidla, ktoré reprezentuje celý súbor s určitým vstupom. Toto pravidlo niekedy obsahuje ďalšie pravidlá predstavujúce hlavné sekcie súboru. Potom sú pravidlá definované postupne od všeobecných abstraktných pravidiel až po pravidlá na nízkej úrovni [8].

#### Prístup zdola nahor

Druhý prístup definovania gramatiky „*zdola nahor*“ spočíva v tom, že sa najprv kladie dôraz na malé elementy. Najskôr sa definuje, ako sa zachytávajú tokeny, ako sú

definované základné výrazy a pod. Potom sa definovaním pokračuje s konštrukciami vyššej úrovne, až kým sa nedefinuje pravidlo reprezentujúce celý súbor so vstupom. Prístup **zdola nahor** umožňuje zamerať sa najskôr na malú časť gramatiky, zabezpečiť jej fungovanie podľa očakávaní, prípadne zabezpečiť pre ňu testy a až potom prejsť na ďalšiu časť gramatiky. Taktiež výhoda tohto prístupu je taká, že práca pri ňom začína s definíciou elementov, ktoré sú v mnohých aspektoch rovnaké medzi mnohými jazykmi. Veľká časť programovacích jazykov má totiž elementy ako komentáre, oddelovacie znaky, biele znaky, identifikátory a pod. [8].

Nevýhoda prístupu zdola nahor je taká, že sa najskôr pri ňom zameriava primárne na lexikálnu analýzu. Pri procese parsovania je vytvorenie parsera dôležitejšie ako definovanie lexera. Parser totiž poskytuje požadovaný výsledok parsovania, lexerom zabezpečíme iba lexikálnu analýzu. Taktiež ked' sa najskôr začína procesom lexikálnej analýzy, ktorý je vo výsledku častokrát menej požadovaný ako proces syntatickej analýzy, hrozí riziko, že gramatiku lexikálnej analýzy bude potrebné neskôr refaktorovať a upravovať podľa definovania pravidiel gramatiky pre parser alebo podľa zmien požiadaviek [8].

## 1.5 ANTLR

Na vykonanie lexikálnej a syntatickej analýzy existuje viacero prístupov a nástrojov. Definovanie vlastného parsera je jednou z možností, ako pristupovať k tomuto problému. Vytvorenie vlastného parsera je však pomerne náročný proces, ktorý pri samotnej implementácii môže v sebe skrývať veľa nezrovnalostí a chýb. Preto je nanajvýš vhodné, pokiaľ je to možné, využiť na parsovanie pomocný nástroj. V našom prípade sme sa rozhodli zvoliť si nástroj ANTLR (skratka z *ANother Tool for Language Recognition*). Názov tohto nástroja môžeme do slovenčiny preložiť ako „*Ďalší nástroj pre rozpoznávanie jazyka*“ [10, 8].

Dôvod, prečo sme sa rozhodli využiť tento nástroj, je najmä taký, že tento nástroj umožňuje jednoducho vytvárať parsery pre rôzne formáty dát. ANTLR taktiež umožňuje výsledné parsovacie stromy efektívne prechádzať, čo vedie k možnosti účinného spracovávania výsledkov parsovania. V našom prípade, ako uvádzame aj v kapitole 2, potrebujeme parsovať súbory typu JSON popisujúce sekvenčné diagramy, pričom samotná štruktúra týchto súborov je pomerne zložitá. Nástroj ANTLR sme sa rozdrobili využiť aj vzhľadom na to, že tento nástroj je jeden z najznámejších z parsovacích nástrojov. Konkrétnie v našom prípade využívame najnovšiu verziu nástroja ANTLR, *ANTLR v4* (ANTLR verzie 4) [10].

ANTLR v4 je výkonný generátor parsera (syntaktického analyzátora) ktorý je možné využiť na spracovávanie, čítanie, prekladanie alebo aj vykonávanie štruktúrova-

ného textu, či binárnych súborov. Jeho využitie je široké, v komunite vývojárov softvéru je používaný na vývoj rámcov (frameworkov), ďalších programovacích jazykov a pod. [10, 8].

Nástroj ANTLR potrebuje na vytvorenie parsera pre požadovaný vstup jazyk, resp. gramatiku. ANTLR pre tento vstup vygeneruje parser, ktorým sa vytvorí parsovaci strom. Gramatika pre nástroj ANTLR sa definuje v samostatnom súbore s príponou „*g4*“. V tomto súbore sú najskôr za sebou v poradí definované pravidlá parsera a následne pravidlá lexera. Štruktúre gramatiky pre ANTLR sa venujeme aj v kapitole 2.1.2.

Forma vstupnej gramatiky pre nástroj ANLTR je v tzv. **EBNF** (Extended Backus-Naur Form) forme [10]. **EBNF** môžeme preložiť ako „*bezkontextová gramatika v rozvinutej Backus–Naurovej forme*“.

ANTLR z gramatiky generuje parser tzv. rekurzívneho zostupu. Tento termín je možné pomenovať aj ako parsovanie **zhora nadol** [10]. Princíp parsovania zhora nadol je v zásade rovnaký, ako opísaný princíp definovania gramatiky v predošej podkapitole. Parsovanie zhora nadol spočíva v tom, že proces parsovania začína v koreni parsovacieho stromu a následne pokračuje smerom k listom tohto stromu.

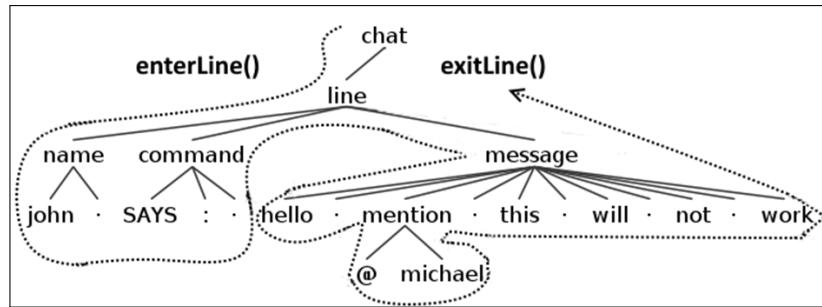
Po vygenerovaní parsovacích stromov je možné prechádzať vrcholy týchto stromov. Jednotlivé spôsoby prechádzania parsovacích stromov uvádzame v nasledujúcej podkapitole.

### 1.5.1 Spôsoby prechádzania vrcholov parsovacích stromov

Prechádzanie jednotlivých vrcholov parsovacieho stromu je v nástroji ANLTR možné adaptovať dvomi spôsobmi. Prvým z nich je prechádzanie pomocou tzv. *listenera* a druhý pomocou tzv. *visitora*.

#### Prechádzanie parsovacieho stromu listenerom

Príklad fungovania prechádzania parsovacieho stromu pomocou **listenera**, resp. podľa návrhového vzoru *Listener*, je zobrazený na obrázku 1.3 (*v preklade z angličtiny slovo „listener“ môžeme preložiť ako „poslucháč“ - v texte ďalej uvádzame ekvivalent tohto slova v angličtine*). V tomto príklade je prerusovanou čiarou zobrazené, ako je parsovaci strom v ANLTR prechádzaný vo všeobecnosti. Metóda *enterLine()* predstavuje metódu, ktorá sa listenerom zavolá, keď listener vstúpi do vrcholu s názvom *line*. Metóda *exitLine()* zas predstavuje metódu zavolanú po odídení z vrcholu *line*. Ako je možné vidieť aj na obrázku 1.3, metóda *exitLine()* je zavolaná až po tom, ako boli prejdené deti vrcholu *line*.



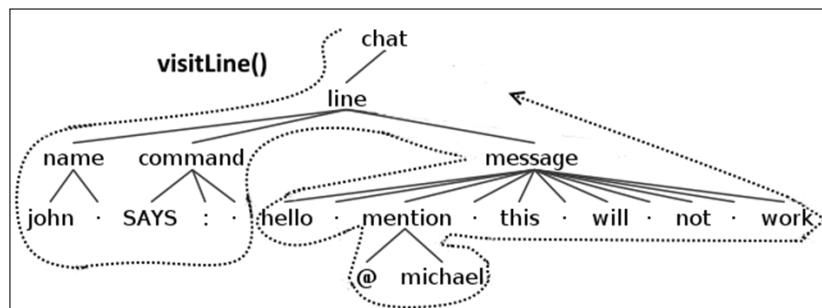
Obr. 1.3: Prechádzanie vrcholov parsovacieho stromu listenerom [8].

### Prechádzanie parsovacieho stromu visitorom

**Visitorom**, resp. prostredníctvom návrhové vzoru *Visitor*, sa parsovací strom prechádza podobne s tým rozdielom, že visitor ponúka oveľa väčšiu flexibilitu pri definovaní samotného prechádzania stromu (*slovo „visitor“ môžeme do slovenčiny z angličtiny preložiť ako „návštěvník“ - v texte ďalej uvádzame ekvivalent tohto slova v angličtine*).

Princíp prechádzania parsovacích stromov visitorom spočíva v tom, že pri navštívení vrcholov parsovacieho stromu je zavolaná príslušná udalosť, resp. metóda pre každý z vrcholov. Konkrétnie má táto metóda tvar *visitX()* kde *X* predstavuje názov daného vrcholu. Avšak prechádzaním stromu pomocou visitoru je možné prepísať metódy *visit()* vygenerované visitorom a prechádzanie stromu je možné prispôsobiť podľa vlastných potrieb. Na obrázku 1.4 je zobrazený príklad prechádzania stromu pomocou visitoru.

V obrázku nie sú zobrazené príslušné metódy *visit* pre každý z vrcholov parsovacieho stromu, je v nej zobrazená iba metóda *visitLine()*, ktorá je zavolaná pri vstupe do vrcholu *line*. Metódu *visitLine()* by bolo možné prepísať napríklad tak, aby sa pri jej zavolaní pri procese prechádzania parsovacieho stromu už ďalej nevchádzalo do detí vrcholu *line*, čo je príklad spomínanej flexibility pri prechádzaní parsovacích stromov.



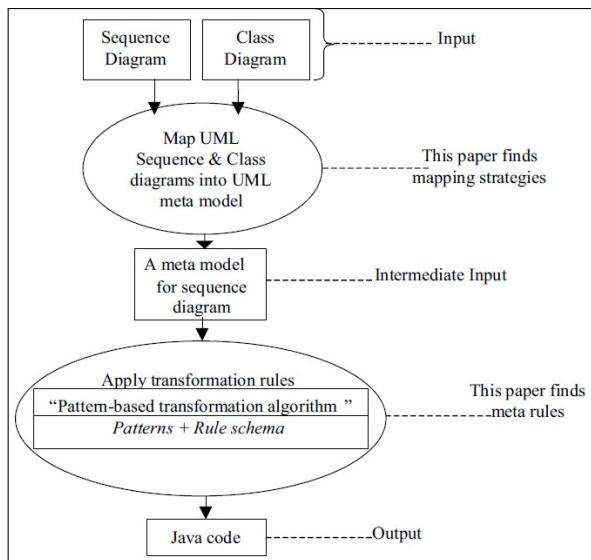
Obr. 1.4: Prechádzanie vrcholov parsovacieho stromu visitorom [8].

## 1.6 Prehľad spôsobov transformovania dynamických modelov do zdrojového kódu

Ako už bolo spomínané, jedným z hlavných cieľov našej práce je transformovať dynamické procesy modelov, znázornených sekvenčným diagramom, do zdrojového kódu, a to konkrétnie do kódu v jazyku *OAL*. V tejto kapitole sa zameriavame na prehľad niektorých publikácií, ktoré mali podobný cieľ. Tieto publikácie opisujú konverziu dynamických modelov do rôznych cieľových jazykov či platforem.

Jednou z takýchto publikácií je článok *Design of Rules for Transforming Sequence Diagrams into Java Code* [11]. Ako aj hovorí názov, tento článok je o generovaní sekvenčných diagramov do kódu v programovacom jazyku Java. Navrhovaný transformačný prístup v tejto publikácii pozostáva z metamodelu pre sekvenčné diagramy a z tzv. transformačných pravidiel. V článku sa tiež vysvetľuje, že ich transformačný algoritmus pre sekvenčné diagramy je založený na vzoroch (patterns), čo znamená, že sa najprv špecifikujú vzory, ktoré sa následne transformujú na schémy pravidiel. Teda transformácia zo sekvenčného diagramu do kódu v programovacom jazyku Java opísaná v článku prebieha mapovaním sekvenčných diagramov na metamodely a aplikovaním transformačných pravidiel, pričom tieto pravidlá sú založené na už spomenutých vzoroch. Navyše, transformácia predstavená v článku dokáže konvertovať do zdrojového kódu v jazyku Java nielen sekvenčné diagramy, ale aj diagramy tried. Spoločné s našou prácou má tento článok napríklad to, že využíva pravidlá (v našom prípade gramatiku) na dosiahnutie konverzie do kódu [11]. Prístup k transformácii sekvenčných diagramov a diagramov tried predstavený v publikácii [11] je znázornený na obrázku 1.5.

Ďalší, v istom zmysle podobný článok, je článok s názvom *Mapping UML Sequence Diagram into the Web Ontology Language* [12]. Tento článok je podobný ako predchádzajúci spomenutý článok, pričom tému tohto článku je mapovanie sekvenčných diagramov do jazyka *OWL*, čo je formálny jazyk na reprezentáciu znalostí a informácií o určitej oblasti v štruktúrovanom a strojovo čitateľnom formáte. Jazyk *OWL*, uľahčuje strojom interpretáciu a spracovanie informácií. V tomto článku sú do hĺbky porovnané rozdiely medzi sekvenčnými diagramami a *OWL*. Na základe týchto rozdielov boli v článku definované pravidlá na konvertovanie sekvenčných diagramov do jazyka *OWL*. Tento článok teda predstavuje formálne mapovanie aspektov sekvenčného diagramu do odlišného jazyka, ktorým je *OWL*. V článku bolo riešenie demonštrované aj na príklade dynamického modelu bankomatu [12].



Obr. 1.5: Transformačný prístup predstavený v článku *Design of Rules for Transforming Sequence Diagrams into Java Code* [11].

Iný, zaujímavý článok, je článok s názvom ***Novel approach to transform UML Sequence Diagram to Activity*** [13]. Tento článok sa týka konvertovania sekvenčných diagramov do diagramov aktivít, ktoré viac pomáhajú používateľom analyzovať prechody aktivít v rámci dynamického správania softvéru. V tomto článku je sekvenčný diagram konvertovaný do tzv. sekvenčnej tabuľky, v ktorej sú stĺpce rozdelené na odošielateľa, príjemcu a správu napísanú v prvoradovej logike. Táto tabuľka je následne transformovaná do tzv. „activity“ tabuľky, obsahujúcej príslušné toky údajov a posielané správy. „Activity“ tabuľka je na záver konvertovaná do diagramu aktivít. V článku sa dopodrobna rozoberá napr. konverzia asynchronných správ a fragmentov typu ALT a LOOP sekvenčného diagramu do „activity“ tabuľky. Riešenie v článku bolo demonštrované na jednoduchej aplikácii na prihlásenie používateľa [13].

Článok ***Automatic Test Case Generation Using Sequence Diagram*** sa týka generovania testovacích prípadov (test cases), pomocou sekvenčných diagramov [14]. Predstavený algoritmus konverzie je odlišný oproti ostatným, ktoré boli opísané v predošlých spomínaných článkoch. Algoritmus v článku zahŕňa konštrukciu sekvenčného diagramu a jeho konverziu na tzv. sekvenčný graf. Potom algoritmus zahŕňa výber predikátových funkcií, transformáciu predikátov do zdrojového kódu, následne konštrukciu tzv. rozšíreného konečného stavového stroja (*EFSM*) a generovanie testovacích údajov zodpovedajúcich transformovaným predikátovým funkciám. Hlavnou myšlienkovou spomienkou v tomto článku bolo získať informácie zo sekvenčného diagramu a previesť ich do zdrojového kódu jazyka Java, konkrétnie do knižnice s názvom *ModelJUnit*. Riešenie v článku bolo demonštrované na príklade modelu bankomatu [14].

# Kapitola 2

## Návrh riešenia generovania zdrojového kódu z dynamického modelu

Na dosiahnutie nášho riešenia bolo potrebné analyzovať rôzne prístupy a technológie, ako aj navrhnutý postup k dosiahnutiu požadovaného výsledku. V tejto kapitole uvádzame, ktoré technológie, resp. programy sme využili a ako sme ich aplikovali do nášho procesu. Uvádzame aj samotný návrh činností a častí nášho riešenia, potrebných k dosiahnutiu cieľa našej práce, a to od fundamentálnych krokov až po činnosti, ktoré priamo nesúvisia s našou prácou.

### 2.1 Proces konverzie sekvenčných diagramov do zdrojového kódu

Proces konverzie sekvenčných diagramov do zdrojového kódu vyžaduje niekoľko krokov k dosiahnutiu nami požadovaného riešenia. V tejto podkapitole prinášame opis celého procesu generovania OAL kódu zo sekvenčných diagramov, vrátane jeho použitia v prototype *AnimArch*.

#### 2.1.1 Vytvorenie sekvenčného diagramu

Ako prvý krok procesu uvádzame vytvorenie samotného sekvenčného diagramu. Hoci táto časť procesu generovania OAL kódu sa priamo netýka nášho riešenia, domnievame sa, že ju je potrebné uviesť ako začiatočný krok procesu generovania OAL kódu.

Sekvenčné diagramy je možné vytvárať v rozličných softvérových produktoch dostupných na trhu. V našom prípade však využívame projekt s názvom *SQD\_Tunder*. Jedná sa o jednoduchý softvérový nástroj, resp. aplikáciu na vytváranie sekvenčných diagramov. Tento nástroj bol vytvorený tímovým projektom *UMLions*. Vznikol pod vedením doc. Poláška na FIIT STU v Bratislave a dostal názov *3D UML*. My využí-

vame jeho najnovšiu verziu, ktorá nesie názov *SQD\_Tunder*. Ukážka tohto nástroja, resp. prototypu sa nachádza na obrázku 2.1.

### 2.1.1.1 SQD\_Tunder

Program **SQD\_Tunder** umožňuje vytvárať sekvenčné diagramy s rôznymi typmi fragmentov. Programom je taktiež používateľovi umožnené jednotlivé sekvenčné diagramy ukladať do úložiska počítača, ako ich aj z úložiska načítavať a ďalej upravovať. Sekvenčné diagramy sú ukladané do súboru vo formáte JSON. Zaujímavostou tejto aplikácie je možnosť vytvárania a upravovania sekvenčných diagramov v 3D prostredí. V rámci diplomovej práce študenta M. Franka bola aplikácia naposledy upravovaná za účelom pridania interakcie s jej funkcionálitami pomocou virtuálnej reality [15]. Aplikácia obsahuje niekoľko drobných chýb a absentuje v nej napríklad možnosť definovania aplikačných blokov na lifeline (v preklade „životná čiara“, alebo „ťažnica života“) sekvenčného diagramu.

**(Poznámka:** kvôli ťažkopádnemu prekladu slova „lifeline“ do slovenčiny budeme ďalej v texte práce používať jeho cudzojazyčný ekvivalent, a teda namiesto slova „životná čiara“ či „ťažnica života“ budeme používať slovo „lifeline“.)

Dôvodov, prečo na vytváranie sekvenčných diagramov využívame túto aplikáciu a nie napríklad nejaký komerčný produkt na modelovanie (UML) diagramov, je viacero. Ako prvý dôvod môžeme uviesť fakt, že projekt *SQD\_Tunder* je Open-source aplikáciou, teda aplikáciou s voľným prístupom, čo znamená, že máme priamy prístup k jej zdrojovému kódu a je taktiež dostupná zadarmo. Druhým a pre nás zásadným dôvodom je jej funkcionálita ukladania a prípadného upravovania vytvorených sekvenčných diagramov. Ako sme spomenuli, aplikácia ukladá používateľom vytvorené sekvenčné diagramy do formátu JSON. Štruktúra ukladaných JSON súborov je zreteľne definovaná, jasne popisuje dané elementy sekvenčného diagramu. Dôležitosť takejto na prvý pohľad nie veľmi významnej funkcionality spočíva v tom, že práve tieto súbory využívame v ďalších krokoch procesu generovania OAL kódu. Štruktúra uložených sekvenčných diagramov vytvorených nástrojom *SQD\_Tunder* je bližšie opísaná v podkapitole 2.1.2.1.

Ako ďalší a posledný dôvod, prečo využívame aplikáciu *SQD\_Tunder*, predstavuje fakt, že bola vyvíjaná v nástroji Unity a v programovacom jazyku C#, rovnako ako aplikácia *AnimArch*. Tento dôvod nie je pre naše riešenie veľmi významný, avšak znamená výhodu v tom, že tento nástroj môže byť spolu s našou implementáciou potenciálne v budúcnosti jednoduchšie využiteľný v rámci vývojového tímu na FMFI UK pod vedením doc. Poláška.

Obr. 2.1: Ukážka prototypu *SQD\_Tunder*.

### 2.1.2 Parsovanie sekvenčných diagramov

Ďalší krok v našom postupe generovania OAL kódu predstavuje syntaktická analýza, resp. parsovanie. Presnejšie povedané, v tomto kroku sa jedná o syntaktickú analýzu súborov definujúcich sekvenčné diagramy, o ktorých sme hovorili v predošej podkapitole. Syntaktická analýza predstavuje v priebehu generovania OAL kódu jeden z najdôležitejších procesov. Procesom syntaktickej analýzy získavame z textového súboru vo formáte JSON definujúceho daný sekvenčný diagram funkčné elementy kódu, resp. objekty tried, z ktorých ďalej vieme definovať OAL kód.

Na syntaktickú analýzu využívame nástroj na generovanie parserov ANTLR, bližšie opísaný v kapitole 1.5. Konkrétnie v rámci využitia nástroja ANTLR dochádza k fázam, ktoré sme opísali v kapitole 1.3. Tieto fázy tu v kontexte nášho riešenia aspoň v krátkosti uvedieme:

#### 1. lexikálna analýza

Prvou fázou parsovania s využitím nástroja ANTLR je generovanie lexera (lexikálneho analyzátora), resp. lexikálna analýza. ANTLR generuje lexer na základe pravidiel uvedených v našej gramatike. Týmto krokom sa uskutoční lexikálna analýza, kde sa náš vstupný súbor rozdelí na postupnosť tokenov, teda najmenšie významné jednotky, ktoré sú pre nás dôležité. O jednotlivých pravidlach lexera našej gramatiky píšeme v kapitole 2.1.2.2.

#### 2. syntaktická analýza

Druhá fáza parsovania je syntaktická analýza, resp. generovanie parsera, teda samotné parsovanie ako také. Na základe pravidiel parsera v našej gramatike, ktorou definujeme syntaktickú štruktúru významných elementov súboru popisu-júceho sekvenčný diagram, generovaný parser prevezme prúd tokenov vytvorených lexerom a vytvorí parsovací strom. Nami definovaná gramatika pre parser je opísaná v kapitole 2.1.2.3.

### 3. sémantická analýza

Ďalšou fázou pri parsovaní je sémantická analýza. Táto analýza nie je realizovaná priamo ako súčasť parsera, avšak v našom prípade nástroj ANLTR generuje tzv. *visitor*, ktorý prechádza cez vrcholy vygenerovaného parsovacieho stromu. Tento *visitor* sme upravili podľa našich potrieb tak, aby prechádzal a zbieran informácie potrebné na generovanie OAL kódu. O prechodech parsovacích stromov, a teda aj o *visitore* sme písali v kapitole 1.5.1.

### 4. Integrácia s našim riešením

Poslednou fázou parsovania sme nazvali ako „*integráciu s našim riešením*“. Keď nami upravený visitor prejde všetky vrcholy parsovacieho stromu, môžu byť v našom riešení ďalej uskutočnené volania, ktoré spustia samotné generovanie OAL kódu.

V tejto kapitole následne bližšie uvedieme spôsob, akým sú parsované súbory, ktoré sú generované nástrojom *SQD\_Tunder* a ktoré definujú sekvenčné diagramy. Najskôr popíšeme štruktúru týchto súborov a neskôr popíšeme pravidlá gramatiky, vďaka ktorým generátor parserov ANTLR vykonáva syntaktickú analýzu na týchto súboroch.

#### 2.1.2.1 Analýza súborov definujúcich sekvenčné diagramy

Ako bolo uvedené aj v kapitole 2.1.1, nástroj *SQD\_Tunder* ukladá sekvenčné diagramy do súborov vo formáte JSON. Tieto súbory majú pre jednotlivé elementy sekvenčného diagramu presnú štruktúru. Každý element sekvenčného diagramu má svoj vlastný jedinečný identifikátor označený ako *XmiId*. Jednotlivé elementy majú definovaný aj svoj typ, ktorý je označený atribútom *XmiType*. Elementy sa v diagrame na seba návzájom odkazujú, čo je stanovené aj v JSON súbore uloženého sekvenčného diagramu. V štruktúre formátu JSON je vzájomné prepojenie elementov definované cez atribút *XmiRef*. Hodnota atribútu *XmiRef* predstavuje hodnotu *XmiId*, teda hodnotu unikátneho identifikátora iného elementu.

Čo sa týka jednotlivých typov elementov, ktoré vygenerovaný súbor so sekvenčným diagramom obsahuje, tak najzákladnejšie typy pre nás sú:

## Lifeline

Tento typ reprezentuje element *lifeline* zo sekvenčného diagramu. V JSON štruktúre súboru je teda spolu s atribútom a hodnotu tento typ označený ako "*XmiType*": "*uml:Lifeline*". Okrem spomenutých atribútov *XmiId* a *XmiType* obsahuje atribút s názvom *name*, pričom hodnota tohto atribútu predstavuje názov daného lifeline. Je dôležité poznamenať, že element lifeline sekvenčného diagramu prekladáme do OAL kódu ako triedu, teda typ *class*.

## Message

Typ *Message* v spomínanom súbore predstavuje element typu message, teda správu medzi elementami lifeline v sekvenčnom diagrame. Tento typ je v súbore označený ako "*XmiType*": "*uml:Message*". Okrem uvedených atribútov *XmiId* a *XmiType* tento typ zahŕňa aj niekoľko ďalších a pre nás podstatných atribútov. Tieto atribúty sú:

- **receiveEvent:** atribút, ktorý označuje, do ktorého elementu daná správa smeruje. Aby sme to uviedli presnejšie, atribút označuje, do akého *lifeline* sa správa posielajúca. Tento atribút je z hľadiska štruktúry formátu JSON objekt, ktorý obsahuje vyššie spomenutý atribút *XmiRef*. V atribúte *XmiRef* je v tomto prípade obsiahnutá hodnota identifikátora (typu *XmiId*) objektu typu *OccurrenceSpecification*, ktorý opisujeme nižšie.
- **sendEvent:** atribút *sendEvent* je podobný ako atribút *receiveEvent*, až na to, že označuje, z akého elementu je dané volanie odoslané. Rovnako, ako v prípade *receiveEvent*, aj tento atribút obsahuje atribút *XmiRef*, ktorý odkazuje na prvok typu *OccurrenceSpecification* (opísaný nižšie).
- **name:** tento atribút je označením názvu správy.

Typ *Message* (teda v sekvenčnom diagrame správa) je do OAL kódu prekladaný ako volanie metódy. Je podstatné podotknúť, že vďaka tomuto elementu vieme identifikovať samotné metódy daných tried. A teda, okrem volania metódy, je správa preložená do OAL kódu aj ako metóda príslušnej triedy. Trieda, do ktorej sa takto identifikovaná metóda pridá, je trieda (resp. lifeline), do ktorej správa smeruje. Viac o konverzii sekvenčných diagramoch do zdrojového kódu píšeme v kapitole 3.

## OccurrenceSpecification

*OccurrenceSpecification* je špeciálny element, vďaka ktorému vieme určiť, do ktorého konkrétneho prvku lifeline sa daná správa posielajúca, resp. z akého elementu lifeline sa správa odosiela. Popri všeobecných atribútoch *XmiId* a *XmiType* element *OccurrenceSpecification* obsahuje pre nás podstatný atribút s názvom *covered*. Atribút *covered* je na základe syntaxe formátu JSON objektom, zahrňujúcim atribút *XmiRef*,

ktorý zas obsahuje referenciu na danú lifeline, teda obsahuje hodnotu identifikátora *XmiId* konkrétnej lifeline sekvenčného diagramu. V súbore je typ *OccurrenceSpecification* označený ako "*XmiType*": "*uml:OccurrenceSpecification*".

### CombinedFragment

Ako hovorí aj názov elementu typu *CombinedFragment*, jedná sa o prvok, ktorý predstavuje fragment v sekvenčnom diagrame. V JSON súbore definujúcim sekvenčný diagram je identifikovateľný ako "*XmiType*": "*uml:CombinedFragment*". Okrem atribútov *XmiId* a *XmiType*, ktoré sú obsiahnuté vo všetkých typoch elementov sekvenčného diagramu nachádzajúcich sa v spomenutom JSON súbore, obsahuje nasledovné podstatné atribúty:

- **interactionOperator:** atribút určujúci, o aký typ fragmentu sa jedná. Hodnota tohto atribútu je celé kladné číslo. Konkrétnie atribút *interactionOperator* nado búda hodnoty od čísla 1 po číslo 12 podľa daného typu fragmentu. Prislúchajúce hodnoty typov fragmentu sú nasledovné: 1: *SEQ*, 2: *ALT*, 3: *OPT*, 4: *BREAK*, 5: *PAR*, 6: *STRICT*, 7: *LOOP*, 8: *CRITI*, 9: *NEG*, 10: *ASSE*, 11: *IGNO*, 12: *CONS*. Pri generovaní sekvenčných diagramov do zdrojového kódu využívame iba nasledujúce hodnoty tohto atribútu: 2: *ALT*, 3: *OPT*, 5: *PAR* a 7: *LOOP*.
- **operand:** atribút *operand* je podľa syntaxe formátu JSON atribútom typu zoznam. Tento atribút obsahuje referencie na dátu ako na podmienky v rámci fragmentu (resp. na elementy typu *InteractionOperand*).

### InteractionOperand

Element, označený v JSON súbore ako "*XmiType*": "*uml:InteractionOperand*", obsahuje okrem atribútov, ktorého ho identifikujú a špecifikujú (*XmiId* a *XmiType*) dva dôležité atribúty:

- **guard:** Týmto atribútom sa element *InteractionOperand* odkazuje na element typu *InteractionConstraint* (cez identifikátor elementu *InteractionConstraint*).
- **ownedElement:** atribút *ownedElement* je atribútom typu zoznam. Tento atribút je pre celkovú identifikáciu pozície fragmentu v rámci sekvenčného diagramu zásadný, nakoľko zahrňa jednotlivé dátu o fragmente. Konkrétnie obsahuje atribúty *XmiRef*, ktorých hodnoty sa odkazujú na ďalšie typy elementov v sekvenčnom diagrame. Zjednodušene povedané, tento element obsahuje referencie na ďalšie fragmenty v prípade vnorených fragmentov, teda na typy *InteractionOperand* a na správy (resp. na typ *OccurrenceSpecification*).

### **InteractionConstraint**

Element **InteractionConstraint** má okrem všeobecných atribútov iba jeden dôležitý atribút, ktorým je atribút pomenovaný ako *specification*. Hodnota atribútu *specification* je referenciu na prvok typu **OpaqueExpression**, teda *specification* obsahuje hodnotu atribútu *XmiId* elementu **OpaqueExpression**.

### **OpaqueExpression**

V prípade elementu s názvom **OpaqueExpression** sa jedná o prvok, vďaka ktorému vieme určiť konkrétnu podmienku daného fragmentu. Táto podmienka sa v rámci tohto typu elementu nachádza ako hodnota atribútu *body*, pričom sa v ňom nachádza ako text (vo formáte JSON typ *string*).

Aby bolo teda možné zistiť hodnotu podmienky fragmentu, je potrebné zistiť referenciu na typ **InteractionOperand** cez hodnotu v atribúte *operand* elementu **CombinedFragment**. Následne sa je potrebné odkázať na konkrétny element typu **InteractionConstraint** cez hodnotu *guard* elementu **InteractionOperand**. Na záver cez hodnotu atribútu *specification* elementu **InteractionConstraint** získavame referenciu na element **OpaqueExpression**, ktorý v spomenutom atribúte *body* obsahuje hľadanú hodnotu podmienky fragmentu.

#### 2.1.2.2 Pravidlá lexera

Pravidlá lexera, ktorými sa nad vstupným JSON súborom realizuje lexikálna analýza, sa nachádzajú na obrázku 2.2. Týmito pravidlami sa teda pre daný JSON súbor vytvorí lexer a znaky súboru sa zoskupia do zmysluplných elementov, teda tokenov.

Väčšina z pravidiel nachádzajúcich sa v pravidlach lexera sú tzv. fragmenty. Fragmenty v kontexte gramatiky predstavujú opakovateľné stavebné bloky pre jednotlivé pravidlá lexikálnej analýzy. Jednoduchšie povedané, jedná sa o pravidlá, ktoré môžu byť použité v iných pravidlach. Podotýkame, že dané lexikálne pravidlá nie sú špecifické pre naše potreby, ale sú všeobecne definované pre štruktúru formátu JSON.

**Jedná sa o tieto pravidlá:**

- **fragment HEX:** začnime s jednoduchým pravidlom, resp. s fragmentom *HEX*. Časť pravidla, resp. regulárny výraz *[0-9a-fA-F]* definuje, ktoré znaky sú v rámci fragmentu *HEX* povolené. V prípade pravidlá *HEX* sa jedná o hexadeximálne číslice.
- **fragment UNICODE:** pravidlo *Unicode* definuje, ako má byť na vstupe reprezentovaný znak v štandarde Unicode. Pravidlo je definované s požitím fragmentu *HEX*.

- **fragment *ESC*:** tento fragment je definíciou pre tzv. *escape* sekvenciu. Znak dvoch lomiek „||“ značí začiatok *escape* sekvencie, ktorá začína znakom „|“, Znak „|“ má v gramatike samostatný význam, preto na špecifikáciu samotného znaku lomky v pravidle je potrebné na začiatok pridať znak „|“. Časť v pravidle ( $/"/| | /bfnrt/$  / UNICODE) definuje dva scenáre, ktoré môžu po znaku „\“ nastaviť. Prvá časť  $/"/| | /bfnrt/$  znamená, že po lomke môže nasledovať jeden zo znakov „"“, „|“, „/“, „b“, „f“, „n“, „r“, „t“, ktoré predstavujú všeobecne známe *escape* znaky (napr. „\n“ pre nový riadok). Druhá časť značí, že namiesto štandardného *escape* znaku môže nasledovať jeden zo znakov v rámci štandardu Unicode.
- **fragment *SAFECODEPOINT*:** fragment *SAFECODEPOINT* definuje, že akceptované sú všetky znaky, okrem dvojitych úvodzoviek, späťnej lomky a kontrolných znakov v rozsahu od  $|u0000$  až po  $|u001F$  (teda od 0 po 31). Výraz „~“ značí negáciu a regulárny výraz „ $/"/| |u0000-|u001F/$ “ znaky „"“, „|“ a rozsah od  $|u0000$  po  $|u001F$ . V Unicode štardarde sú znaky kódované od „ $|u0000$ “ až „ $|u001F$ “ rezervované pre tzv. riadiace kódy. Dá sa povedať, že toto pravidlo vymedzuje znaky, ktoré pre reťazce znakov formátu JSON nie sú „bezpečné“.
- ***STRING*:** pravidlom *STRING* sú definované reťazce znakov vo formáte JSON. Správny reťazec znakov podľa tohto pravidla obsahuje na začiatku aj na konci znak „"“ a ľubovoľný počet výskytov znakov (vrátane žiadneho) definovaných vo fragmentoch *ESC* a *SAFECODEPOINT*. Kardinalita množiny znakov definovaných v *ESC* a *SAFECODEPOINT*, ktorá sa môže rovnať nule, je v pravidle vyjadrená znakom „\*“.
- **fragment *INT*:** týmto fragmentom sú vymedzené v lexikálnej gramatike kladné celé čísla, vrátane nuly. Podľa tohto pravidla celé číslo môže byť jednociferné číslo 0 alebo ľubovoľné číslo väčšie ako nula.
- **fragment *EXP*:** fragmentom s názvom *EXP* je v gramatike definovaný exponent použitý pri zápisе vedeckých čísel.  $[Ee]$  v pravidle predstavuje písmená „E“ alebo „e“ označujúce exponent,  $/+|-/?$  predstavuje použitie jedného zo znakov „+“ a „-“ a  $/0-9/$  použitie aspoň jednej cifry od 0 po 9.
- ***NUMBER*:** pravidlo *NUMBER* kombinuje pravidlá *INT* a *EXP* a definuje ľubovoľné číslo, ktoré vďaka znaku „-“ môže predstavovať záporné číslo, vďaka znaku „.“ zas desatinné číslo a znakmi definovanými fragmentom *EXP* môže predstavovať číslo vyjadrené vedeckým zápisom.
- ***WS*:** pravidlo, ktoré zabezpečuje preskočenie tzv. bielych znakov (whitespaces) počas lexikálnej analýzy. Vďaka tomuto pravidlu sú tak lexerom preskočené znaky

medzery, tabulátora (znak „|t“) a znaky nových riadkov („|r“ a „|n“). Vynechanie týchto znakov je v pravidle zabezpečené znakom pomocou konštrukcie „-> skip“.

### 2.1.2.3 Pravidlá parsera

Gramatické pravidlá, na základe ktorých je generovaný parser a vytvorený parsovaci strom, sa nachádzajú na obrázkoch 2.3, 2.4 a 2.5. Prvé pravidlá definované v tejto časti gramatiky sú všeobecne platné pre formát súborov typu JSON. Ostatné pravidlá boli definované vzhľadom na naše potreby, resp. vzhľadom na štruktúru súborov popisujúcich uložené sekvenčné diagramy. Štruktúre týchto súborov sme sa bližšie venovali v kapitole 2.1.2.1.

**Jednotlivé pravidlá parsera sú tieto:**

- **json:** jedná sa o pravidlo na najvrchnejšej úrovni hierarchie procesu parsovania, teda o pravidlo, ktorého hodnota sa nachádza v koreni výsledného parsovacieho stromu. Toto pravidlo určuje, že platný vstup pozostáva z jednej hodnoty definovej v pravidle *value*, za ktorou nasleduje koniec súboru, označený ako *EOF*.
- **obj:** toto pravidlo popisuje štruktúru objektu, ktorý obsahuje dátu typu kľúč-hodnota (definované v pravidle *pair*), alebo neobsahuje žiadne dátu (teda je to prázdný objekt). Prázdný objekt vo formáte JSON vyzerá ako „{}“, čo je možné vidieť v definícii tohto pravidla.
- **arr:** pravidlo *arr* je pravidlo definujúce pole v JSON formáte. Pole môže byť prázdnne, čo je zobrazené ako „[]“. Ak pole nie je prázdnne, tak obsahuje viac hodnôt definovaných v parsovacom pravidle *value*.
- **value:** pravidlo *value* reprezentuje všetky základné hodnoty formátu JSON. Tieto hodnoty môžu byť reťazec, číslo (obidve definované v pravidlách lexera ako *STRING* a *NUMBER*), objekt (pravidlo *obj*), pole (pravidlo *arr*) a literály „true“, „false“ a „null“.
- **pair:** toto pravidlo reprezentuje hodnotu typu kľúč-hodnota. Kľúč a hodnota sú vo formáte JSON oddelené dvojbodkou, čo toto pravidlo zohľadňuje. Kľúčom je vždy reťazec definovaný v pravidle lexera *STRING*. Hodnota môže byť reprezentovaná v rámci gramatického pravidla *value* alebo môže byť reprezentovaná jedným zo špecifických pravidiel súvisiacich so štruktúrou súborov popisujúcich sekvenčné diagramy.
- **Konkrétna pravidlá pre sekvenčné diagramy:** všetky ostatné pravidlá obsiahnuté v časti gramatických pravidiel parsera reprezentujú buď definície typov elementov sekvenčného diagramu, alebo definície podstatných atribútov týchto

elementov. V kapitole 2.1.2.1 sa nachádzajú bližšie informácie týkajúce sa jednotlivých typov elementov a atribútov, vyjadrených týmto gramatickými pravidlami.

```

STRING
: '"' (ESC | SAFECODEPOINT)* '"'
;

fragment ESC
: '\\' (["\\/\bf\nrt"] | UNICODE)
;

fragment UNICODE
: 'u' HEX HEX HEX HEX
;

fragment HEX
: [0-9a-fA-F]
;

fragment SAFECODEPOINT
: ~ [\u0000-\u001F]
;

NUMBER
: '-'? INT ('.' [0-9] +)? EXP?
;

fragment INT
: '0' | [1-9] [0-9]*
;

fragment EXP
: [Ee] [+|-]? [0-9]+
;

WS
: [ \t\r\n\f] + -> skip
;

```

```

JSON
: value EOF
;

obj
: '{' pair (',' pair)* '}'
| '{}'
;

pair
: STRING ':' value
| name
| lifeline
| xmiId
| xmiIdRef
| message
| occurrenceSpecification
| covered
| receiveEvent
| sendEvent
| combinedFragment
| interactionOperator
| interactionOperand
| guard
| interactionConstraint
| specification
| opaqueExpression
| operand
| fragments
| body
| ownedElements
;

arr
: '[' value (',' value)* ']'
| '[]'
;

```

Obr. 2.2: Pravidlá lexera našej gramatiky.

Obr. 2.3: Pravidlá parsova - 1. časť.

### 2.1.3 Konverzia sekvenčných diagramov do zdrojového kódu

Vďaka viacerým spomínaným prínosným vlastnostiam nástroja ANTLR nám tento nástroj pri parsovaní súborov definujúcich sekvenčné diagramy umožnil efektívne obiahnuť jednotlivé prvky sekvenčných diagramov. Kvôli pomerne veľkej komplexnosti týchto súborov bol však na samotnú konverziu týchto diagramov do zdrojového kódu ANTLR nepostačujúci. Na konverziu do OAL kódu sme museli definovať celú skupinu

```

value
: STRING
| NUMBER
| obj
| arr
| 'true'
| 'false'
| 'null'
;

name
: '"name"' ':' value
;

xmiId
: '"XmiId"' ':' value
;

xmiIdRef
: '"XmiIdRef"' ':' value
;

lifeline
: STRING ':' '"uml:Lifeline"'
;

message
: STRING ':' '"uml:Message"'
;

occurrenceSpecification
: STRING ':' '"uml:OccurrenceSpecification"'
;

combinedFragment
: STRING ':' '"uml:CombinedFragment"'
;

interactionOperand
: STRING ':' '"uml:InteractionOperand"'
;

```

```

guard
: '"guard"' ':' value
;

specification
: '"specification"' ':' value
;

interactionConstraint
: STRING ':' '"uml:InteractionConstraint"'
;

opaqueExpression
: STRING ':' '"uml:OpaqueExpression"'
;

covered
: '"covered"' ':' value
;

receiveEvent
: '"receiveEvent"' ':' value
;

sendEvent
: '"sendEvent"' ':' value
;

body
: '"body"' ':' value
;

interactionOperator
: '"interactionOperator"' ':' value
;

operand
: '"operand"' ':' value
;

ownedElements
: '"ownedElement"' ':' value
;

```

Obr. 2.4: Pravidlá parsera - 2. časť.

Obr. 2.5: Pravidlá parsera - 3. časť.

tried, prostredníctvom ktorých sme na základe výsledkov z parsovania realizovali transformáciu sekvenčných diagramov do zdrojového kódu. Triedy a ich metódy umožňujúce takúto transformáciu sme definovali v programovacom jazyku C#.

V jazyku C# sme v skutočnosti realizovali aj definovanie a správu nástroja ANTLR. Parsovanie a následná konverzia sekvenčných diagramov sú spolu zoskupené v jednom C# programe. Výsledky parsovania sú tak priamo dostupné pre nami definované triedne objekty, realizujúce transformovanie do OAL kódu.

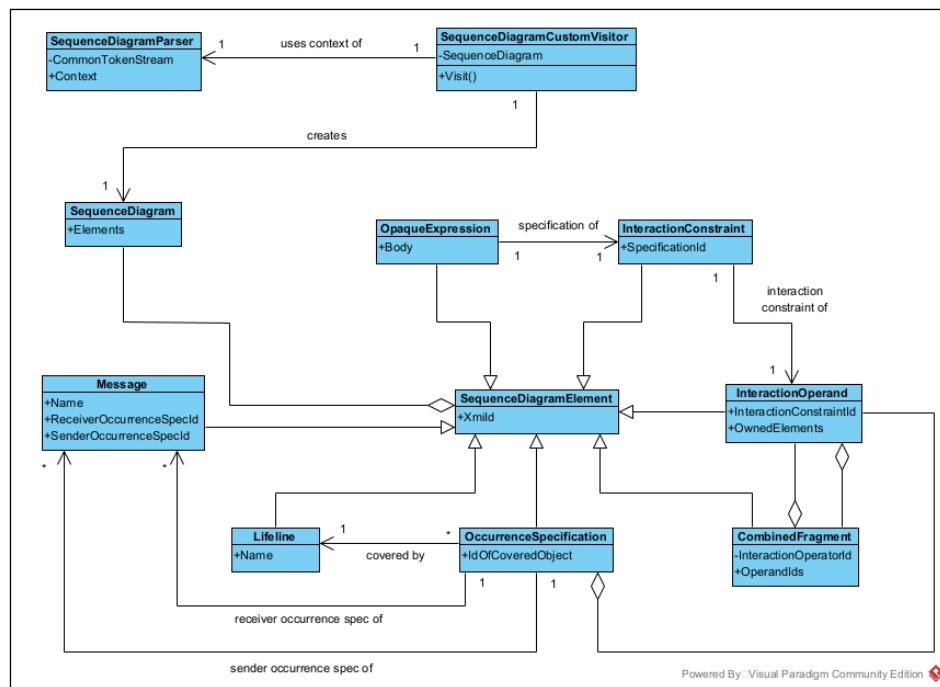
Dôvod, prečo sme sa rozhodli konverziu sekvenčných diagramov, a teda aj využitie nástroja ANTLR realizovať práve v jazyku C#, súvisí s faktom, že zmienené nástroje *AnimArch* a *SQD\_Tunder* sú taktiež implementované v tomto jazyku. Realizovaním nášho riešenia v jazyku C# sme pôvodne chceli dosiahnuť potenciálne využitie niektorých aspektov implementovaných v programoch a *AnimArch* a *SQD\_Tunder*. Predovšetkým sme však chceli dosiahnuť potenciálne znovupoužitie a samotné využitie našej

implementácie v týchto programoch v budúcnosti, či už pre výskumné alebo osobné účely v akademickom prostredí.

V nasledujúcich podkapitolách priblížime, ako sme implementovali konverziu sekvenčných diagramov do OAL kódu.

### 2.1.3.1 Zoskupenie elementov sekvenčného diagramu do objektov

Procesom parsovania sú identifikované potrebné elementy sekvenčného diagramu, definované v konkrétnych súboroch formátu JSON. Atribúty elementov diagramu sú počas prechádzania parsovacieho stromu nástrojom ANLTR použité pri definovaní objektov tried, ktoré vyjadrujú tieto elementy sekvenčných diagramov. Diagram tried, ktorý znázorňuje spomenuté triedy vyjadrujúce časti sekvenčných diagramov, je zobrazený na obrázku 2.6.



Obr. 2.6: Diagram tried s objektmi vyjadrujúcimi elementy sekvenčného diagramu.

Ako je možné z diagramu na obrázku 2.6 vidieť, trieda s názvom `SequenceDiagramCustomVisitor`, ktorá vystihuje náš upravený a adaptovaný visitor v generátorovi parserov ANLTR, využíva tzv. „context“ triedy `SequenceDiagramParser`. `SequenceDiagramParser` zas v našom riešení vyjadruje samotný parser. `SequenceDiagramParser` má okrem parametra `Context` aj parameter `CommonTokenStream` predstavujúci prúd tokenov získaných lexerom, pričom lexer v tomto diagrame kvôli lepšej prehľadnosti nezobrazujeme.

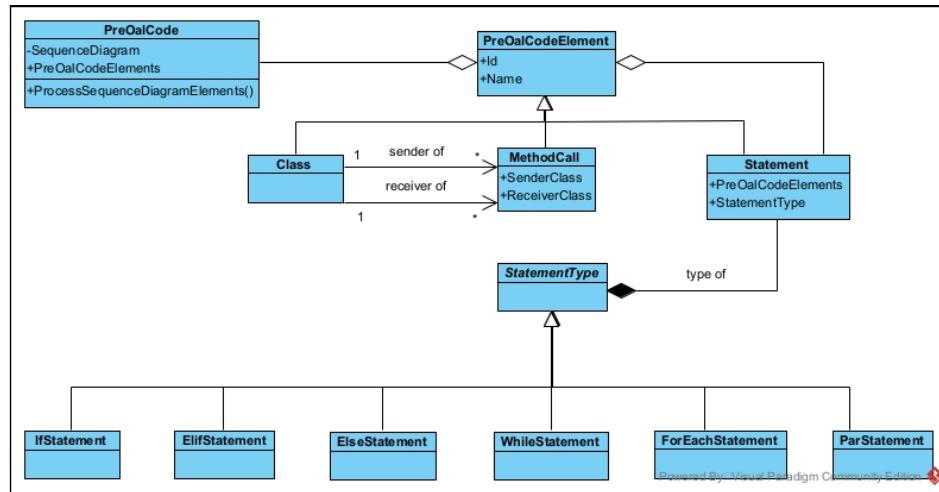
Objekt zmienenej triedy reprezentujúcej visitor, `SequenceDiagramCustomVisitor`, počas vykonávania svojej metódy `Visit()` vytvára objekt typu `SequenceDiagram`. Atri-

bút *Elements* objektu triedy *SequenceDiagram* metóda *Visit()* v priebehu svojho behu (resp. v priebehu prechádzania parsovacieho stromu) napĺňa elementami typu *SequenceDiagramElement*. Trieda *SequenceDiagramElement* je triedou, ktorej podriedy vyjadrujú samotné elementy sekvenčného diagramu. Jednotlivé podriedy majú štruktúru na základe súborov definujúce sekvenčné diagramu, pričom túto štruktúru sme analyzovali v kapitole 2.1.2.1.

### 2.1.3.2 Predpríprava na konverziu do OAL kódu

Konverziu do OAL kódu vykonávame najmä na základe objektov typu *Lifeline*, *Message* a *CombinedFragment*, vyjadrujúcich príslušné elementy sekvenčného diagramu (teda lifeline, správa a fragment). Objekty typu *SequenceDiagramElement*, spomenuté v predošej podkapitole, sú žiaľ medzi sebou prepojené komplikovaným spôsobom. Preto tieto objekty analyzujeme, aby sme ich mapovali na nové objekty, na základe ktorých je možné samotnú konverziu do zdrojového kódu vykonať jednoduchšie.

Dá sa povedať, že takto vykonávame predspracovanie (v angličtine „preprocessing“) pred transformáciou sekvenčných diagramov na OAL kód. Táto časť v rámci procesu konverzie do zdrojového kódu je v kóde vykonávaná prostredníctvom tried, ktoré sú zobrazené v diagrame tried na obrázku 2.7.



Obr. 2.7: Diagram tried predprípravy na konverziu do OAL kódu.

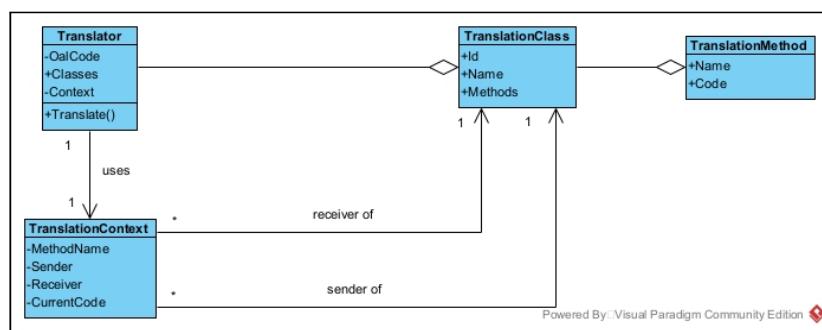
Predspracovanie elementov sekvenčného diagramu je v našom kóde vykonávané cez triedu *PreOalCode*. Trieda *PreOalCode* má cez atribút *SequenceDiagram* referenciu na objekt *SequenceDiagram*, o ktorom sme písali v predošej podkapitole. Metódou *ProcessSequenceDiagramElements* trieda *PreOalCode* prechádza elementy sekvenčného diagramu, analyzuje ich, mapuje na objekty typu *PreOalCodeElement* a následne ukladá do svojho atribútu *PreOalCodeElements*. Objekty typu *PreOalCodeElement* sú také objekty, ktoré uľahčujú neskôršiu konverziu do OAL kódu.

Konkrétny objekt tohto typu, resp. podrieda triedy *PreOalCodeElement*, s názvom *Class*, predstavuje objekt v OAL kóde, ktorá v sekvenčnom diagrame bola vyjadrená elementom lifeline. Trieda *MethodCall* zas predstavuje volanie metódy, pričom obsahuje referenciu na triedu, z ktorej bolo volanie metódy vyvolané (atribút *SenderClass*) a referenciu na triedu, do ktorej volanie metódy smeruje (atribút *ReceiverClass*). Volanie metódy v sekvenčnom diagrame pre nás predstavuje prvok správa, resp. message. Tretia a posledná podrieda triedy *PreOalCodeElement*, podrieda *Statement*, predstavuje v kontexte OAL kódu príkazy, cykly a paralelný blok. Táto trieda je v kompozícii s triedou *StatementType*, ktorá zas vyjadruje konkrétny typ príkazu, v závislosti od svojich podriedied. Podriedy *StatementType* môžu predstavovať príkazy „*if*“, „*elif*“, „*else*“, „*for each*“, „*while*“ a „*par*“.

Jednotlivé príkazy sa do OAL kódu prekladajú podľa konkrétnych fragmentov v danom sekvenčnom diagrame. Keďže fragmenty v sekvenčnom diagrame môžu v sebe obsahovať iné fragmenty, či správy, trieda *Statement* obsahuje atribút *PreOalCodeElements* obsahujúci ďalšie objekty typu *PreOalCodeElement*, ktoré daný príkaz, resp. fragment obsahuje.

### 2.1.3.3 Transformácia do OAL kódu

Z predpripravených objektov typu *PreOalCodeElement*, uvedených v predchádzajúcej časti, sa postupne vytvára zdrojový kód v jazyku OAL. Samotný OAL kód sa vytvára prostredníctvom triedy *Translator* a z tried prepojených s touto triedou. Diagram tried obsahujúci tieto triedy sa nachádza na obrázku 2.8.



Obr. 2.8: Diagram tried konverzie do OAL kódu.

Objekty typu *PreOalCodeElement* sa v metóde *Translate()* triedy *Translator* prechádzajú sekvenčne podľa toho, ako boli predpripravené v triede *PreOalCode*. V prípade, že sa v rámci prechádzania objektov v metóde *Translate()* prejde na objekt typu *Statement*, jeho elementy sa v rámci nášho riešenia prechádzajú rekurzívne. Pri prechádzaní elementov sekvenčného diagramu sa tiež kontroluje, či bol sekvenčný diagram na vstupe korektný, teda či ho je možné preložiť do adekvátneho OAL kódu. V prípade,

že korektný neboli, po odhalení tejto skutočnosti sa prechádzanie elementov diagramu ukončí.

V triede *Translator* počas konverzie do OAL kódu je využitá trieda *TranslationContext*. Trieda *TranslationContext* slúži na enkapsuláciu niekoľkých užitočných dát počas prekladu do OAL kódu. Objekt tejto triedy obsahuje napríklad meno aktuálnej metódy, ktorá sa prekladom vytvára, odkaz na odosielajúcu a prijímajúcu triedu v rámci procesu prekladu, či text kódu, ktorý sa v danom momente vytvára. Napríklad v situáciach, keď sa pri transformovaní do OAL kódu prejde na volanie metódy, ktorá smeruje z inej triedy ako predošlé volanie, v objekte *TranslationContext* sa takáto skutočnosť zaznamená a aktuálny kód sa použije pri vytvorení novej metódy alebo aktualizovaní kódu už existujúcej metódy.

Metódy sa v rámci prekladu zaznamenávajú do objektov *TranslationClass* (vyjadrujúce triedy OAL kódu) a ich atribútu *Methods*, ktorý zas predstavuje metódy OAL kódu. Metódy OAL kódu sú vyjadrené triedou *TranslationMethod*. Triedy *TranslationClass* aj *TranslationMethod* obsahujú svoj názov v atribúte *Name*, pričom objekty *TranslationMethod* obsahujú v atribúte *Code* aj konkrétny kód v jazyku OAL, prislúchajúci k danej metóde.

Trieda *Translator* jednotlivé triedy OAL kódu, znázornené triedou *TranslationClass*, zoskupuje v atribúte *Classes*. Všetky tieto preložené triedy, spolu s ich metódami, sú využité pri generovaní súborov s OAL kódom.

#### 2.1.3.4 Generovanie súboru s OAL kódom

Výstupom našej metódy konverzie sekvenčných diagramov do zdrojového kódu je OAL kód. OAL kód ako výstup prostredníctvom súborov poskytujeme dvojakým spôsobom.

Jedným z týchto spôsobov je pomocou textového súboru s OAL kódom. Tento súbor obsahuje kód v konvenciách syntaxe jazyka OAL, teda napr. triedy a metódy sú ukončené štruktúrou „*end class;*“ resp. „*end method;*“, triedy obsahujú konštruktory a pod.

Druhý spôsob poskytnutia OAL kódu je prostredníctvom súborov vo formáte JSON, ktoré definujú tzv. animácie v nástroji *AnimArch*. V rámci tohto typu súboru je pre každú triedu zobrazený jej názov a prípadné atribúty, ako aj jej metódy s názvom a OAL kódom. V súbore definujúcim animáciu v programe *AnimArch* sú v kontexte OAL kódu zanedbané niektoré konštrukcie, ako konštruktory tried, či ukončenia tried a metód štruktúrou „*end*“. Ukážka obsahu animačného súboru sa nachádza na obrázku 2.9.

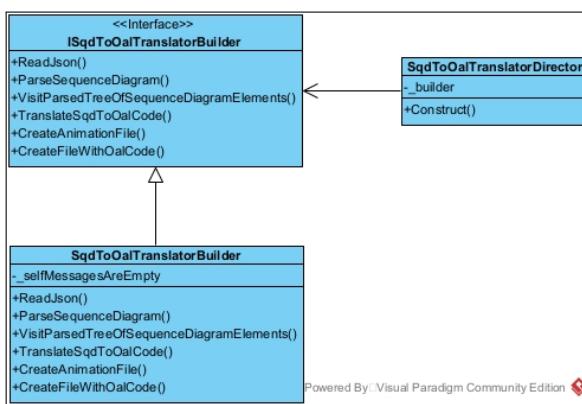
V prípade, že daný sekvenčný diagram neboli korektný, a teda ho nebolo možné preložiť do adekvátneho OAL kódu, výstupný súbor s OAL kódom obsahuje iba komentár o tejto skutočnosti, a v prípade animačného súboru tento súbor neobsahuje

```
{
  "Code": "",
  "AnimationName": null,
  "StartClass": "Client",
  "StartMethod": "FirstMethod",
  "MethodsCodes": [
    {
      "Name": "Client",
      "Methods": [
        {
          "Name": "FirstMethod",
          "Code": "create object instance Veterinarian_inst of Veterinarian;\nVeterinarian_inst.Register();\n\nVeterinarian_inst.Register();\nVeterinarian_inst.SetDate();\nVeterinarian_inst.SetDate();\n"
        }
      ]
    },
    {
      "Name": "Veterinarian",
      "Methods": [
        {
          "Name": "Register",
          "Code": "create object instance Animal_inst of Animal;\nAnimal_inst.SetVaccinationDate();\n"
        },
        {
          "Name": "SetDate",
          "Code": "self.VaccinateAnimals();\nfor each Animal in self.RegisteredAnimals\ncreate object instance Animal_inst of Animal;\nAnimal_inst.ReceiveVaccine();\nend for;\n"
        },
        {
          "Name": "VaccinateAnimals",
          "Code": ""
        }
      ]
    },
    {
      "Name": "Animal",
      "Methods": []
    }
  ]
}
```

Obr. 2.9: Ukážka obsahu animačného súboru prototypu *AnimArch*.

žiadne triedy, ani metódy s kódom.

Na dosiahnutie výsledných súborov s OAL kódom je v našom kóde potrebné na viacerých úrovniach konfigurovať vytváranie jednotlivých objektov. Pre lepšiu kontrolu nad konštrukciou týchto objektov, ako aj pre uľahčenie ich samotnej konštrukcie, či pre oddelenie konštrukcie od reprezentácie daných objektov sme vytváranie súborov s OAL kódom implementovali v súlade s návrhovým vzorom *Builder*. Diagram tried implementácie návrhového vzoru *Builder* v našom riešení sa nachádza na obrázku 2.10.

Obr. 2.10: Diagram tried vzoru *Builder* v našom riešení.

Ako je možné z obrázku 2.10 vidieť, rozhranie *ISqdToOalTranslatorBuilder* predstavuje v kontexte návrhového vzoru *Builder* element „*Builder*“, trieda *SqdToOalTranslatorBuilder* zas element „*Concrete Builder*“ a trieda *SqdToOalTranslatorDirector* ele-

ment „*Director*“. Trieda *SqdToOalTranslatorDirector* vyvoláva svojou metódou *Construct()* metódy triedy *SqdToOalTranslatorBuilder* (spolu s prípadnými požadovanými atribútmi) a určuje poradie vykonania týchto metód.

Metódy triedy *ISqdToOalTranslatorBuilder*, resp. *SqdToOalTranslatorBuilder* predstavujú takmer všetky kroky, ktoré sme opisovali v tejto kapitole. Metóda *ReadJson()* načíta JSON súbor definujúci sekvenčný diagram, metódou *ParseSequenceDiagram* sa parsovanie sekvenčného diagramu vyvolá a metóda *VisitParsedTreeOfSequenceDiagramElements* vyvolá nami upravené prechádzanie parsovacieho stromu prostredníctvom visitoru. Následne metóda *TranslateSqdToOalCode* postupne preloží jednotlivé elementy sekvenčného diagramu získané parsovaním, resp. visitorom. Metóda s názvom *CreateAnimationFile* ukladá preložený OAL kód do animačného súboru pre nástroj *AnimArch* a metóda *CreateFileWithOalCode* obyčajný OAL kód ukladá do textového súboru.

Trieda *SqdToOalTranslatorBuilder* obsahuje parameter *\_selfMessagesAreEmpty*, ktorý určuje, či tzv. vlastné správy, teda správy s rovnakým odosielateľom a príjemcom majú byť vo výslednom kóde prázdne alebo nie. Túto skutočnosť určuje používateľ pred zavolaním metódy *Construct()*. O dvojakom chápaní vykonávania vlastných správ v kontexte konverzie sekvenčných diagramov na OAL kód (teda toho, či vlastné správy majú, resp. nemajú byť prázdne) píšeme viac v kapitole 3.1.4.

#### 2.1.3.5 Spustenie animácie

Ako posledný krok procesu generovania OAL kódu zo sekvenčného diagramu považujeme jeho využitie v konkrétnom prípade, a to jeho využitie v nástroji *AnimArch*. Ako sme uviedli v predošej podkapitole, jedným z výsledkov konverzie súboru popisujúceho sekvenčný diagram je súbor pre nástroj *AnimArch*, definujúci animáciu procesov UML modelov a obsahujúci potrebné elementy OAL kódu na realizáciu danej animácie.

Aby bol však tento súbor v programe *AnimArch* aplikovateľný, musí byť spustený pre diagram tried obsahujúci triedy a metódy použité v sekvenčnom diagrame. Takáto požiadavka zahŕňa samozrejme zachovanie rovnakých názvov tried a metód, ako aj zachovanie daných podmienok pre jednotlivé cykly a podmienky v metódach tried.

Viac o vytváraní a spúštaní animácií v *AnimArch*-u píšeme v kapitole 2.2.2.

#### 2.1.4 Diagram aktivít procesu generovania OAL kódu

Proces konverzie sekvenčného diagramu do zdrojového kódu so sebou prináša celú škálu fáz, ktoré sme opisovali v predošlých podkapitolách. Niektoré zo spomenutých krokov sú z pohľadu používateľa voliteľné, ako napr. spustenie animácie v programe *AnimArch*, alebo nepotrebné, čo je napr. vytvorenie sekvenčného diagramu – samozrejme iba v prípade, ak daný sekvenčný diagram už vytvorený bol.

Pre komplexnosť jednotlivých krokov procesu riešenia nášho problému pokladáme za užitočné dané kroky vizualizovať. Vizualizáciu procesu transformácie sekvenčného diagramu do OAL kódu prinášame v tejto podkapitole, a to pomocou dynamického modelu. Konkrétnie sme sa rozhodli využiť UML diagram aktivít. Diagram aktivít predstavujúci spomínaný proces transformácie sa nachádza na obrázku 2.11.

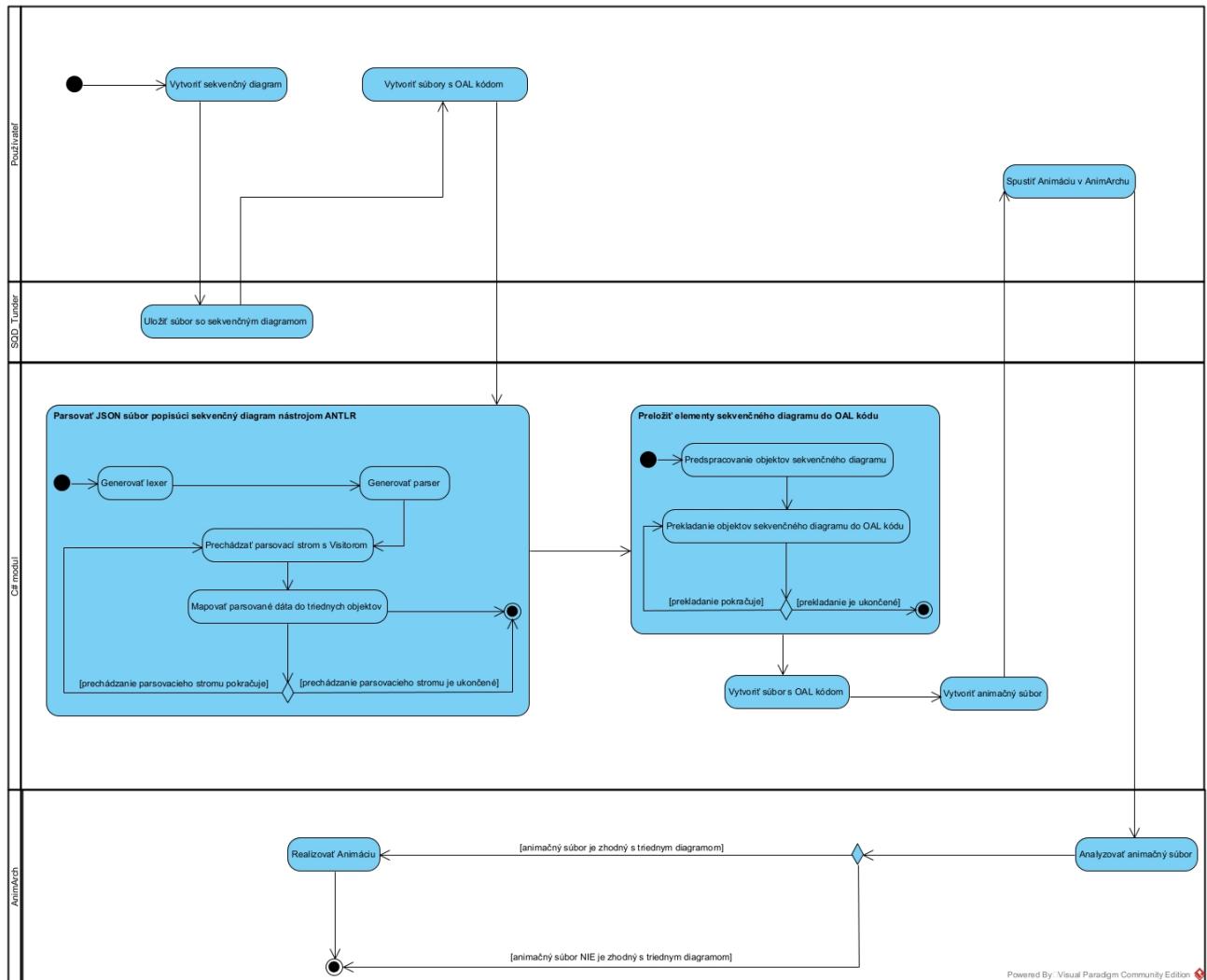
Konkrétnie je v tomto dynamickom modeli, resp. v diagrame aktivít znázornený prípad použitia, kde užívateľ vytvorí sekvenčný diagram, následne prostredníctvom našej implementácie s vytvoreným diagramom na vstupe vytvorí súbor s OAL kódom a animačný súbor, a na záver na základe novovytvoreného animačného súboru spustí animáciu v programe *AnimArch*.

V diagrame aktivít sme použili niekoľko „plávacích dráh“ (v angličtine „swimlines“). Prvá plávacia dráha sa týka akcií vykonaných používateľom. Keďže na vykonanie realizovania generovania OAL kódu sa vyžaduje viacero interakcií s používateľom, všetky tieto interakcie sú ako akcie priradené do takejto plávacej dráhy.

Druhá plávacia dráha obsahuje akciu realizovanú cez projekt, resp. aplikáciu na vytváranie sekvenčných diagramov, *SQD\_Tunder*. Do tejto plávacej dráhy sme pridali najdôležitejšie akcie vykonávané aplikáciou *SQD\_Tunder*, ktorými sú vytváranie a ukladanie sekvenčného diagramu. Ako sme popísali v predošlých kapitolách, *SQD\_Tunder* ukladá sekvenčné diagramy do súborov vo formáte JSON, na základe ktorých sa potom vykonáva konverzia do OAL kódu.

Tretia plávacia dráha opisuje akcie vykonané nami implementovaným programom, resp. modulom v jazyku C#. V rámci tohto modulu prebieha parsovanie JSON súboru definujúceho sekvenčný diagram, mapovanie elementov sekvenčného diagramu na konkrétnie triedne objekty, preklad týchto objektov do OAL kódu a vytvorenie súborov s OAL kódom. Proces parsovania súboru so sekvenčným diagramom nástrojom ANTLR je v plávacej dráhe tohto modulu znázornený v samostatnej aktivite, ktorá obsahuje jednotlivé akcie. Taktiež je v samostatnej aktivite zobrazený preklad do OAL kódu. Táto plávacia dráha obsahuje ešte akcie znázorňujúce vytváranie súborov s OAL kódom.

V poslednej plávacej dráhe sú znázornené akcie v rámci nástroja *AnimArch*. Nástrojom *AnimArch* je spustená animácia cez súbor, ktorý ju definuje a ktorý bol vytvorený nami implementovaným modulom. V tejto plávacej dráhe je znázornená aj kontrola, či súbor špecifikujúci animáciu je korektný, teda či je zhodný s diagramom tried, v rámci ktorého má byť v prototype *AnimArch* spustený.



Obr. 2.11: Diagram aktivít procesu generovania OAL kódu.

Používateľ teda na začiatku procesu vytvorí sekvenčný diagram programom *SQD\_Tunder*, ktorý tento sekvenčný diagram uloží do samostatného súboru. Nasleduje transformovanie tohto sekvenčného diagramu do OAL kódu pomocou nášho riešenia, pričom OAL kód sa generuje do textového súboru s obyčajným OAL kódom, ako aj do súboru umožňujúceho spúšťať animáciu v *AnimArch*-u.

Po zavolaní nášho modulu sa začína v rámci parsovania lexikálna a syntaktická analýza nástrojom ANTLR, potom prechádzanie vrcholov parsovacieho stromu a mapovanie elementov sekvenčného diagramu na triedne objekty. Prechádzanie vrcholov parsovacieho stromu pokračuje dovtedy, dokým sa tento strom neprejde celý. Po ukončení aktivít súvisiacich s nástrojom ANTLR nasleduje samotný preklad jednotiek sekvenčného diagramu do OAL kódu, pričom prekladu do OAL kódu predchádza predspracovanie jednotiek sekvenčného diagramu, kvôli ich komplexnosti. V konkrétnej aktivite „*Preložiť elementy sekvenčného diagramu do OAL kódu*“ nie je explicitne vyjadrené, či prekladanie do OAL kódu bolo ukončené správne alebo bolo ukončené kvôli nekorekt-

nosti vstupného sekvenčného diagramu. V kontexte ďalších akcií v diagrame aktivít túto skutočnosť nie je nutné znázorňovať.

Po preložení sekvenčného diagramu do OAL kódu sa vytvorí súbor s OAL kódom a animačný súbor. Poradie vykonania týchto dvoch akcií je zanedbateľné, keďže vytvorenie daného typu súboru s OAL kódom nijakým spôsobom neovplyvňuje vytvorenie druhého súboru, ako aj interakciu s týmto druhým súborom. Po vytvorení animačného súboru používateľ chce tento súbor využiť pri spustení animácie v nástroji *AnimArch*. Predpokladá sa, že používateľ už má v programe *AnimArch* pripravený diagram tried, v ktorom môže danú animáciu spustiť. Pokiaľ nemá, alebo pokiaľ animácia definovaná v animačnom súbore nekorešponduje s diagramom v *AnimArch-u*, animácia sa nespustí, čo je v našom diagrame aktivít aj znázornené. Taktiež pokiaľ vytvorený sekvenčný diagram neboli korektný, animačný súbor neobsahuje potrebné OAL triedy a metódy, na základe ktorých by sa animácia dala spustiť, hoci by diagram tried bol správne definovaný.

Naším diagramom aktivít na obrázku 2.11 sme teda na konkrétnom prípade použitia demonstrovali hlavné kroky vedúce ku generovaniu OAL kódu, ako aj interakciu nášho riešenia cez používateľa s prototypmi *SQD\_Tunder* a *AnimArch*.

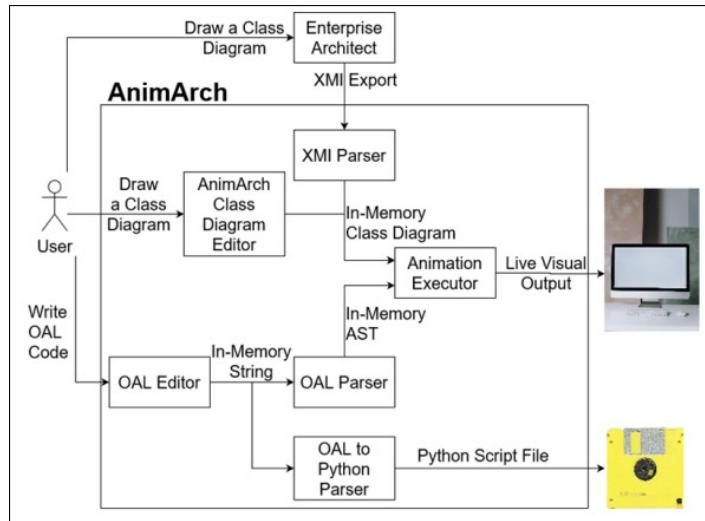
Čo sa týka znázornenia nášho programu resp. modulu v štrukturálnej podobe, v predošej podkapitole 2.1.3 sme pre jednotlivé časti procesu konverzie sekvenčných diagramov do OAL kódu predstavili diagramy tried. Diagram tried zobrazujúci celý nami definovaný program si je možné pozrieť (kvôli jeho rozsiahlosti) v prílohe A.

## 2.2 Nástroj AnimArch

*AnimArch* je prototyp vybudovaný v programovacom jazyku C#, konkrétniešie vo vývojom prostredí *Unity 3D*. Používateľským cieľom prototypu *AnimArch* je modelovanie softvérových systémov. Jeho nezanedbateľným používateľským cieľom je aj možnosť zjednodušenia porozumenia daných modelov. Nástroj je tak možné využiť aj na didaktické účely. S pomocou interaktívnych prvkov, ktoré ponúka, sa dajú napríklad efektívnejšie vysvetliť architektonické štýly, či návrhové vzory.

Tok údajov v nástroji *AnimArch* je pomerne komplexný. Ako aj uvedieme v tejto kapitole, v nástroji *AnimArch* je možné zobrazovať diagramy tried, vytvárať alebo upravovať diagramy tried, definovať a spúštať animácie procesov v týchto diagramoch, ako aj počas vytvárania animácií priamo definovať OAL kód. Tento OAL kód je následne možné konvertovať do kódu v programovacom jazyku Python. Vizualizáciou na obrázku 2.12 je zosumarizovaný tok údajov pri práci v nástroji *AnimArch*.

Z hľadiska našej práce je hlavným cieľom umožniť spustenie vizualizácie procesov sekvenčných diagramov v nástroji *AnimArch*. V tejto kapitole vysvetlíme všetky

Obr. 2.12: Tok údajov v nástroji *AnimArch* [16].

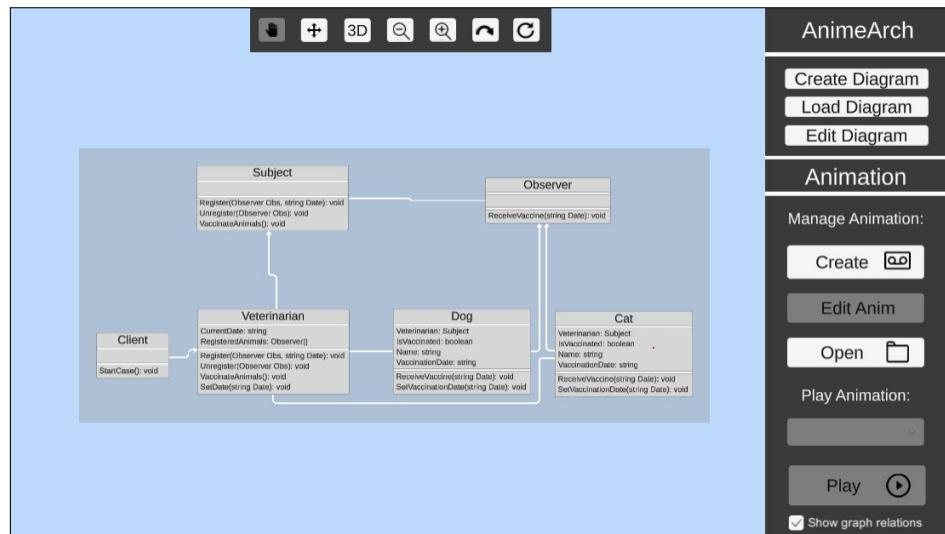
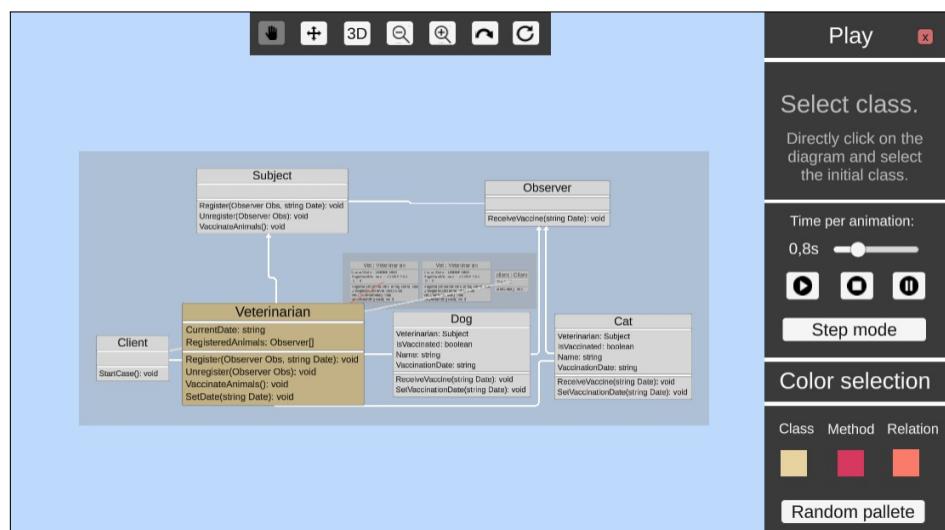
základné funkcionality nástroja *AnimArch*.

### 2.2.1 Načítanie a vytvorenie diagramu tried

Diagram tried sa v prototype *AnimArch* dá načítať kliknutím na tlačidlo „*Load diagram*“. Diagramy tried sú v *AnimArch*-u načítavané zo súborov vo formáte *XMI*, kompatibilných so softvérom *Enterprise Architect*, čo je program slúžiaci na modelovanie UML diagramov. Vytváranie nových diagramov tried je umožnené stlačením tlačidla „*Create diagram*“. Pri procese vytvárania diagramov tried je možné daným triedam definovať ich metódy, atribúty, parametre metód, návratový typ metód a pod. Diagramy je možné upravovať po kliknutí na tlačidlo „*Edit diagram*“. Vytvorené diagramy je možné uložiť vo formáte JSON, alebo ako súbor použiteľný v programe *Enterprise Architect* vo formáte *XMI*. Na obrázku 2.13 je zobrazený nástroj *AnimArch* spolu s uvedenými tlačidlami a so zobrazeným diagramom tried.

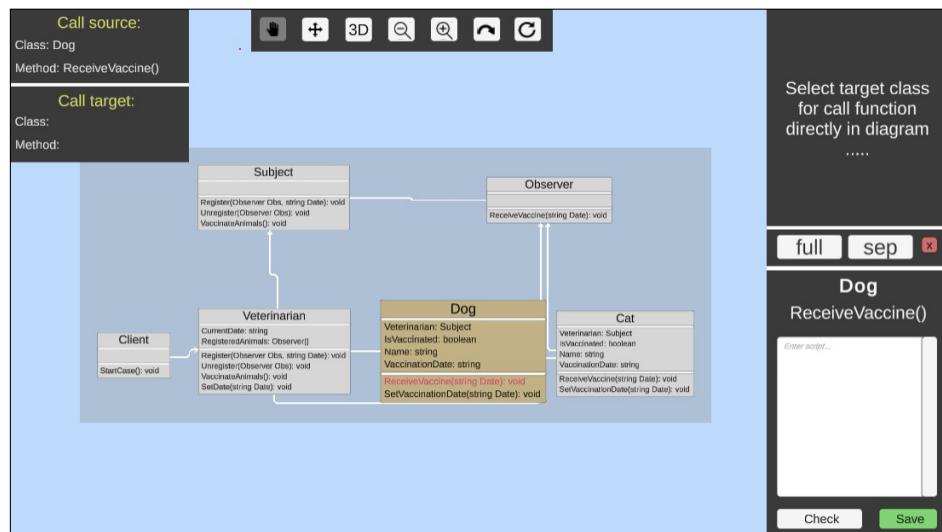
### 2.2.2 Spustenie animácie

Nástroj *AnimArch* umožňuje zobrazovať procesy v rámci daných tried, a to zvýrazňovaním tried, zvýrazňovaním daných metód a jednotlivých hrán. Tento priebeh udalostí je v rámci prototypu *AnimArch* nazývaný pojmom **animácia**. Animácia je v *AnimArch*-u spustená tak, že používateľ najskôr klikne na konkrétnu triedu diagramu tried, potom na jej metódu a následne klikne na tlačidlo „*Play*“. Počas procesu animácie sa zobrazuje na pozadí diagramu tried aj diagram komunikácie. Používateľ tak môže sledovať interakciu jednotlivých procesov aj v rámci ďalšej vrstvy prostredníctvom odlišného typu diagramu. Animácia v prototype *AnimArch* je vykonávaná prostredníctvom kódu v jazyku OAL. Ukážka animácie je zobrazená na obrázku 2.14.

Obr. 2.13: Ukážka nástroja *AnimArch*.Obr. 2.14: Ukážka vykonávania animácie v nástroji *AnimArch*.

### 2.2.3 Vytvorenie animácie

Po načítaní diagramu tried v nástroji *AnimArch* je možné pre daný diagram tried načítať, vytvoriť, prípadne editovať animáciu. Súbory s animáciou sú ukladané vo formáte JSON. Animáciu je možné vytvoriť, resp. editovať aj prostredníctvom definovania OAL kódu pre jednotlivé metódy tried. Druhý a jednoduchší spôsob vytvárania procesu animácie je prostredníctvom priameho klikania na jednotlivé triedy a následne na ich metódy. Príklad vytvárania animácie v *Animarch-u* je možné vidieť na obrázku 2.15.



Obr. 2.15: Ukážka vytvárania animácie v nástroji *AnimArch*.

### 2.2.4 OAL kód v nástroji AnimArch

V predchádzajúcej podkapitole sme uviedli, že animácia v nástroji *AnimArch* je spušťaná prostredníctvom OAL kódu. OAL kód je vytváraný pre každú triedu, resp. metódu samostatne, a to buď kliknutím na jednotlivé triedy a ich metódy, alebo prostredníctvom definovania OAL kódu. V OAL kóde v prototype *AnimArch* nie sú potrebné všetky konštrukcie jazyka OAL, ako napríklad konštruktory, a preto takéto konštrukcie pri výtváraní animácií ani nie sú definované.

Z hľadiska jazyka OAL použitého v prototype *AnimArch* stručne opíšeme najhlavnejšie príkazy a konštrukcie tohto jazyka, ktoré sú v *AnimArch-u* využívané. Pri ich opise vychádzame z oficiálnej dokumentácie jazyka OAL [17], ako aj z diplomovej práce [18], kde boli predstavené a vytvorené nové konštrukcie v jazyku OAL, ktoré sú v programe *AnimArch* využívané .

Najvyužívanejšie príkazy a konštrukcie jazyka OAL v programe *AnimArch* sú tieto:

- **Priadenie atribútov**

Príkaz priadenia má v syntaxi jazyka OAL takúto štruktúru:

```
[assign] <instance handle>.<attribute> = <expression>;
```

V príkaze časť `<instance handle>` predstavuje inštanciu danej triedy, `<attribute>` konkrétny atribút a `<expression>` výraz, ktorý môže byť typu boolean, reťazec alebo aritmetický výraz. Výraz `assign` je v príkaze nepovinný. Príkaz priradenia funguje tak, že hodnotu v `<expression>` priradí do atribútu (`<attribute>`) v inštancii špecifikovanej prostredníctvom `<instance handle>`.

- **Vytvorenie inštancie**

Inštancie v jazyku OAL sú vytvárané takýmto spôsobom:

```
create object instance <instance handle> of <keyletter>;
create object instance of <keyletter>;
```

Štruktúra `<keyletter>` v príkaze je názov danej triedy a `<instance handle>` je názvom inštancie tejto triedy. Za `<instance handle>` v uvedenom príkaze nie je možné použiť výraz `.self.` Výraz `self.` predstavuje volanie metódy v rámci jednej triedy, teda volanie vlastnej metódy. Zaujímavosťou je, že podľa syntaxe jazyka OAL je možné vytvoriť inštanciu danej triedy aj bez definovania názvu tejto inštancie.

- **Volanie metód**

Metódu je možné v jazyku OAL volať ako:

```
<instance handle>.<method name>;
<variable> = <instance handle>.<method name>;
```

V prvom prípade sa jedná iba o zavolanie metódy (kde `<instance handle>` je názov inštancie triedy a `<method name>` názov metódy). V druhom prípade je znázornnené aj priradenie do určite premennej `<variable>` v situácii, kde volaná metóda vracia aj nejaký výsledok.

- **Konštrukcia *if***

Konštrukcia `if` má v syntaxe jazyka OAL takúto štruktúru:

```
if (<boolean expression>
    <statements>
    elif (<boolean expression>
        <statements>
    else
        <statements>
    end if;
```

Syntax konštrukcie **if** je podobná ako v rôznych programovacích jazykoch. Ak v **if** vetve nie sú splnené podmienky v <boolean expression>, vykoná sa vetva **elif**. Ak podmienka v **if** vetve nie je splnená, resp. ak ani žiadna z podmienok v **elif** vetvách nie je splnená, vykoná sa časť <statements> vo vetve **else**.

Na uvedenej syntaxi **if** konštrukcie si je zaujímavé všimnúť, že konštrukcia **if** je v prípade prítomnosti **elif** vetiev alebo **else** vetvy ukončená spoločne príkazom **end if;**.

- **Cyklus for each**

V jazyku OAL je možné prechádzať elementy určitej kolekcie cyklom **for each**. Takýto cyklus má syntax:

```
for each <instance handle> in <instance handle set>
    <statements>
end for;
```

V štruktúre cyklu **for each** časť <instance handle> predstavuje lokálnu premennú pre objekty v kolekcii a <instance handle set> predstavuje premennú odkazujúcu sa na konkrétnu kolekciu. Príkazy v rámci **for each** cyklu sú prítomné v časti <statements>.

- **Cyklus while**

Cyklus, ktorý je vykonávaný, dokým je splnená nejaká podmienka, je rovnako ako v iných programovacích jazykoch vykonávaný v jazyku OAL cyklom s názvom **while**. V prípade jazyka OAL má cyklus **while** syntax:

```
while (<boolean expression>)
    <statements>
end while;
```

V cykle **while** je časť <boolean expression> logickým výrazom, ktorý je možné ohodnotiť pravdivostnou hodnotou ako pravdivý alebo nepravdivý. V rámci cyklu **while** táto časť podmieňuje jeho vykonávanie dovtedy, kým je vyhodnotená ako pravdivá.

- **Príkaz paralného bloku par**

V *AnimArch-u* je možné vykonávať procesy paralelne pomocou konštrukcie s názvom **par**, ktorý má takúto syntax:

```

par (<boolean expression>
      thread
          <statements>
      end thread;
  end par;

```

Konštrukcia **par** obaľuje vlákna, ktoré majú byť v rámci paralelného bloku vykonané. Každé vlákno je definované pomocou výrazu **thread** a končí výrazom **end thread;**. Paralelný blok môže obsahovať viac vlákiem, a teda aj viac konštrukcií **thread**.

Táto konštrukcia nie je súčasťou pôvodného jazyka OAL, ale v rámci implementácie programu *AnimArch* bola do jazyka OAL pridaná.

- **Vytvorenie zoznamu a pridanie/odstránenie prvku** V rámci implementácie prototypu *AnimArch* boli do jazyka OAL pridané aj konštrukcie na vytvorenie zoznamu, na pridanie prvku do zoznamu, ako aj na odstránenie prvku do zoznamu. Tieto konštrukcie majú v syntaxe rozšíreného jazyka OAL takýto tvar:

```

create list <list name> of <keyletter>;
add <instance handle> to <list>;
remove <instance handle> from <list>;

```

Konštrukciou **create list <list name> of <keyletter>;** je vytvorený zoznam s názvom **<list name>**, ktorý môže obsahovať prvky triedy s názvom **<keyletter>**.

Výraz **add <instance handle> to <list>;** predstavuje pridanie prvku danej triedy (časť **<instance handle>**) do zoznamu s názvom **<list>**.

Odstránenie prvku zo zoznamu je uskutočnené výrazom **remove <instance handle> from <list>;**, ktorého štruktúra je v zásade rovnaká ako štruktúra výrazu na pridanie prvku do zoznamu.

Kód v jazyku OAL je v nástroji *AnimArch* parsovaný s použitím nástroja ANTLR. Časť najzákladnejších pravidiel, na základe ktorých sú spomenuté konštrukcie jazyka OAL v prototype *AnimArch* extrahované, sú zobrazené na obrázku 2.16.

#### *Jednotlivé pravidlá znamenajú:*

- **exeCommandAssignment:** pravidlo definujúce priradenie.
- **exeCommandCall:** pravidlo definujúce volanie metód.
- **exeCommandQueryCreate:** pravidlo definujúce vytvorenie inštancie triedy.

- **ifCommand**: pravidlo pre konštrukciu príkazu `if`. Na základe tohto pravidlá sa parsujú vety `if`, `elif` a `else`.
- **foreachCommand**: pravidlo pre `for each` cyklus.
- **whileCommand**: pravidlo pre `while` cyklus.
- **parCommand**: pravidlo pre paralelný blok, teda konštrukciu `par`.
- **exeCommandCreateList**: pravidlo pre vytvorenie zoznamu.
- **exeCommandAddingToList**: pravidlo pre priradenie prvku do zoznamu.
- **exeCommandRemovingFromList**: pravidlo pre odstránenie prvku zo zoznamu.

```

exeCommandQueryCreate
:   'create object instance ' instanceHandle ' of ' keyLetter ';'
|   'create object instance of ' keyLetter ';'
;

exeCommandCall
:   instanceHandle '.' methodName '(' (expr (',' expr)*)? ')' ';'
;

exeCommandAssignment
:   ('assign')? instanceHandle '=' expr ';'
;

ifCommand
:   'if' expr line* ('elif' '(' expr ')' line+)* ('else' line+)? 'end if' ';'
;

foreachCommand
:   'for each' variableName ' in ' instanceHandle line+ 'end for' ';'
;

whileCommand
:   'while' '(' expr ')' line+ 'end while' ';'
;

parCommand
:   'par' 'thread' line+ 'end thread' ';' ('thread' line+ 'end thread' ';')+ 'end par' ';'
;

exeCommandCreateList
:   'create list' instanceHandle ' of ' keyLetter ('{' instanceHandle (',' instanceHandle)* '}')? ';'
;

exeCommandAddingToList
:   'add' instanceHandle ' to ' instanceHandle ';'
;

exeCommandRemovingFromList
:   'remove' instanceHandle ' from ' instanceHandle ';'
;

```

Obr. 2.16: Časť pravidiel gramatiky parsera v nástroji *AnimArch*.

# Kapitola 3

## Náš prístup k problematike generovania zdrojového kódu z dynamických modelov

Na problematiku generovania zdrojového kódu z dynamických modelov, resp. zo sekvenčných diagramov existuje viacero prístupov, pričom niektoré z nich sme uviedli a analyzovali v kapitole 1.6. Môžeme konštatovať, že naša metóda transformácie sekvenčných diagramov na kód vo vysoko-abstraktnom jazyku OAL je vo svojej podstate unikátna.

V tejto kapitole najskôr postupne uvedieme, ako sú konkrétnie typy sekvenčných diagramov preložené do kódu v jazyku OAL. Potom predstavíme tzv. pravidlá, na základe ktorých identifikujeme z nášho pohľadu nekorektné sekvenčné diagramy, ktoré do kódu v jazyku OAL v našom riešení neprekladáme.

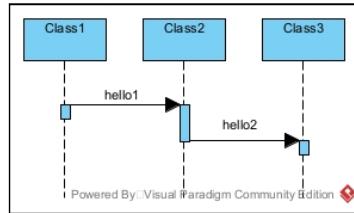
### 3.1 Prístupy konverzie sekvenčných diagramov do OAL kódu

V tejto časti postupne predstavujeme, ako do OAL kódu prekladáme rôzne varianty sekvenčných diagramov.

#### 3.1.1 Konvertovanie sekvenčného diagramu s jednou správou

Ako prvý opíšeme preklad jednoduchého sekvenčného diagramu s dvoma lifelinami a jednou správou. Tento sekvenčný diagram je zobrazený na obrázku 3.1. Správa v sekvenčnom odoslaná z prvej lifeline (označenej ako *Class1*) a smerujúca do druhej lifeline (označenej ako *Class2*) predstavuje volanie metódy triedy *Class2*, pričom názov metódy je *hello*. Kód v jazyku OAL, zodpovedajúci tomuto sekvenčnému diagramu, je

zobrazený v kóde 3.1. Metódu, ktorou sa v OAL kóde spúšťajú prvé volania znázornené v sekvenčnom diagrame, označujeme ako *StartMethod*. V kóde uvádzame konštruktory tried, definíciu jednotlivých metód, vytvorenie inštancie triedy *Class2* a vyvolanie metódy *hello1* triedy *Class2* v metóde *StartMethod*.



Obr. 3.1: Sekvenčný diagram s jednou správou.

Ukážka kódu 3.1: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.1.

---

```

class Class1
    constructor Class1()
    end constructor;

    method StartMethod()
        create object instance Class2_inst of Class2;
        Class2_inst.hello();
    end method;
end class;

class Class2
    constructor Class2()
    end constructor;

    method hello()
    end method;
end class;

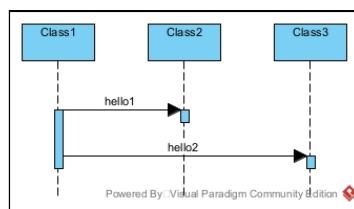
```

---

### 3.1.2 Konvertovanie sekvenčných diagramov s viacerými správami a lifelinami

Diagram na obrázku 3.2 je podobný ako predchádzajúci diagram na obrázku 3.1 z predchádzajúceho prípadu s tým, že správa z prvej lifeline smeruje aj do ďalšej lifeline s názvom *Class3*. V OAL kóde 3.2 sú tak dané správy sekvenčného diagramu (označené ako *hello1* a *hello2*) preložené ako metódy príslušných tried (*Class2* a *Class3*) a v triede *Class1* aj ako samotné volania týchto tried.

**Pre lepšiu prehľadnosť v OAL kódoch budeme vynechávať definíciu konštruktorov tried.**



Obr. 3.2: Sekvenčný diagram s viacerými správami a lifelinami.

Ukážka kódu 3.2: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.2.

---

```

class Class1
    method StartMethod()
        create object instance Class2_inst of Class2;
        Class2_inst.hello1();
        create object instance Class3_inst of Class3;
        Class3_inst.hello2();
    end method;
end class;

class Class2
    method hello1()
    end method;
end class;

class Class3
    method hello2()
    end method;
end class;

```

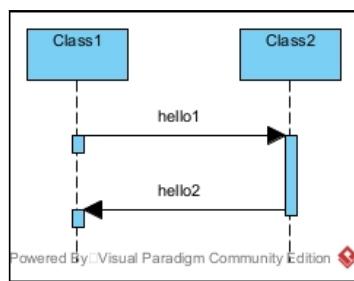
---

### 3.1.3 Konvertovanie sekvenčných diagramov so správami z iných tried

Obrázok 3.3 obsahuje diagram, ktorý prekladáme do OAL kódu mierne odlišným spôsobom než v predchádzajúcich prípadoch. Tento diagram obsahuje správu smerujúcu z prvej lifeline do druhej a následne správu smerujúcu z druhej lifeline do prvej. V nástroji *SQD\_Tunder* nevieme definovať aktivačné bloky pre jednotlivé správy, čo sme spomenuli aj v kapitole 2.1.1, a preto pri konverzii považujeme, že tieto správy majú implicitný návrat (return). Takýto predpoklad následne aplikujeme aj pri konvertovaní sekvenčných diagramov do OAL kódu.

Pre zachovanie toku procesov, resp. správ zobrazených v sekvenčnom diagrame musíme tok volaní rešpektovať aj v kóde. Ak po istej správe nasleduje správa, ktorá sa odosiela z inej triedy ako predošlá správa, tok volaní sa v kóde zmení a volania budú v prípadoch, aký je napríklad zobrazený v diagrame na obrázku 3.3, pokračovať poslednou metódou, ktorá bola v kóde vyvolaná. Tento koncept uvedieme na základe kódu 3.3, ktorý zodpovedá diagramu zobrazenému na 3.3. Prvá metóda *StartMethod* triedy *Class1* zavolala metódu *hello1* triedy *Class2*. V sekvenčnom diagrame správa *hello2* smeruje z prvej lifeline do druhej, čo v kóde znamená, že *Class2* volá metódu *hello2* triedy *Class1*.

Aby sme zachovali tok zobrazených procesov v sekvenčnom diagrame, musíme pokračovať volaniami z triedy *Class2*. Tokom volaní v tomto prípade budeme pokračovať tak, že volanie odosielané z triedy *Class2* bude odosielané v rámci metódy *hello1*, ktorá bola naposledy v kóde vyvolaná. V metóde *hello1* sa tak vytvára inštancia triedy *Class1* a volá sa metóda *hello2*, ktorá je v triede *Class1* adekvátne definovaná.



Obr. 3.3: Sekvenčný diagram so správou z inej lifeline.

Ukážka kódu 3.3: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.3.

---

```

class Class1
    method StartMethod()
        create object instance Class2_inst of Class2;
        Class2_inst.hello1();
    end method;

    method hello2()
    end method;
end class;

class Class2
    method hello1()
        create object instance Class1_inst of Class1;
        Class1_inst.hello2();
    end method;
end class;
  
```

---

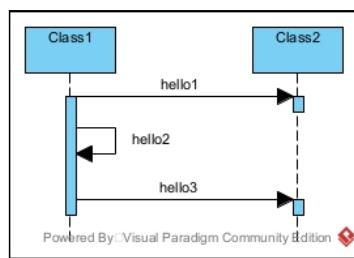
### 3.1.4 Konvertovanie sekvenčných diagramov s vlastnými správami

Sekvenčné diagramy môžu obsahovať tzv. „*self-messages*“, čo môžeme preložiť ako „*vlastné správy*“. Vlastné správy sú také správy, ktoré majú rovnakého odosielateľa a príjemcu, teda lifeline, z ktorej správa smeruje, je rovnaká ako lifeline, do ktorej táto správa smeruje.

Pri analyzovaní možnosti konverzie takéhoto typu správ do zdrojového kódu sme zistili, že vlastné správy je možné do zdrojového kódu preložiť dvojakým spôsobom.

#### 1) Vlastné správy ako prázdne metódy

Prvou z možných interpretácií prekladu vlastných správ do kódu je taká, že vlastné správy sú vo výslednom kóde prázdne metódy. Takúto situáciu si vysvetlíme na konkrétnom príklade, konkrétnie na sekvenčnom diagrame s vlastnou správou na obrázku 3.4. Tento sekvenčný diagram obsahuje 3 správy smerujúce z lifeline *Class1*, pričom správa *hello1* smeruje do triedy *Class2*, správa *hello2* ako vlastná správa smeruje do rovnakej lifeline *Class1* a správa *hello3* smeruje opäť do lifeline *Class2*.



Obr. 3.4: Sekvenčný diagram s vlastnou správou.

OAL kód, ktorý zodpovedá tomuto sekvenčnému diagramu, je kód 3.4. Kód 3.4 však zároveň zohľadňuje takú interpretáciu vlastných správy sekvenčného diagramu, kde vlastné správy sú preložené ako prázdne metódy. Ako je možné vidieť v kóde 3.4, metóda *StartMethod* zavolá metódu *hello1* triedy *Class2*, potom metódu svojej vlastnej triedy *hello2* a na záver metódu triedy *Class2* s názvom *hello3*.

Metóda *hello2* je v kóde definovaná ako vlastná metóda triedy *Class1* a zároveň ako prázdna metóda. Po jej zavolaní sa nepokračuje v toku volaní touto metódou, ale pokračuje sa v pôvodnej metóde, v našom prípade v metóde *StartMethod*.

Takýto pohľad na preklad vlastných správ do zdrojového kódu je možné teda vnímať tak, že vlastné správy sú v toku volaní zavolané ako prázdne metódy, ale v samotnom toku volaní sa nepokračuje týmito prázdnymi metódami - v toku volaní sa pokračuje v rámci metódy, z ktorej boli metódy reprezentujúce vlastné správy volané.

Ukážka kódu 3.4: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.4 pre prípad, kde sú vlastné správy zobrazené ako prázdne metódy.

---

```

class Class1
    method StartMethod()
        create object instance Class2_inst of Class2;
        Class2_inst.hello1();
        self.hello2();
        Class2_inst.hello3();
    end method;

    method hello2()
    end method;
end class;

class Class2
    method hello1()
    end method;

    method hello3()
    end method;
end class;
  
```

---

## 2) Vlastné správy ako štandardné metódy

Druhou z možných interpretácií prekladu vlastných správ do zdrojového kódu je taká interpretácia, kde vlastné správy sú do kódu preložené ako vlastné metódy

v rámci jednej triedy, avšak zároveň ako také metódy, v ktorých sa po ich zavolaní pokračuje v toku volaní.

Majme rovnaký sekvenčný diagram ako v predošлом prípade, teda diagram na obrázku 3.4. Kód, ktorý zodpovedá tomuto sekvenčnému diagramu, je kód 3.5. Kód 3.5 je podobný kódu 3.4 z predošlého prípadu. V kóde 3.5 je však metóda *hello2*, predstavujúca rovnomennú vlastnú správu, preložená ako štandardná metóda - teda ako taká metóda, ktorou sa po jej zavolaní pokračuje v toku volaní.

A teda vo všeobecnosti druhý pohľad na transformáciu vlastných správ do kódu je možné vnímať tak, že vlastné správy sa prekladajú do kódu tak isto, ako volania smerujúce do rozdielnych tried - s tým rozdielom, že tieto volané metódy sa vytvoria v rámci triedy, z ktorej sú volané.

Ukážka kódu 3.5: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.4 pre prípad, kde sú vlastné správy zobrazené ako štandardné metódy.

---

```

class Class1
    method StartMethod()
        create object instance Class2_inst of Class2;
        Class2_inst.hello1();
        self.hello2();
    end method;

    method hello2()
        create object instance Class2_inst of Class2;
        Class2_inst.hello3();
    end method;
end class;

class Class2
    method hello1()
    end method;

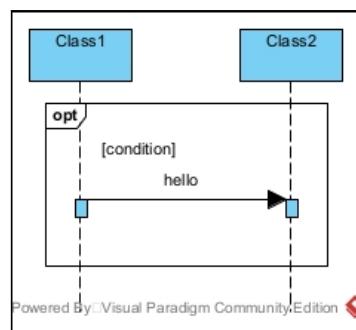
    method hello3()
    end method;
end class;

```

---

### 3.1.5 Konvertovanie sekvenčných diagramov s fragmentom typu OPT

V opise konverzie procesov sekvenčných diagramov do jazyka OAL pokračujeme so sekvenčnými diagramami obsahujúcimi fragmenty. Ako prvý s týchto fragmentov uvádzame sekvenčný diagram s fragmentom typu *OPT* (obrázok 3.5), obsahujúci podmienku *conditon*, pričom tento fragment vo svojom vnútri obsahuje aj správu *hello*. Ako môžeme vidieť v kóde 3.6, fragment je preložený do jazyka OAL ako príkaz *if*, pričom podmienka tohto fragmentu je aj podmienkou príkazu *if*,



Obr. 3.5: Sekvenčný diagram s fragmentom typu OPT.

Ukážka kódu 3.6: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.5.

```

class Class1
    method StartMethod()
        if (condition)
            create object instance Class2_inst of Class2;
            Class2_inst.hello();
        end if;
    end method;
end class;

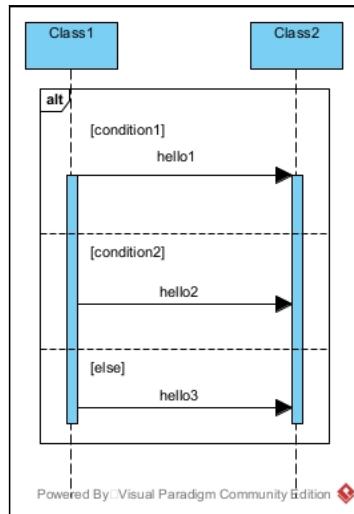
class Class2
    method hello()
    end method;
end class;

```

### 3.1.6 Konvertovanie sekvenčných diagramov s fragmentom typu ALT

Podobne ako sekvenčný diagram s fragmentom OPT je do OAL kódu prekladaný aj sekvenčný diagram s fragmentom typu ALT. Rozdiel medzi fragmentom OPT a ALT je, že fragment ALT môže obsahovať viac vetiev, resp. operandov s podmienkami. Sekvenčný diagram s fragmentom typu ALT, nachádzajúci sa na obrázku 3.6, obsahuje vo svojich operandoch podmienky *condition1*, *condition2* a *else* a v každom z nich jednu správu z *Class1* do *Class2*. Do kódu (kód 3.7), je takýto fragment preložený pomocou príkazov *if*, *elif* a *else*. V kóde si je možné všimnúť, že keďže inštancia triedy *Class2* nebola vytvorená pred príkazom *if* (pretože v sekvenčnom diagrame pred fragmentom ALT nesmerovala žiadna správa do *Class2*), tak v každej vetve príkazu je potrebné vytvoriť túto inštanciu.

V kóde 3.7 pre lepšiu prehľadnosť je zobrazený iba kód pre triedu *Class1*.



Obr. 3.6: Sekvenčný diagram s fragmentom typu ALT.

Ukážka kódu 3.7: Úkážka časti OAL kódu pre sekvenčný diagram na obrázku 3.6.

```

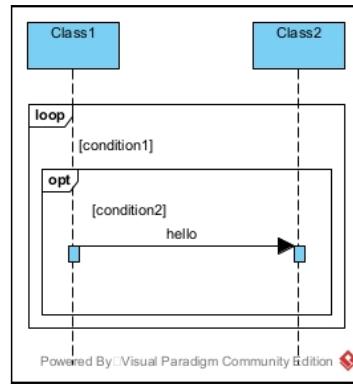
class Class1
method StartMethod()
    if (condition1)
        create object instance Class2_inst of Class2;
        Class2_inst.hello1();
    elif (condition2)
        create object instance Class2_inst of Class2;
        Class2_inst.hello2();
    else
        create object instance Class2_inst of Class2;
        Class2_inst.hello3();
    end if;
end method;
end class;

```

### 3.1.7 Konvertovanie sekvenčných diagramov s fragmentom typu LOOP

Sekvenčný diagram s fragmentom LOOP je prekladaný do OAL kódu dvojako. Ak fragment LOOP vo svojej podmienke obsahuje podmienku, ktorá začína slovom „*for*“, do OAL kódu je preložený ako cyklus *for each*. Ak sa v podmienke uvedené slovo ne nachádza, do OAL kódu je fragment LOOP preložený ako cyklus *while* s príslušnou podmienkou. Dodávame, že v našom riešení nekontrolujeme z hľadiska jazyka OAL syntax podmienky začínajúcej slovom „*for*“ vo fragmente typu LOOP, teda nekontrolujeme, či je podmienka začínajúca spomenutým slovom zapísaná korektnie ako cyklus *for each* v jazyku OAL.

Obrázok 3.7 obsahuje sekvenčný diagram s LOOP fragmentom. Tento sekvenčný diagram navyše obsahuje aj vnorený fragment, a to fragment OPT vnorený vo fragmente LOOP. Kód 3.8 je adekvátnym kódom pre tento sekvenčný diagram, pričom fragment LOOP bol v takomto prípade preložený ako cyklus *while*.



Obr. 3.7: Sekvenčný diagram s fragmentom typu LOOP.

Ukážka kódu 3.8: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.7.

```

class Class1
    method StartMethod()
        while (condition1)
            if (condition2)
                create object instance Class2_inst of Class2;
                Class2_inst.hello();
            end if;
        end while;
    end method;
end class;

class Class2
    method hello()
    end method;
end class;

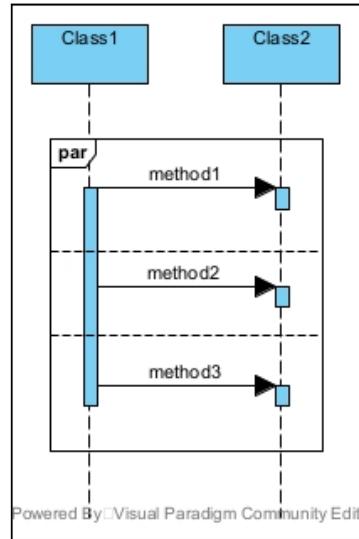
```

### 3.1.8 Konvertovanie sekvenčných diagramov s fragmentom typu PAR

Rozšírený jazyk OAL, predstavený v diplomovej práci študenta F. Nováka, umožňuje paralelné vykonávanie príkazov [18]. Preto je v našom riešení možné prekladať aj sekvenčné diagramy s fragmentom typu PAR. Fragment typu PAR je zložením podobný ako fragment ALT, s tým rozdielom, že neobsahuje vo svojich operandoch podmienky. Príklad sekvenčného diagramu s fragmentom PAR je na obrázku 3.8. OAL kód prislúchajúci tomuto diagramu je zobrazený v 3.9. OAL kód pripravujúci fragment PAR je zobrazený v 3.10. V kóde je zobrazená iba prvá metóda *StartMethod*.

Ako je možné v kóde 3.9 vidieť, v jazyku OAL sa paralelné vykonávanie kódu začína príkazom „*par*“ a končí príkazom „*end par*“. V rámci príkazu *par* sa potom vykonávajú vlákna, ktoré začínajú konštrukciou *thread* a končia príkazom „*end thread*“. Príkaz *par* zo sekvenčných diagramov prekladáme na základe fragmentu typu PAR. Jednotlivé vlákna v rámci príkazu PAR sú preložené na základe operandov fragmentu PAR. Diagram na obrázku 3.8 obsahuje fragment PAR s 3 operandmi, pričom každý operand obsahuje jednu správu. Vo výslednom kóde teda metóda *StartMethod* obsahuje príkaz

*par*, resp. paralelný blok s tromi vláknenami, pričom každé z týchto vlákin obsahuje dané volanie metódy triedy *Class2*.



Obr. 3.8: Sekvenčný diagram s fragmentom typu PAR.

Ukážka kódu 3.9: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.8.

---

```

class Class1
    method StartMethod()
        par
            thread
                create object instance Class2_inst of Class2;
                Class2_inst.method1();
            end thread;
            thread
                create object instance Class2_inst of Class2;
                Class2_inst.method2();
            end thread;
            thread
                create object instance Class2_inst of Class2;
                Class2_inst.method3();
            end thread;
        end par;
    end method;
end class;

```

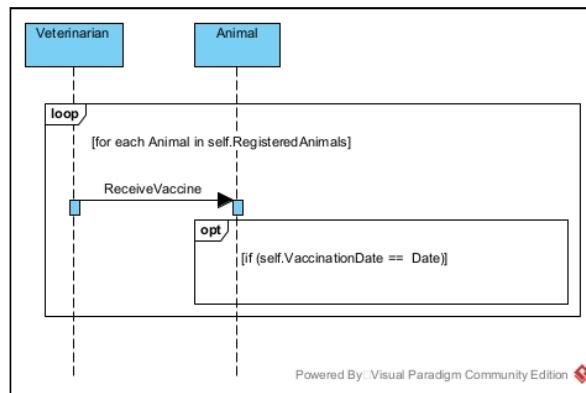
---

### 3.1.9 Konvertovanie sekvenčných diagramov s prázdnym operandom vo fragmente

Pri analyzovaní niektorých OAL kódov napísaných používateľom sme usúdili, že určitý sekvenčný diagram, ktorý má aspoň sčasti reprezentovať charakter a procesy daných kódov, musí obsahovať fragment s prázdnym operandom, teda operand, ktorý neobsahuje žiadne správy, ani iné fragmenty. Takáto situácia môže nastať napríklad vtedy, keď pôvodný OAL kód obsahuje určitú konštrukciu (príkaz *if*, cyklus *for each* a pod.), avšak telo tejto konštrukcie z viacerých príčin nie je možné priamočiaro zaznamenať v diagrame. Príklad takejto situácie je, keď telo príkazu obsahuje priradenie nejakej premennej, aritmetické operácie, priradenie prvku do zoznamu a pod.

Príklad sekvenčného diagramu s fragmentom s prázdnym operandom, resp. s prázdnym fragmentom typu OPT, sa nachádza na obrázku 3.9.

Pri konvertovaní operandov v sekvenčnom diagrame sa tieto prázdnne operandy pridajú ako daná konštrukcia OAL kódu do poslednej volanej, resp. vytvorennej metódy v kóde. V kóde 3.10, ktorý prislúcha diagramu na obrázku 3.9, sa zavolá v metóde *StartMethod* triedy *Veterinarian* metóda *ReceiveVaccine* triedy *Animal*. Metóda *ReceiveVaccine* sa v triede *Animal* aj vytvorí. Prázdný fragment OPT sa v sekvenčnom diagrame nachádza po správe *ReceiveVaccine*, a teda v kóde bol ako prázdný príkaz *if* priradený do poslednej vytvorennej metódy *ReceiveVaccine*.



Obr. 3.9: Sekvenčný diagram s fragmentom s prázdnym operandom.

#### Ukážka kódu 3.10: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.9.

---

```

class Veterinarian
method StartMethod():
    for each Animal in self.RegisteredAnimals
        create object instance Animal_inst of Animal;
        Animal_inst.ReceiveVaccine();
    end for;
end method;
end class;

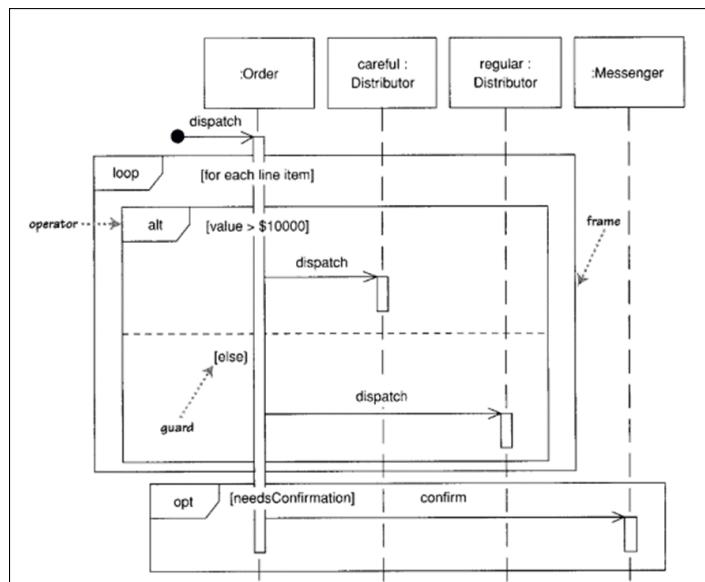
class Animal
    method ReceiveVaccine()
        if (self.VaccinationDate == Date)
            end if;
    end method
end class;

```

---

### 3.1.10 Konvertovanie komplexných sekvenčných diagramov

Na záver uvádzame príklad konverzie sekvenčného diagramu na obrázku 3.10 do OAL kódu znázornenom v 3.11. Jedná sa o sekvenčný diagram z knihy Martina Fowlera *UML distilled: a brief guide to the standard object modeling language* [19], zahrňujúci všetky prvky, ktoré sme spomínali v tejto kapitole v súvislosti s konverziou do kódu OAL. V prípade tohto sekvenčného diagramu je však fragment typu LOOP preložený ako cyklus *for each*. Prvá správa *dispatch* v sekvenčnom diagrame nie je v kóde zobrazená ako metóda *dispatch*, ale ako metóda *StartMethod* triedy *Order*.



Obr. 3.10: Komplexný sekvenčný diagram [19].

Ukážka kódu 3.11: Úkážka OAL kódu pre sekvenčný diagram na obrázku 3.10.

```

class Order
    method StartMethod()
        for each line item
            if (value > $10000)
                create object instance Careful_inst of Careful;
                Careful_inst.dispatch();
            else
                create object instance Regular_inst of Regular;
                Regular_inst.dispatch();
            end if;
        end for;
        if (needs confirmation)
            create object instance Messenger_inst of Messenger;
            Messenger_inst.confirm();
        end if;
    end method;
end class;

class Careful
    method Dispatch()
    end method;
end class;

class Regular
    method Dispatch()
    end method;
end class;

class Messenger
    method Confirm()
    end method;
end class;

```

## 3.2 Potreba rekurzie pri generovaní OAL kódu

Sekvenčné diagramy môžu obsahovať fragmenty, ktoré v sebe môžu obsahovať ďalšie fragmenty. Príklady takýchto sekvenčných sme uviedli na obrázkoch 3.7 a 3.10. Z hľadiska konverzie takýchto sekvenčných diagramov do OAL kódu bolo v rámci procesu

ich prekladania nutné využiť rekurziu. Vďaka rekurzii je možné jednotlivé vnorené fragmenty spracovať samostatne. Po ich úspešnom preložení do OAL kódu nastáva proces vynorenia sa z rekurzie a kód vytvorený v rekurzii je tak pripojený do spoločného výsledku konverzie sekvenčného diagramu do zdrojového kódu.

### 3.3 Identifikácia nekorektných sekvenčných diagramov

Pri analyzovaní prístupov ku konverzii sekvenčných diagramov do zdrojového kódu sme zistili, že niektoré sekvenčné diagramy nie je možné preložiť do príslušného OAL kódu kvôli rôznym anomáliám, ktoré tieto sekvenčné diagramy obsahujú. Rozhodli sme sa preto, že pri generovaní zdrojového kódu budeme validitu sekvenčných diagramov kontrolovať. Pre sekvenčné diagramy, ktoré sú nekorektné, OAL kód nevygenerujeme a prípadného používateľa nášho riešenia o nekorektnosti daného sekvenčného diagramu informujeme.

Správnosť sekvenčných diagramov kontrolujeme na základe rôznych kritérií. Preto v tejto kapitole predstavujeme definované pravidlá, ktoré majú validné sekvenčné diagramy splňať.

Prvé pravidlo sme definovali nasledovne:

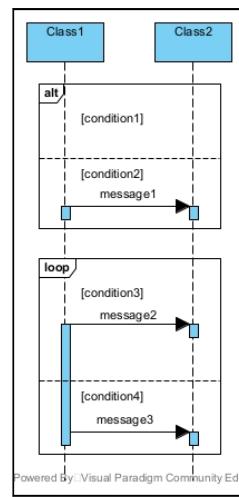
**Pravidlo 1.** *Ak sekvenčný diagram obsahuje fragment typu LOOP, tento fragment nesmie mať viac ako jeden operand, resp. viac ako jednu vetvu.*

V sekvenčnom diagrame môže nastať aj situácia, kde fragment typu ALT má len jeden operand, alebo fragment OPT má viac ako jeden operand. Hoci z pohľadu štandardu UML sekvenčné diagramy s takýmito fragmentmi nie sú korektné, takéto sekvenčné diagramy je naším riešením možné preložiť do OAL kódu, a to ako konštrukciu *if*, *elif*, alebo *else*.

Ako druhé pravidlo, ktoré majú splňať validné sekvenčné diagramy, sme definovali nasledovné pravidlo:

**Pravidlo 2.** *Ak sekvenčný diagram obsahuje fragment s prázdnym operandom, teda s takým operandom, ktorý neobsahuje žiadne elementy (správy, či ďalšie fragmenty), tento operand, resp. fragment sa v sekvenčnom diagrame nesmie vyskytovať ako prvý element.*

Príklad sekvenčného diagramu, ktorý nespĺňa pravidlo 2, je na obrázku 3.11. Tento diagram obsahuje ako prvý element fragment typu ALT, ktorého prvý operand (s podmienkou *condition1*) je prázdnym operandom. Diagram na obrázku 3.11 taktiež nespĺňa pravidlo 1, keďže obsahuje fragment typu LOOP s dvoma operandmi.



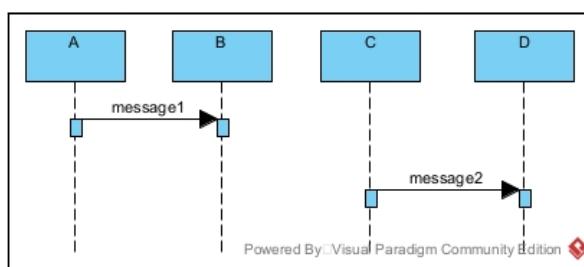
Obr. 3.11: Sekvenčný diagram nesplňajúci pravidlo 1 a 2.

Ďalšie pravidlo, ktoré majú splňať všetky sekvenčné diagramy na to, aby boli v našom riešení validné, sme definovali takto:

**Pravidlo 3.** Ak  $M$  je prvá správa odoslaná z lifeline  $A$  a zároveň  $M$  nie je prvou správou sekvenčného diagramu, potom musí existovať nejaká správa  $N$ , ktorá je v sekvenčnom diagrame vyššie ako správa  $M$  a ktorá smeruje do lifeline  $A$ .

Na obrázku 3.12 je diagram, ktorý nespĺňa toto pravidlo. Ako je možné vidieť, správa **message2** začína v lifeline **C**, do ktorej predtým nesmerovala žiadna iná správa. Pri transformovaní do OAL kódu by sa správa **message2** v kóde ako volanie metódy a ani ako samotná metóda nezobrazila.

Pre korektné transformovanie správ do OAL kódu je potrebné zachovať kontinuitu medzi jednotlivými správami, aby sa prvou správou, resp. prvým volaním postupne v kóde zobrazili všetky správy sekvenčného diagramu, a to ako volanie metód a aj ako metódy samotné. Ako je možné vidieť na obrázku 3.12, kontinuita medzi správami **message1** a **message2** neexistuje, a preto takýto diagram ani nie je možné správne preložiť do OAL kódu.



Obr. 3.12: Sekvenčný diagram nesplňajúci pravidlo 3.

Komplexnejšie pravidlo, ktoré v našom prístupe korektné sekvenčné diagramy splňajú, je pravidlo:

**Pravidlo 4.** Nech  $M$  je správa odoslaná z lifeline  $A$ , a  $N$  je správa smerujúca do lifeline  $A$ , pričom  $M$  je v sekvenčnom diagrame nižšie ako  $N$  a zároveň medzi  $M$  a  $N$  žiadna iná správa nebola odoslaná z lifeline  $A$  a ani nesmeruje do lifeline  $A$ . Potom nesmie existovať žiadna iná správa  $O$ , nachádzajúca sa v sekvenčnom diagrame medzi správami  $M$  a  $N$  a zároveň odoslaná z lifeline, ktorá je v sekvenčnom diagrame naľavo od lifeline  $A$ .

Takéto komplexnejšie pravidlo si vysvetlíme na sekvenčnom diagrame na obrázku 3.13, u ktorého je pravidlo 4 porušené. V sekvenčnom diagrame na obrázku 3.13 správa  $message1$  smeruje do lifeline  $B$  a správa  $message3$  smeruje zas z lifeline  $B$ , čím je dodržané pravidlo 3. Medzi správou  $message1$  a  $message3$  je správa  $message2$ , ktorá smeruje z lifeline  $A$ , pričom  $A$  je v tomto sekvenčnom diagrame pred lifeline  $B$ .

Kód, ktorý by potenciálne mohol reprezentovať tento sekvenčný diagram, je kód 3.12.

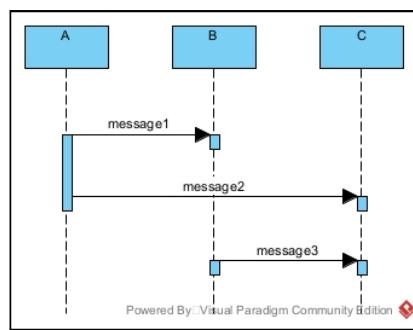
V kóde 3.12 sa v metóde *StartMethod* zavolá metóda  $message1$  triedy  $B$  a potom sa zavolá metóda  $message2$  triedy  $C$ . Správa  $message3$  v sekvenčnom diagrame je pretransformovaná na volanie metódy  $message3$  triedy  $C$ , pričom toto volanie sa pridá do kódu poslednej, resp. prvej metódy triedy  $B$ , ktorou je metóda  $message1$ .

Z metódy  $message1$  triedy  $B$  je teda volaná metóda  $message3$  triedy  $C$ . Zavolaním  $message1$  v prvej metóde *StartMethod* tok volaní zároveň prejde do metódy  $message1$ . V metóde  $message1$  sa zavolá metóda  $message3$ .

Metóda  $message3$  triedy  $C$  je prázdna a metóda  $message1$  triedy  $B$  už neobsahuje žiadne ďalšie volania. Preto sa v toku volaní pokračuje v metóde *StartMethod*.

Ako je možné si všimnúť, poradie toku volaní je odlišné ako poradie správ v sekvenčnom diagrame. Konkrétnie metóda  $message3$  sa v toku volaní zavolá pred metódou  $message2$ , čo je v sekvenčnom diagrame na obrázku 3.13 zobrazené presne naopak. A teda kód 3.12 nezodpovedá tomuto sekvenčnému diagramu.

Môžeme konštatovať, že tento diagram ani nie je možné preložiť do OAL kódu tak, aby daný OAL kód zodpovedal presne takémuto sekvenčnému diagramu. Kód by bolo možné v OAL kóde korektne zobraziť, ak by správa  $message2$  končila v lifeline  $B$  alebo ak by  $message2$  v diagrame bola zobrazená po správe  $message3$ .



Obr. 3.13: Sekvenčný diagram nesplňajúci pravidlo 4.

Ukážka kódu 3.12: Úkážka potenciálneho OAL kódu pre sekvenčný diagram na obrázku 3.13.

---

```

class A
  method StartMethod()
    create object instance B_inst of B;
    B_inst.Message1();
    create object instance C_inst of C;
    C_inst.Message2();
  end method;
end class;

class B
  method Message1()
    create object instance C_inst of C;
    C_inst.Message3();
  end method;
end class;

class C
  method Message2()
  end method;

  method Message3()
  end method;
end class;
  
```

---

Ďalšie pravidlo určujúce korektnosť sekvenčných diagramov sa týka sekvenčných diagramov s fragmentom typu ALT. Toto pravidlo je definované takto:

**Pravidlo 5.** Vo všetkých operandoch, resp. vetvách fragmentu typu ALT musí prvá správa začínať v lifeline, v ktorej začínala prvá správa v prvej vetve fragmentu.

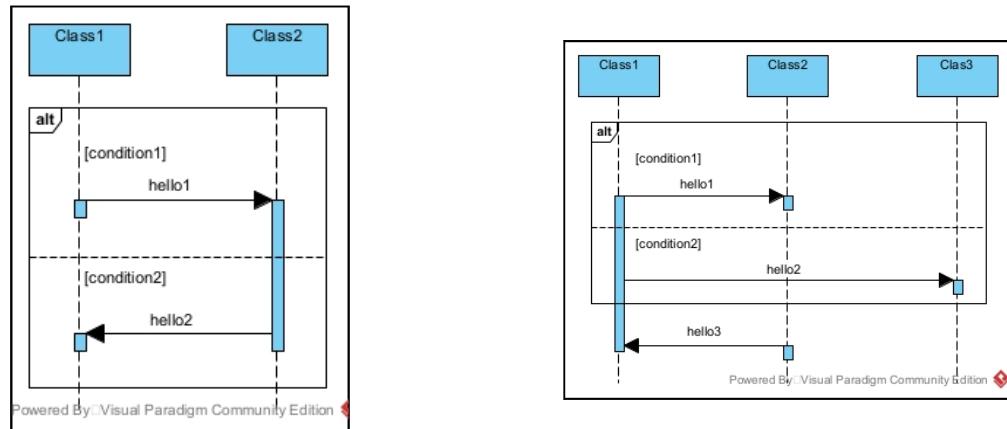
Príklad, kedy toho pravidlo nie je splnené, je uvedený na sekvenčnom diagrame zobrazenom na 3.14. Tento sekvenčný diagram nesplňa uvedené pravidlo, keďže v druhej vetve fragmentu ALT prvá správa začína v druhej lifeline, pričom v prvej vetve prvá správa začínala v prvej lifeline. Takýto sekvenčný diagram by teda nebolo možné preložiť do OAL kódu, nakoľko by pri konverzii do OAL kódu v druhej vetve príkazu *if* (resp. príkazu *elif*) nebolo jasné, z akej metódy má tok volaní pokračovať.

Posledné pravidlo validácie sekvenčných diagramov sme definovali nasledovne:

**Pravidlo 6.** Prvá správa po fragmente nesmie začínať v takej lifeline, v ktorej končila posledná správa niektoľ z vetiev tohto fragmentu. Správa po fragmente musí taktiež začínať v takej lifeline, aby na základe ostatných správ (aj tých pred fragmentom) boli splnené podmienky pravidiel 3 a 4.

Príklad sekvenčného diagramu, ktorý nesplňa pravidlo 6, je zobrazený na obrázku 3.15. Z obrázku vidíme, že správa *hello1* končí v lifeline *Class2* a správa *hello2* končí v lifeline *Class3*.

Prvá správa po fragmente *hello3* začína v *Class2*. Správa *hello1*, končiacia v lifeline *Class2*, je prvou a zároveň poslednou správou v rámci prvej vetvy fragmentu ALT. Práve správa *hello1* spolu so správou *hello3* predstavujú pri prekladaní do zdrojového kódu problém, ktorý sme sa snažili vymedziť pravidlom 6. Keďže prvá správa po fragmente, správa *hello3*, začína v *Class2*, v ktorej končila posledná správa prvej vetvy ALT fragmentu (teda správa *hello1*), takýto sekvenčný diagram by nebolo možné preložiť do zdrojového kódu. Ak by totiž bola splnená podmienka *condition2*, tok volaní by v OAL kóde pokračoval metódou *hello2* triedy *Class3* a nebolo by možné pokračovať v toku volaní metódou *hello3* triedy *Class2*, ako je to v sekvenčnom diagrame zobrazené.



Obr. 3.14: Sekvenčný diagram nespĺňajúci pravidlo 5.

Obr. 3.15: Sekvenčný diagram nespĺňajúci pravidlo 6.

# Kapitola 4

## Evaluácia

Naším riešením a prístupom k problematike transformovania dynamických modelov do zdrojového sme úspešne dosiahli generovanie OAL kódu zo sekvenčných diagramov. Pre zistenie relevancie nami navrhnutého riešenia však získané výsledky budeme evaluovať. Evaluáciu vykonáme pomocou metodiky **relevance judgment** (v preklade z angličtiny „*hodnotenie relevancie*“). Hodnotenie relevancie uskutočníme konkrétnie pomocou metrík **precision** (v preklade „*presnosť*“) a **recall** (v preklade ako „*úplnosť*“).

Relevance judgment, precision a recall sú jednými zo základných konceptov na hodnotenie efektívnosti rôznych systémov v špecifických oblastiach, ako je napríklad informačné vyhľadávanie alebo spracovanie dát. Hodnotenie relevancie je proces hodnotenia toho, či sú určité položky (napr. dokumenty, obrázky, alebo informácie) relevantné vo vzťahu k zadanej otázke alebo požiadavke [20].

**Precision** (presnosť) je metrika, ktorá meria presnosť systému pri identifikácii relevantných položiek. Je definovaná ako podiel relevantných položiek medzi načítanými položkami. Metrika s názvom **recall** (úplnosť) hodnotí, aký veľký podiel všetkých relevantných informácií systém dokázal nájsť a je definovaná ako podiel počtu správne identifikovaných relevantných položiek ku všetkým relevantným položkám [20].

Konkrétnie presnosť vieme určiť pomocou vzorca

$$p = \frac{TP}{TP + FP}$$

a úplnosť zas pomocou vzorca

$$r = \frac{TP}{TP + FN},$$

pričom  $TP$  je skratka pre „*true positive*“ (v preklade ako „skutočne pozitívny“),  $FN$  je skratka pre „*false negative*“ (teda z angličtiny „falošne negatívny“) a  $FP$  pre „*false positive*“ (v preklade „falošne pozitívny“) [21].

Napríklad v článku [20] boli hodnotenie relevancie spolu s presnosťou a úplnosťou využívané pri porovnaní výkonnosti dvoch vyhľadávacích algoritmov pomocou sady

preddefinovaných *úsudkov o relevancii* (v angličtine „*relevance judgments*“) na vyhodnotenie toho, ktorý algoritmus efektívnejšie vyhľadal relevantné dokumenty k určitým dopytom.

## 4.1 Postup evaluácie generovania zdrojového kódu

V našom prípade evaluáciu vykonáme tak, že očakávaný zdrojový OAL kód porovnáme s OAL kódom vygenerovaným naším riešením. Očakávaný OAL kód bude vopred pripravený a funkčný, využívaný v nástroji *AnimArch*, alebo taký kód, ktorý podľa nášho úsudku reprezentuje konkrétny sekvenčný diagram. V prípade vopred definovaného očakávaného OAL kódu najskôr pripravíme sekvenčný diagram, ktorý čo najvernejšie odráža daný kód. Tieto OAL kódy budú teda v procese evaluácie naše *úsudky o relevancii* (*relevance judgments*). Našou implementáciou na základe pripraveného sekvenčného diagramu vygenerujeme zdrojový kód.

Porovnanie uskutočníme tak, že v očakávanom OAL kóde, ako aj vo vygenerovanom OAL kóde spočítame počet daných elementov zdrojového kódu. Tento počet následne porovnáme pomocou spomínaných metrík *presnosť* a *úplnosť*.

Čo sa týka určenia hodnôt pre presnosť a úplnosť, ako falošne pozitívne (FP) budeme označovať tie elementy v generovanom OAL kóde, ktoré sú oproti očakávanému OAL kódu v generovanom kóde navyše. Ako falošne negatívne elementy (FN) budeme zas považovať tie elementy, ktoré sa v generovanom OAL kóde nenachádzajú, zatiaľ čo v predpokladanom OAL kóde áno.

Kvôli rozsiahlosti niektorých OAL kódov v nasledujúcich častiach zobrazujeme iba časti týchto kódov. Všetky OAL kódy sa nachádzajú v elektronickej prílohe práce.

Konkrétnie budeme v spomenutom počte uvažovať tieto elementy zdrojového kódu:

- triedy
- metódy
- vytvorenie inštancie
- cykly (*for each* a *while*)
- paralelné bloky (konštrukcia *par*)
- vlákna (konštrukcia *thread*)
- podmienky cyklov
- príkazy (*if*, *elif* a *else*)
- volania metód

- ukončenia cyklov, príkazov, vlákien a paralelných blokov (konštrukcia *end*)
- vytváranie zoznamov
- pridávanie prvkov do zoznamu
- odstraňovanie prvkov zo zoznamu
- priradenie hodnoty do premennej
- atribúty metód a tried

Niektoré z uvedených elementov OAL kódu v našej implementácii nedefinujeme, keďže na základe sekvenčných diagramov, vytvorených nástrojom *SQD\_Tunder*, ich ani nie je možné v danom diagrame zobraziť. Konkrétnie nedefinujeme v našom riešení vytváranie zoznamov, pridávanie a odstraňovanie prvkov do a zo zoznamu, priradenie hodnoty do premennej a použitie atribútov.

Pri porovnávaní elementov OAL kódu zanedbávame počet medzier a prázdnych riadkov, ako aj rozdielne názvy inštancií, prípadne názvy prvých metód (korešpondujúcich s prvou správou v sekvenčnom diagrame). Taktiež zanedbávame vytváranie konštruktorov a poradie definovania tried a metód (nie však poradie volania metód).

V predpokladanom, ako aj v generovanom kóde budeme vlastné správy považovať za štandardné metódy (viď kapitolu 3.1.4).

## 4.2 Evaluácia vygenerovaného zdrojového kódu

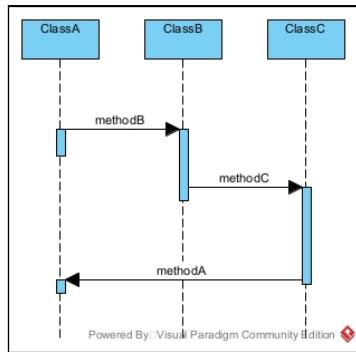
V tejto podkapitole postupne predstavíme jednotlivé prípady evaluácie nášho riešenia na základe rôznych sekvenčných diagramov, resp. OAL kódov.

### 4.2.1 Prvý prípad evaluácie (kontinuita správ)

Prvú evaluáciu vykonáme na základe porovnania vygenerovaného OAL kódu s OAL kódom, ktorý sme napísali na základe jednoduchého sekvenčného diagramu. Tento diagram sa nachádza na obrázku 4.1.

Vidíme, že v tomto diagrame jednotlivé správy nadvádzajú kontinuálne na seba. Očakávali sme, že v OAL kóde majú preto takto na seba nadvázovať volania, čo je aj teda zobrazené v nami definovanom OAL kóde na obrázku 4.2. OAL kód vygenerovaný našou implementáciou sa nachádza na obrázku 4.3.

Ako je možné vidieť, medzi očakávaným OAL kódom na obrázku 4.2 a vygenerovaným OAL kódom na obrázku 4.3 nie sú v podstate žiadne rozdiely. Jediný rozdiel spočíva v názvoch inštancií, čo však, ako sme uviedli, v porovnaní elementov OAL



Obr. 4.1: Sekvenčný diagram prvého prípadu evaluácie.

```

class ClassA
    method StartMethod()
        create object instance ClassB_inst of ClassB;
        ClassB_inst.methodB();
    end method;

    method methodA()
    end method;
end class;

class ClassB
    method methodB()
        create object instance ClassC_inst of ClassC;
        ClassC_inst.methodC();
    end method;
end class;

class ClassC
    method methodC()
        create object instance ClassA_inst of ClassA;
        ClassA_inst.methodA();
    end method;
end class;

```

```

class ClassA
    method StartMethod()
        create object instance ClassB_inst of ClassB;
        ClassB_inst.methodB();
    end method;

    method methodA()
    end method;
end class;

class ClassB
    method methodB()
        create object instance ClassC_inst of ClassC;
        ClassC_inst.methodC();
    end method;
end class;

class ClassC
    method methodC()
        create object instance ClassA_inst of ClassA;
        ClassA_inst.methodA();
    end method;
end class;

```

Obr. 4.2: Ukážka očakávaného kódu 1. prípadu evaluácie.

Obr. 4.3: Ukážka generovaného kódu 1. prípadu evaluácie.

kódov zanedbávame. Porovnanie počtu elementov medzi týmito dvoma kódmi sa nachádza v tabuľke 4.1. V tabuľke neuvádzame prvky OAL kódu, ktoré sa nenachádzajú ani v jednom z týchto kódov, ako napríklad cykly.

Tabuľka 4.1: Porovnanie počtu elementov kódu v 1. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	3	3
Metóda	4	4
Vytvorenie inštancie	3	3
Volanie metódy	3	3
<b>Všetky elementy</b>	<b>13</b>	<b>13</b>

Počet elementov vo všetkých kategóriách v prípade očakávaného kódu, ako aj v prípade generovaného OAL kódu, je rovnaký. Preto počet falošne pozitívnych a falošne negatívnych elementov je rovný nule.

A teda v prípade tohto prípadu evaluácie presnosť je rovná počtu

$$\frac{13}{13 + 0} = 1$$

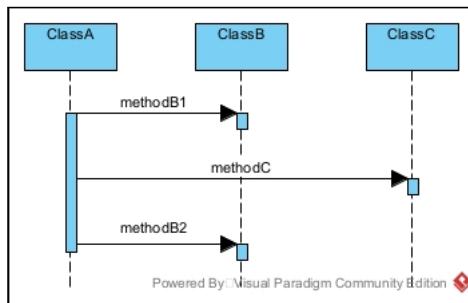
a úplnosť zas rovná počtu:

$$\frac{13}{13+0} = 1$$

Metriky presnosť a úplnosť ukazujú že naša metóda je pre tento prípad generovania OAL kódu zo sekvenčného diagramu úplne relevantná.

#### 4.2.2 Druhý prípad evaluácie (postupnosť správ)

Druhý prípad evaluácie taktiež vykonáme na základe sekvenčného diagramu, ktorý sa nachádza na obrázku 4.4.



Obr. 4.4: Sekvenčný diagram druhého prípadu evaluácie.

Ako je možné vidieť, aj v tomto prípade máme jednoduchší sekvenčný diagram, v ktorom správy postupne vychádzajú iba z prvej lifeline. Preto sme predpokladali, že v OAL kóde budú volania metód vykonané iba v rámci prvej metódy prvej triedy. Nami predpokladaný OAL kód sa nachádza na obrázku 4.5. OAL kód vygenerovaný našou implementáciou sa zas nachádza na obrázku 4.6.

```
class ClassA
    method StartMethod()
        create object instance inst_B of ClassB;
        inst_B.methodB1();
        create object instance inst_C of ClassC;
        inst_C.methodC();
        inst_B.methodB2();
    end method;
end class;

class ClassB
    method methodB1()
    end method;

    method methodB2()
    end method;
end class;

class ClassC
    method methodC()
    end method;
end class;
```

```
class ClassA
    method StartMethod()
        create object instance ClassB_inst of ClassB;
        ClassB_inst.methodB1();
        create object instance ClassC_inst of ClassC;
        ClassC_inst.methodC();
        ClassB_inst.methodB2();
    end method;
end class;

class ClassB
    method methodB1()
    end method;

    method methodB2()
    end method;
end class;

class ClassC
    method methodC()
    end method;
end class;
```

Obr. 4.5: Ukážka očakávaného kódu 2. prípadu evaluácie.

Obr. 4.6: Ukážka generovaného kódu 2. prípadu evaluácie.

OAL kódy z obrázkov sú taktiež, ako v minulom prípade, v podstate totožné a jediný rozdiel v nich spočíva opäť iba v rozdielnych názvoch inštancií tried. Počty elementov jazyka OAL pre tieto 2 kódy sa nachádzajú v tabuľke 4.2.

Tabuľka 4.2: Porovnanie počtu elementov kódu v 2. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	3	3
Metóda	4	4
Vytvorenie inštancie	2	2
Volanie metódy	3	3
<b>Všetky elementy</b>	<b>12</b>	<b>12</b>

Počty elementov v oboch prípadoch OAL kódu sú rovnaké, čo znamená, že presnosť je rovná počtu

$$\frac{12}{12+0} = 1$$

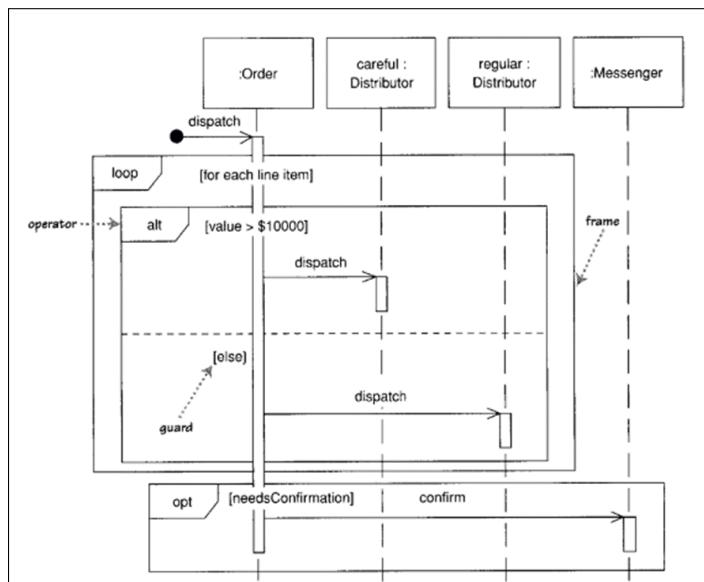
a úplnosť zas rovná počtu:

$$\frac{12}{12+0} = 1$$

Presnosť aj úplnosť sú v tomto prípade evaluácie opäť rovné číslu 1, čo znamená, že pre typy sekvenčných diagramov, ako je sekvenčný diagram na obrázku 4.4, je naša metóda veľmi relevantná.

#### 4.2.3 Tretí prípad evaluácie (komplexnejší sekvenčný diagram)

V tomto prípade evaluácie bude pre nás tvoriť základ sekvenčný diagram z knihy *UML distilled: a brief guide to the standard object modeling language* od M. Fowlera [19]. Tento sekvenčný diagram je zobrazený na obrázku 4.7. Kód, ktorý podľa nášho úsudku reprezentuje daný sekvenčný diagram, sa nachádza na obrázku 4.8. Generovaný kód je zas znázornený na obrázku 4.9.



Obr. 4.7: Sekvenčný diagram tretieho prípadu evaluácie [19].

```

class Order
    int[] items;
    bool needsConfirmation;
    method StartMethod()
        for each line_item in items
            if (line_item > 10000)
                create object instance inst_Careful of Careful;
                Careful_inst.dispatch();
            else
                create object instance inst-Regular of Regular;
                Regular_inst.dispatch();
            end if;
        end for;
        if (needsConfirmation)
            create object instance inst_Messenger of Messenger;
            Messenger_inst.confirm();
        end if;
    end method;
end class;

class Careful
    method dispatch()
    end method;
end class;

class Regular
    method dispatch()
    end method;
end class;

class Messenger
    method confirm()
    end method;
end class;

```

Obr. 4.8: Ukážka očakávaného kódu 3. prípadu evaluácie.

```

class Order
    method StartMethod()
        for each line item
            if (value > $10000)
                create object instance Careful_inst of Careful;
                Careful_inst.dispatch();
            else
                create object instance Regular_inst of Regular;
                Regular_inst.dispatch();
            end if;
        end for;
        if (needs confirmation)
            create object instance Messenger_inst of Messenger;
            Messenger_inst.confirm();
        end if;
    end method;
end class;

class Careful
    method dispatch()
    end method;
end class;

class Regular
    method dispatch()
    end method;
end class;

class Messenger
    method confirm()
    end method;
end class;

```

Obr. 4.9: Ukážka generovaného kódu 3. prípadu evaluácie.

Ako je možné vidieť, zobrazené kódy sú v tomto prípade mierne odlišné. V nami definovanom kóde na obrázku 4.8, znázorňujúcim sekvenčný diagram na 4.7, sme doplnili atribúty triedy *Order*. Podľa konvencii jazyka OAL by sme mali do kódu pridať aj konštruktor, v ktorom by sme inicializovali tieto atribúty. *V tomto, ale aj v ostatných ukážkach kódov pre lepšiu prehľadnosť konštruktory nezobrazujeme.* V kóde na obrázku 4.8 je rozdiel oproti obrázku 4.9 aj v cykle *for each* a v podmienkach *if* príkazov. V nami napísanom OAL kóde sme sa snažili zabezpečiť funkčnosť tohto kódu. Preto sme do kódu pridali atribúty a primerane sme upravili *for each* cyklus a podmienky *if* príkazov tak, aby využívali dané atribúty.

V našej metóde generovania zdrojového kódu zo sekvenčných diagramov však nekontrolujeme správnosť *for each* cyklu, podmienku cyklu *while*, ako aj podmienky príkazov *if* a *else if*. Kontrola uvedených elementov OAL kódu by totiž značne zasahovala nad rámec cieľov našej práce, keďže takáto kontrola by vyžadovala aj úpravu vytvárania sekvenčných diagramov. A preto v generovanom kóde na obrázku 4.9 sa *for each* cyklus a podmienky *if* príkazov nachádzajú tak, ako sú zobrazené v sekvenčnom diagram na obrázku 4.7.

Pri počítaní hodnoty metrík presnosť a úplnosť však uvedené rozdiely v kóde nezohľadňujeme. V rámci daných metrík berieme do úvahy iba počet konkrétnych prvkov zdrojového kódu. Počty jednotlivých prvkov obidvoch OAL kódov, znázornených na obrázkoch 4.8 a 4.9, sa nachádzajú v tabuľke 4.3. V tejto tabuľke už zobrazujeme aj počty triednych atribútov a jednotlivých cyklov, keďže tieto elementy sa nachádzajú aspoň v jednom z uvedených kódov.

Tabuľka 4.3: Porovnanie počtu elementov kódu v 3. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	4	4
Metóda	4	4
Vytvorenie inštancie	3	3
Volanie metódy	3	3
Cyklus <i>for each</i>	1	1
Príkaz <i>if</i>	2	2
Príkaz <i>else</i>	1	1
Konštrukcia <i>end</i>	3	3
Triedne atribúty	2	0
<b>Všetky elementy</b>	<b>23</b>	<b>21</b>

Ked'že v generovanom kóde chýbajú triedne atribúty, počet elementov v takomto kóde je iba mierne odlišný od očakávaného kódu, a to konkrétnie o 2 elementy. Počet správne určených prvkov OAL kódu je 21. Nakoľko žiadnen prvak v OAL kóde neboli vygenerované navyše, teda žiadnen prvak nie je falošne pozitívny, presnosť bude opäť rovná číslu 1, čiže:

$$\frac{21}{21 + 0} = 1$$

V generovanom kóde vyšli 2 prvky ako falošne negatívne (chýbajúce triedne elementy), preto úplnosť nám v tomto prípade evaluácie vychádza ako:

$$\frac{21}{21 + 2} = \frac{21}{23} \approx 0.91$$

Presnosť vyšla v tomto prípade tak isto, ako v predchádzajúcich prípadoch, čiže podľa tejto metriky je naša metóda veľmi relevantná. Úplnosť s hodnotou približne 0.91 vyšla oproti predchádzajúcim prípadom mierne horsie, avšak s takýmto výsledkom úplnosti je pri sekvenčnom diagrame na obrázku naše riešenie stále pomerne relevantné.

Samozrejme, tieto metriky neodrážajú rozdielnosť cyklu *for each* a podmienok *if* príkazov. Kvôli priamočiaremu použitiu *for each* cyklu a podmienok príkazov *if* zo sekvenčného diagramu je generovaný kód z hľadiska syntaxe jazyka OAL nekorektný, čo z neho robí výraznejšie rozdielny kód oproti nášmu úsudku o relevancii (relevance judgment) - teda nami predpokladanému OAL kódu. A preto, hoci metriky presnosť a úplnosť ukazujú, že generovaný kód je pomerne presný a kompletný, nesprávnosť v generovanom zdrojovom kóde znižuje jeho praktickú využiteľnosť.

#### 4.2.4 Štvrtý prípad evaluácie (Observer)

Oproti predchádzajúcim je tento prípad evaluácie mierne rozdielny. Odlišnosť na rozdiel od predchádzajúcich prípadov spočíva v predpokladom OAL kóde, teda v našom úsudku o relevancii. Očakávaný OAL kód je v tomto prípade OAL kód využívaný pri

vykonávaní animácií objektov v programe *AnimArch*. Daný OAL kód predstavoval modelový prípad vakcinácie zvierat a bol napísaný v rámci návrhového vzoru *Observer*. Bol spomínaný taktiež v diplomovej práci [18] z roku 2022. Ukážka tohto kódu sa nachádza na obrázku 4.10. Kvôli rozsiahlosti tohto kódu je na obrázku 4.10 zobrazená iba jeho časť.

```

class Client
    method StartCase()
        create object instance Vet of Veterinarian;
        create list Vet.RegisteredAnimals of Observer;
        create object instance Husky of Dog;
        Husky.Veterinarian = Vet;
        Husky.IsVaccinated = FALSE;
        Vet.Register(Husky, "1.4.2022");
        create object instance PersianCat of Cat;
        PersianCat.Veterinarian = Vet;
        PersianCat.IsVaccinated = FALSE;
        Vet.Register(PersianCat, "2.4.2022");
        Vet.SetDate("1.4.2022");
        Vet.SetDate("2.4.2022");
    end method;
end class;

class Veterinarian
    string CurrentDate;
    Observer[] RegisteredAnimals;
    method Register(Observer Obs, string Date)
        add Obs to self.RegisteredAnimals;
        Obs.SetVaccinationDate(Date);
    end method;

    method Unregister(Observer Obs)
        remove Obs from self.RegisteredAnimals;
        Obs.SetVaccinationDate(UNDEFINED);
    end method;

    method VaccinateAnimals()
        for each Animal in self.RegisteredAnimals
            Animal.ReceiveVaccine(self.CurrentDate);
        end for;
    end method;

    method SetDate(string Date)
        self.CurrentDate = Date;
        self.VaccinateAnimals();
    end method;
end class;

class Dog
    Subject Veterinarian;
    bool IsVaccinated;
    string Name;
    string VaccinationDate;
    method ReceiveVaccine(string Date)
        if (VaccinationDate == Date)
            self.IsVaccinated = TRUE;
        end if;
    end method;

    method SetVaccinationDate(string Date)
        self.VaccinationDate = Date;
    end method;
end class;

class Cat
    Subject Veterinarian;
    bool IsVaccinated;
    string Name;
    string VaccinationDate;

```

```

class Client
method FirstMethod()
    create object instance Veterinarian_inst of Veterinarian;
    Veterinarian_inst.Register();
    Veterinarian_inst.Register();
    Veterinarian_inst.SetDate();
    Veterinarian_inst.SetDate();
end method;
end class;

class Veterinarian
    method Register()
        create object instance Animal_inst of Animal;
        Animal_inst.SetVaccinationDate();
    end method;

    method SetDate()
        self.VaccinateAnimals();
    end method;

    method VaccinateAnimals()
        for each Animal in self.RegisteredAnimals
            create object instance Animal_inst of Animal;
            Animal_inst.ReceiveVaccine();
        end for;
    end method;
end class;

class Animal
    method SetVaccinationDate()
    end method;

    method ReceiveVaccine()
        if (self.VaccinationDate == Date)
            self.VaccinationDate = Date;
        end if;
    end method;
end class;

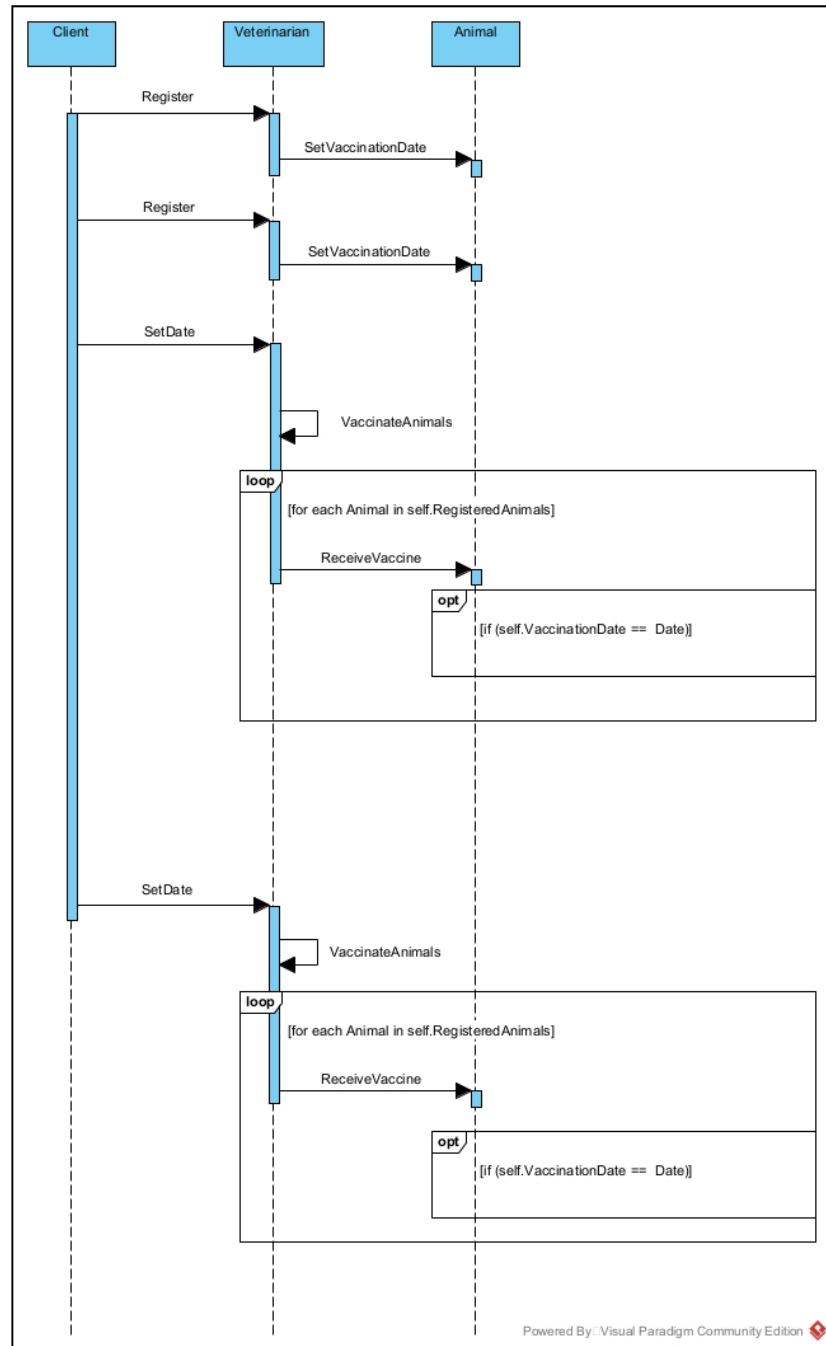
```

Obr. 4.11: Ukážka generovaného kódu 4. prípadu evaluácie.

Obr. 4.10: Ukážka očakávaného kódu 4. prípadu evaluácie.

Nakoľko východiskový OAL kód sme už mali určený, pre realizovanie generovania OAL kódu našou implementáciou sme museli vytvoriť sekvenčný diagram, ktorý aspoň sčasti odráža charakter východiskového OAL kódu. Daný sekvenčný diagram sa nachá-

dza na obrázku 4.12. V tomto sekvenčnom diagrame napríklad namiesto objektov *Cat* a *Dog*, ktoré sa nachádzajú v pôvodnom kóde, uvádzame iba triedu *Animal* (triedy *Cat* a *Dog* boli v skutočnosti v kóde totožné). V tomto sekvenčnom diagrame sa nachádza aj prázdný fragment, konkrétnie prázdný fragment typu OPT. V kóde príkaz *if*, ktorý má daný prázdný fragment zobrazovať, obsahoval priradenie hodnoty do triednej premennej. Keďže v sekvenčných diagramoch vytvorenými programom *SQD\_Tunder* priradenie premennej nemáme ako zobraziť, vo výslednom sekvenčnom diagrame môžeme znázorniť iba prázdný fragment typu OPT.



Obr. 4.12: Sekvenčný diagram štvrtého prípadu evaluácie.

Transformovaný OAL kód zo sekvenčného diagramu na obrázku 4.12 je zobrazený na obrázku 4.11. Ako je možné vidieť, oproti pôvodnému kódu na obrázku 4.10 má generovaný OAL kód značne menej elementov. Komparácia počtu elementov pôvodného a generovaného OAL kódu sa nachádza v tabuľke 4.4.

Tabuľka 4.4: Porovnanie počtu elementov kódu v 4. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	4	3
Metóda	9	6
Vytvorenie inštancie	3	3
Volanie metódy	8	7
Cyklus <i>for each</i>	1	1
Príkaz <i>if</i>	2	1
Konštrukcia <i>end</i>	3	2
Vytvorenie zoznamu	1	0
Pridanie prvku do zoznamu	1	0
Odstránenie prvku zo zoznamu	1	0
Triedne atribúty	4	0
Parametre metód	8	0
Priadenie hodnoty do atribútu	9	0
<b>Všetky elementy</b>	<b>54</b>	<b>23</b>

Naša metóda sa aj v tomto prípade evaluácie ukázala ako veľmi presná, nakoľko v generovanom kóde neboli vytvorené žiadne prvky navyše (falošne pozitívne prvky). Presnosť má teda hodnotu 1, nakoľko:

$$\frac{23}{23 + 0} = 1$$

Čo sa týka metriky úplnosť, v tomto prípade jej hodnota ukazuje slabšie výsledky. Počet neurčených prvkov v OAL kóde (falošne negatívnych) oproti očakávanému OAL kódu je až 31, nakoľko všetkých spočítaných elementov v očakávanom kóde je 54 a v generovanom kóde iba 23 (a teda  $54 - 23 = 31$ ). Úplnosť má teda hodnotu

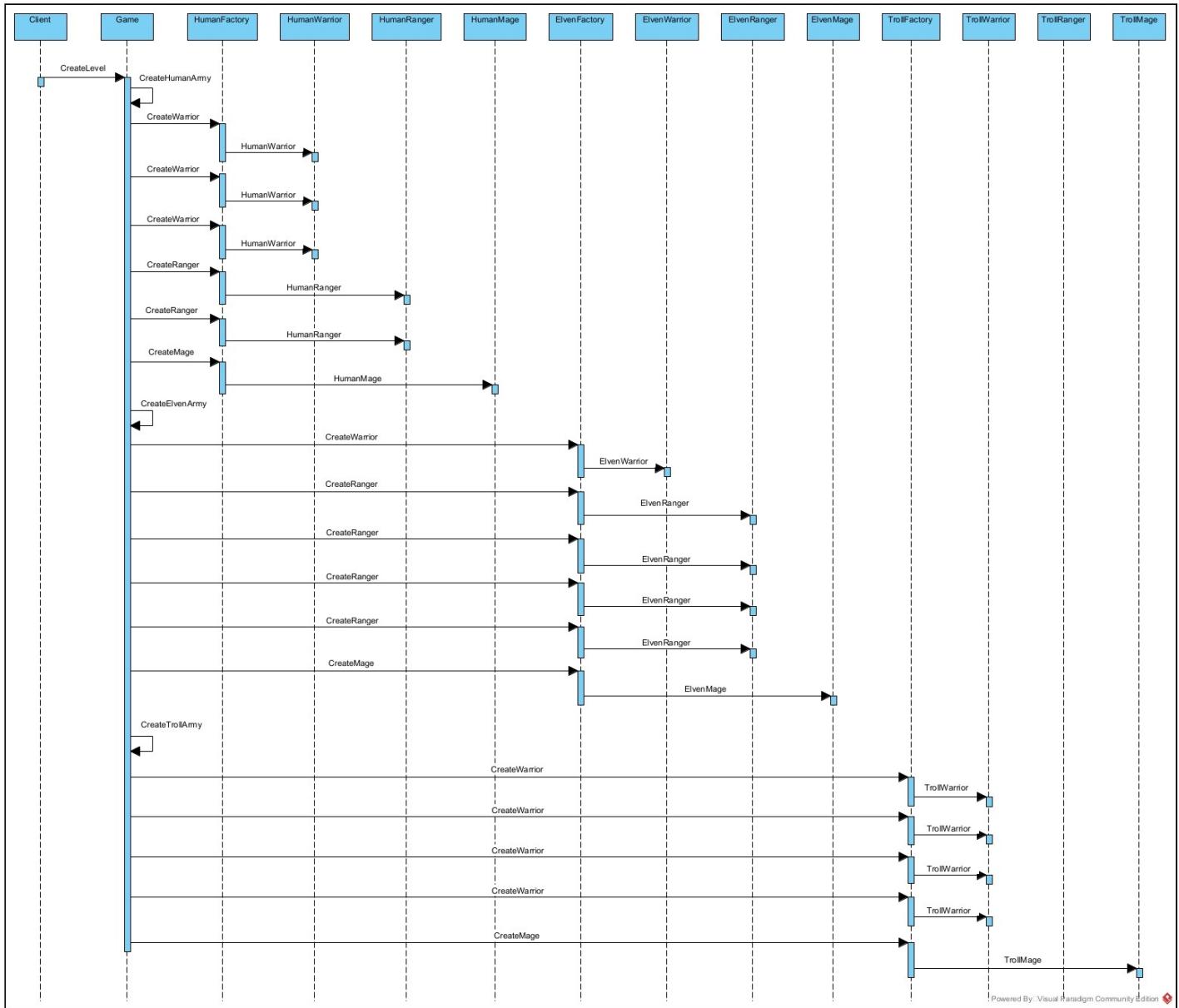
$$\frac{23}{23 + 31} = \frac{23}{54} \approx 0.43$$

Úplnosť so svojou relatívne nízkou hodnotou 0.43 ukazuje len to, čo je možné vidieť aj pri pohľade na vygenerovaný kód, ktorý je podstatne odlišný ako pôvodný kód. A teda môžeme konštatovať, že v prípade takéhoto sekvenčného diagramu, ktorý bol vytvorený na základe komplexného OAL kódu, je naša metóda kvôli nemožnosti zobrazenia mnohých elementov OAL kódu v sekvenčnom diagrame nedostatočná.

#### 4.2.5 Piaty prípad evaluácie (Abstract factory)

Rovnako, ako v predchádzajúcim prípade, aj v tomto prípade evaluácie máme vzorový kód, na základe ktorého sme vytvorili sekvenčný diagram a na základe ktorého sme

vygenerovali zdrojový OAL kód. Východiskový OAL kód znázorňuje vytváranie objektov počítačovej hry a je napísaný v súlade s návrhovým vzorom *Abstract factory*. Časť tohto kódu je zobrazená na obrázku 4.14. Kód z obrázku 4.14 je dynamicky zobrazený v sekvenčnom diagrame na obrázku 4.13. OAL kód vytvorený naším riešením sa sčasti nachádza na obrázku 4.15.



Obr. 4.13: Sekvenčný diagram piateho prípadu evaluácie.

V pôvodnom OAL kóde sú viaceré metódy volané viackrát. Takáto skutočnosť je zobrazená aj v sekvenčnom diagrame, kde sa na viacerých miestach nachádza niekoľko rovnakých správ za sebou. Vďaka tomu bolo možné viacnásobné volanie tých istých metód znázorniť aj v generovanom OAL kóde. Ako je možné vidieť aj z ukážok OAL kódov z obrázkov 4.14 a 4.15, generovaný OAL kód sa veľmi nelísi oproti pôvodnému kódu.

```

class Client
    method RunGame()
        create object instance game_1 of Game;
        game_1.CreateLevel();
    end method;
end class;

class Game
    method CreateLevel()
        self.CreateHumanArmy();
        self.CreateElvenArmy();
        self.CreateTrollArmy();
    end method;

    method CreateHumanArmy()
        create object instance humanfactory_1 of HumanFactory;
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateRanger();
        humanfactory_1.CreateRanger();
        humanfactory_1.CreateMage();
    end method;

    method CreateElvenArmy()
        create object instance elvenfactory_1 of ElvenFactory;
        elvenfactory_1.CreateWarrior();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateMage();
    end method;

    method CreateTrollArmy()
        create object instance trollfactory_1 of TrollFactory;
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateMage();
    end method;
end class;

class HumanFactory
    method CreateWarrior()
        create object instance humanwarrior_1 of HumanWarrior;
        humanwarrior_1.HumanWarrior();
    end method;

    method CreateRanger()
        create object instance humanranger_1 of HumanRanger;
        humanranger_1.HumanRanger();
    end method;

```

```

class Client
    method StartMethod()
        create object instance Game_inst of Game;
        Game_inst.CreateLevel();
    end method;
end class;

class Game
    method CreateLevel()
        self.CreateHumanArmy();
    end method;

    method CreateHumanArmy()
        create object instance HumanFactory_inst of HumanFactory;
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateRanger();
        HumanFactory_inst.CreateRanger();
        HumanFactory_inst.CreateMage();
        self.CreateElvenArmy();
    end method;

    method CreateElvenArmy()
        create object instance ElvenFactory_inst of ElvenFactory;
        ElvenFactory_inst.CreateWarrior();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateMage();
        self.CreateTrollArmy();
    end method;

    method CreateTrollArmy()
        create object instance TrollFactory_inst of TrollFactory;
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateMage();
    end method;
end class;

class HumanFactory
    method CreateWarrior()
        create object instance HumanWarrior_inst of HumanWarrior;
        HumanWarrior_inst.HumanWarrior();
    end method;

    method CreateRanger()
        create object instance HumanRanger_inst of HumanRanger;
        HumanRanger_inst.HumanRanger();
    end method;

```

Obr. 4.14: Ukážka očakávaného kódu 5. prípadu evaluácie.

Obr. 4.15: Ukážka generovaného kódu 5. prípadu evaluácie.

V metóde *CreateLevel* triedy Game v pôvodnom kóde sú všetky vlastné metódy tejto triedy volané v rámci metódy *CreateLevel*. V generovanom kóde sú na základe sekvenčného diagramu odrážajúceho pôvodný kód vlastné metódy volané postupne z jednotlivých vlastných metód (každá vlastná metóda je volaná prostredníctvom posledného volania predchádzajúcej vlastnej metódy). Funkčnosť generovaného kódu to však nemení a poradie volania daných metód je v generovanom kóde rovnaké ako v očakávanom kóde. Nepodstatný rozdiel v zdrojových kódoch spočíva aj v rozdielnych názvoch inštancií. Vďaka neveľkému rozdielu medzi týmito kódmi nie sú veľmi odlišné ani počty jednotlivých prvkov OAL kódu, čo je viditeľné v tabuľke 4.5.

Tabuľka 4.5: Porovnanie počtu elementov kódu v 5. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	14	14
Metóda	23	21
Vytvorenie inštancie	13	12
Volanie metódy	30	29
<b>Všetky elementy</b>	<b>80</b>	<b>76</b>

Na základe údajov z tabuľky 4.5 je zrejmé, že počet falošne negatívnych, teda nevytvorených elementov v generovanom kóde, je rovný číslu 4, keďže rozdiel v počtoch elementov je 4 ( $80 - 76 = 4$ ).

Presnosť našej metódy je znova rovná číslu 1, pretože počet správne určených prvkov je 76 a falošne pozitívnych 0, čiže:

$$\frac{76}{76 + 0} = 1$$

Úplnosť v tomto prípade evaluácie je rovná číslu

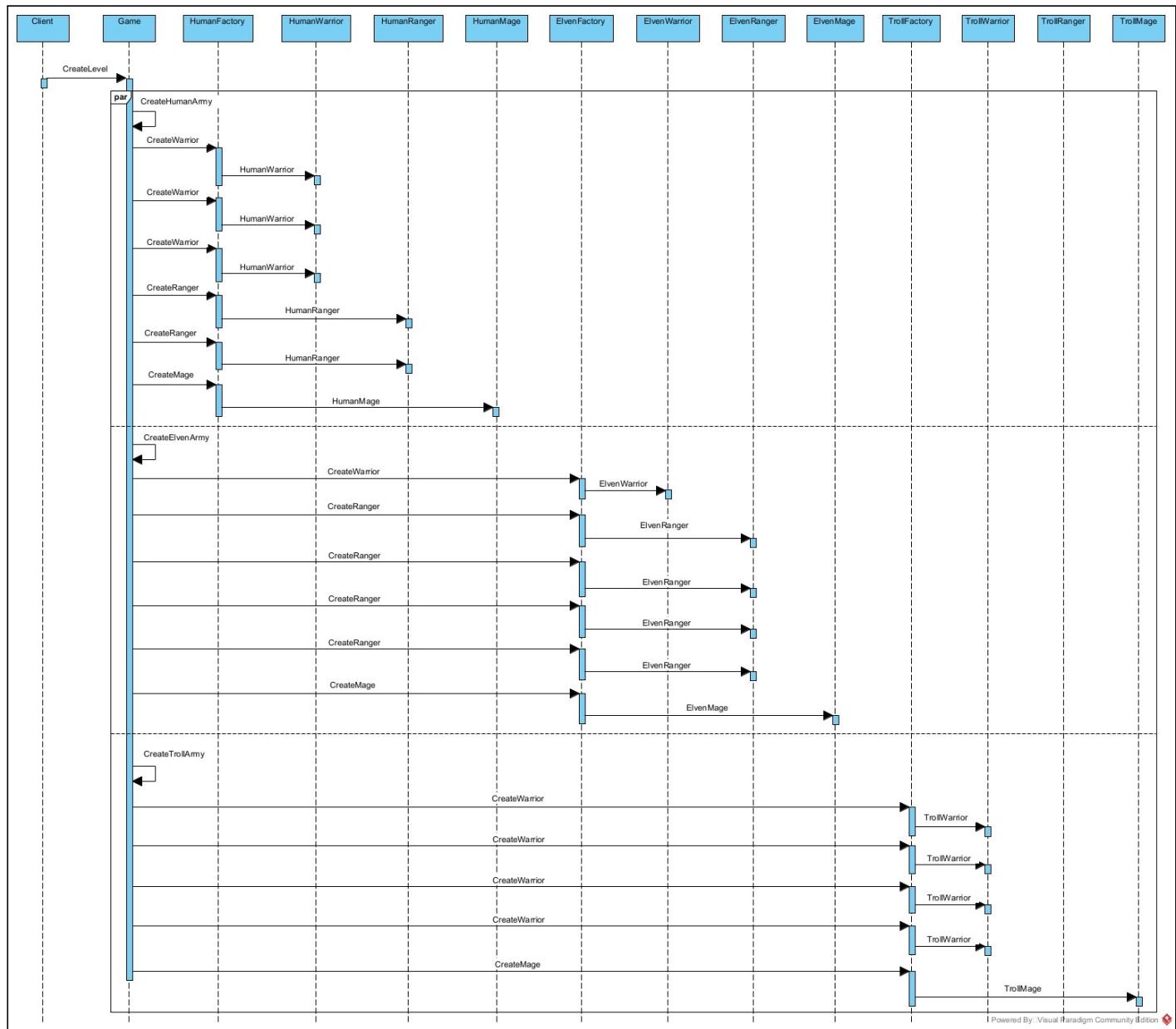
$$\frac{76}{76 + 4} = \frac{76}{80} = 0.95$$

Odlišnosť medzi počtami (a teda aj nižšia hodnota úplnosti) je spôsobená tým, že jedna metóda aj spolu s volaním inej metódy v pôvodnom kóde je súčasťou definovaná, avšak daná metóda v pôvodnom kóde nie je nikde volaná. A teda správy znázorňujúce volania daných metód nie sú vo východiskovom sekvenčnom diagrame zobrazené, čo znamená, že v generovanom kóde dané metódy ani nie sú definované. Takáto odlišnosť spolu s mierne nižšou hodnotou úplnosti však neznižuje relevanciu našej metódy. Generovaný kód, aj napriek chýbajúcej definícii daných metód, je v animácii programu *AnimArch* plne funkčný kód, rovnako ako pôvodný kód. A tak môžeme konštatovať, že naša metóda sa v tomto prípade evaluácie ukázala ako dostatočne relevantná.

#### 4.2.6 Šiesty prípad evaluácie (Abstract factory s paralelizáciou)

Šiesty a zároveň posledný prípad evaluácie je veľmi podobný predchádzajúcemu prípadu. Originálny kód je takmer totožný, až na to, že využíva paralelné vykonávanie príkazov. Ukážka originálneho kódu sa nachádza na obrázku 4.17. Časť generovaného kódu je zas zobrazená na 4.18. Sekvenčný diagram, na základe ktorého bol kód generovaný, je znázornený na obrázku 4.16.

Na základe ukážok OAL kódov sú dané OAL kódy takmer rovnaké, rozdiel je jedine v iných názvoch inštancií objektov. Oproti predchádzajúcemu prípadu evaluácie sú volania vlastných metód aj v generovanom kóde volané naraz v metóde *CreateLevel*,



Obr. 4.16: Sekvenčný diagram šiesteho prípadu evaluácie.

```

class Client
    method RunGame()
        create object instance game_1 of Game;
        game_1.CreateLevel();
    end method;
end class;

class Game
    method CreateLevel()
        par
            thread
                self.CreateHumanArmy();
            end thread;
            thread
                self.CreateElvenArmy();
            end thread;
            thread
                self.CreateTrollArmy();
            end thread;
        end par;
    end method;

    method CreateHumanArmy()
        create object instance humanfactory_1 of HumanFactory;
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateWarrior();
        humanfactory_1.CreateRanger();
        humanfactory_1.CreateRanger();
        humanfactory_1.CreateMage();
    end method;

    method CreateElvenArmy()
        create object instance elvenfactory_1 of ElvenFactory;
        elvenfactory_1.CreateWarrior();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateRanger();
        elvenfactory_1.CreateMage();
    end method;

    method CreateTrollArmy()
        create object instance trollfactory_1 of TrollFactory;
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateWarrior();
        trollfactory_1.CreateMage();
    end method;
end class;

class HumanFactory

```

```

class Client
    method StartMethod()
        create object instance Game_inst of Game;
        Game_inst.CreateLevel();
    end method;
end class;

class Game
    method CreateLevel()
        par
            thread
                self.CreateHumanArmy();
            end thread;
            thread
                self.CreateElvenArmy();
            end thread;
            thread
                self.CreateTrollArmy();
            end thread;
        end par;
    end method;

    method CreateHumanArmy()
        create object instance HumanFactory_inst of HumanFactory;
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateWarrior();
        HumanFactory_inst.CreateRanger();
        HumanFactory_inst.CreateRanger();
        HumanFactory_inst.CreateMage();
    end method;

    method CreateElvenArmy()
        create object instance ElvenFactory_inst of ElvenFactory;
        ElvenFactory_inst.CreateWarrior();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateRanger();
        ElvenFactory_inst.CreateMage();
    end method;

    method CreateTrollArmy()
        create object instance TrollFactory_inst of TrollFactory;
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateWarrior();
        TrollFactory_inst.CreateMage();
    end method;
end class;

class HumanFactory

```

Obr. 4.17: Ukážka očakávaného kódu 6. prípadu evaluácie.

Obr. 4.18: Ukážka generovaného kódu 6. prípadu evaluácie.

pričom sú volané v rámci paralelného bloku. Rozdiel v počtoch jednotlivých objektov sa nachádza v tabuľke 4.6.

Tabuľka 4.6: Porovnanie počtu elementov kódu v 6. prípade evaluácie.

Elementy OAL kódu	Počet elementov v očakávanom kóde	Počet elementov v generovanom kóde
Trieda	14	14
Metóda	23	21
Vytvorenie inštancie	13	12
Volanie metódy	30	29
Paralelný blok	1	1
Vlákno	3	3
Konštrukcia <i>end</i>	4	4
<b>Všetky elementy</b>	<b>88</b>	<b>84</b>

Ako vo všetkých predošlých prípadoch, aj v tomto prípade je presnosť rovná hodnote 1, keďže žiadnen element neboli našou implementáciou vygenerovaný navyše, teda:

$$\frac{84}{84 + 0} = 1$$

Rovnako ako v predošлом prípade, rozdiel v počtoch prvkov OAL kódu medzi porovnanými kódmi je 4. Dôvod je ten istý ako v piatom prípade evaluácie - v sekvenčnom diagrame, na základe ktorého sa generoval kód, nebola zobrazená správa znázorňujúca volanie, ktoré sa v pôvodnom kóde ani nevykonalo. A teda metrika úplnosť je rovná hodnote

$$\frac{84}{74 + 4} = \frac{84}{88} \approx 0.95,$$

čo sa taktiež rovná hodnote úplnosti z predošlého prípadu evaluácie.

Na základe uvedených faktov môžeme teda konštatovať, že aj pri sekvenčných diagramoch zobrazujúcich paralelné vykonávanie príkazov je naša metóda veľmi relevantná.

### 4.3 Relevantnosť našej metódy v prototype AnimArch

Jeden z cieľov našej práce bol obohatiť prototyp *AnimArch* o možnosť generovať zdrojový kód z dynamického modelu. Konkrétnie sme tento nástroj nepriamo rozšírili o možnosť spúštať OAL kód vygenerovaný na základe sekvenčného diagramu v tzv. animáciách interakcií objektov UML modelu, resp. diagramu tried.

Funkčnosť animácií v programe *AnimArch* závisí od korektnosti OAL kódu zapísaného vo vstupných animačných súboroch. Na základe zistení v predchádzajúcej kapitole vyplýva, že pokiaľ nie potrebné, aby kód v animáciách obsahoval štruktúry, ktoré momentálne nie je možné znázorniť v nami vytváraných sekvenčných diagramoch, naša metóda generovania zdrojového kódu na základe sekvenčných diagramov je veľmi relevantná. Výsledné generované kódy, predstavené v prvých dvoch a v posledných dvoch prípadoch evaluácie v predošnej podkapitole, je možné prostredníctvom animačných súborov v programe *AnimArch* spustiť ako riadne funkčné animácie.

V situáciach, ktoré vznikli v treťom a štvrtom prípade evaluácie nášho riešenia (kapitoly 4.2.3 a 4.2.4), chýbali v generovanom kóde viaceré elementy, ktoré by umožnili korektnosť tohto kódu a zároveň vykonanie animácie v *AnimArch-u*. Pri takýchto situáciach je nutné vygenerovaný OAL kód ručne upraviť tak, aby splňal požadované očakávania. V ďalších fázach vývoja nášho riešenia by sa uvedený problém dal odstrániť aj úpravou nástroja na vytváranie sekvenčných diagramov, používaného v rámci našej výskumnej skupiny (prototyp *SQD\_Tunder*). V tomto nástroji by napríklad potenciálne bolo možné pridať funkcionality, ktorá by umožňovala do sekvenčného diagramu (napr. v rámci správ) pridať elementy zdrojového kódu, ako vytváranie zoznamov, vytváranie premenných a pod.

## 4.4 Zhrnutie výsledkov evaluácie

Komparáciou rôznych typov zdrojových kódov sme v jednotlivých prípadoch evaluácie úspešne určili relevantnosť našej metódy generovania zdrojových kódov na základe dynamického modelu, konkrétnie sekvenčného diagramu.

Na základe pôvodných, resp. očakávaných OAL kódov, ktoré v našej metóde evaluácie tvorili úsudky o relevancii (relevance judgments), sme porovnali tieto kódy s kódmi vygenerovanými podľa sekvenčných diagramov, pričom dané sekvenčné diagramy zároveň dynamicky znázorňovali procesy pôvodných OAL kódov. Metrikami presnosť (precision) a úplnosť (recall) sme určili relevanciu našej metódy na základe počtov najdôležitejších elementov pôvodného kódu a vygenerovaného kódu.

Vo všetkých prípadoch evaluácie sa ukázalo, že naša metóda je na základe metriky presnosť veľmi relevantná, pretože v generovaných kódoch sa nenachádzali žiadne elementy navyše. V situáciách, kde v sekvenčnom diagrame nebolo možné zobraziť vyžadované zložitejšie štruktúry zdrojového kódu, bola hodnota úplnosti nižšia. Nízku hodnotu dosahovala úplnosť najmä v štvrtom prípade evaluácie (kapitola 4.2.4).

Kedže sme metrikami presnosť a úplnosť porovnávali iba počet elementov zdrojových kódov, hodnota týchto metrík, najmä metriky úplnosť, nie vždy korešpondovala s relevanciou generovaného kódu, resp. s jeho reálnou funkčnosťou. V stave, aký nastal napríklad v treťom prípade evaluácie (kapitola 4.2.3), počty prvkov OAL kódov neboli veľmi rozdielne, avšak vygenerovaný kód, obzvlášť kvôli priamemu použitiu cyklu *for each* a podmienok *if* príkazov, nebol funkčný. V posledných dvoch častiach evaluácie zas úplnosť ukazovala mierne nižšiu hodnotu, avšak funkčnosť generovaného kódu bola rovnaká ako funkčnosť pôvodného zdrojového kódu, keďže chýbajúce prvky, nenachádzajúce sa v generovanom kóde, boli pre jeho vykonateľnosť nepodstatné.

Na základe nášho úsudku, ako aj na základe hodnôt spomínaných metrík môžeme konštatovať, že naše výsledné riešenie generovania zdrojového kódu z dynamických modelov je dostatočne relevantné, najmä v situáciách, ktoré nepožadujú vo vygenerovanom zdrojovom kóde komplexné štruktúry. Nedostatky vyplývajúce z evaluácie našej implementácie zachádzajú nad rámec našej metódy, keďže na vytváranie momentálne absentujúcich elementov v generovaných zdrojových kódoch by bolo potrebné upraviť funkcionality vytvárania sekvenčných diagramov.

# Záver

Jeden z hlavných cieľov našej práce spočíval v obohatení existujúceho prototypu modelovania softvérovej architektúry o možnosť generovať zdrojový kód z dynamického modelu. Presnejšie povedané, náš cieľ spočíval v umožnení transformácie dynamických modelov, vyjadrených sekvenčným diagramom jazyka UML, do zdrojového kódu tak, aby sme zároveň umožnili na základe takéhoto vygenerovaného kódu vykonať animáciu dynamiky UML modelu v existujúcom prototype s názvom AnimArch. Na základe rozsiahleho návrhu, podrobnej analýzy a evaluácie nášho výsledného riešenia môžeme konštatovať, že cieľ našej práce bol splnený.

V prvej kapitole tejto publikácie sme priblížili metodiku súvisiacu s našou pracou z prostredia softvérového inžinierstva, ktorá nesie názov modelom riadený vývoj (Model-driven development). Ďalej sme v nej predstavili využité technológie a postupy. Na základe cieľov zadania práce sme mali možnosť generovať zdrojový kód v jazyku OAL alebo Python. Pre vysokú úroveň abstrakcie jazyka OAL a pre jeho používanie v prototype AnimArch sme si vyбрали práve tento jazyk. V rámci tejto kapitoly sme opísali aj generátor parserov ANTLR a s ním súvisiace teoretické aspekty. Nástroj ANTLR sme použili ako prvý krok pri generovaní zdrojového kódu, konkrétnie na parsovanie vstupných súborov definujúcich dynamické modely, teda sekvenčné diagramy. Pre komplexnosť vstupných súborov sa však tento nástroj v ďalšom postupe práce ukázal ako nevhodný a proces generovania OAL kódu sme po parsovaní museli uskutočniť vlastnou implementáciou. V kontexte prvej kapitoly sme analýzou zvolených a nami využitých metód modelovania softvéru zároveň splnili ďalšie z cieľov práce.

Druhá kapitola tejto práce obsahuje rozsiahly návrh riešenia generovania zdrojového kódu zo sekvenčných diagramov, teda návrh nášho základného cieľa. Popri návrhu postupu splnenia požadovaných výsledkov sme v tejto kapitole opísali aj prototypy využívané v rámci našej výskumnnej skupiny. Týmito prototypmi sú nástroj SQD\_Tunder a program AnimArch. Prvý spomínaný prototyp slúžil na vytváranie a ukladanie sekvenčných diagramov, resp. súborov definujúcich sekvenčné diagramy a druhý program sme nepriamo dopĺňali o možnosť spustenia animácií procesov zobrazených v dynamickom modeli, konkrétnie v sekvenčnom diagrame.

Analýzu nášho postupu transformácie sekvenčných diagramov do OAL kódu sme uviedli v tretej kapitole. V tejto kapitole sme predstavili aj definované pravidlá, na

základe ktorých sme identifikovali sekvenčné diagramy, ktoré sme v procese generovania zdrojového kódu považovali za nevalídne. Poznamenávame, že definíciou a aplikovaním spomínaných pravidiel táto práca zachádza nad rámec svojho pôvodného cieľa.

V poslednej kapitole sme priniesli evaluáciu vytvorenej implementácie generovania zdrojového kódu zo sekvenčných diagramov. Výsledky evaluácie ukázali, že pokial' nie je nutné, aby výsledný OAL kód disponoval štruktúrami nezobraziteľnými v nami vytváraných sekvenčných diagramoch, je naše riešenie veľmi relevantné a generovaný OAL kód je možné použiť v animáciách programu AnimArch.

Existuje niekoľko potenciálnych možností rozšírenia našich výsledkov. Vo využívanom nástroji na vytváranie sekvenčných diagramov, SQD\_Tunder, by napríklad bolo užitočné doplniť funkcionality umožňujúcu pridať konkrétné príkazy, ako aj logiku kódu v určitom jazyku. Taktiež by bolo možné do programu AnimArch priamo zakomponovať vytváranie sekvenčných diagramov a následné generovanie zdrojového kódu, spolu so spúšťaním animácií UML modelov na základe týchto diagramov. Podotýkame, že aby sme uľahčili aplikovanie nášho riešenia v ďalšom vývoji a výskume, všetku funkcionality sme implementovali v programovacom jazyku C#, ktorým sú vybudované aj spomínané prototypy SQD\_Tunder a AnimArch.

Na záver môžeme konštatovať, že implementáciou transformácie sekvenčných diagramov do vysoko abstraktného kódu v jazyku OAL sme vytvorili v oblasti modelovania softvéru unikátne dielo a zároveň sme splnili všetky ciele tejto práce. Veríme, že naše výsledky nájdú uplatnenie v ďalšom výskume a vo sfére softvérového inžinierstva aj praktické uplatnenie.

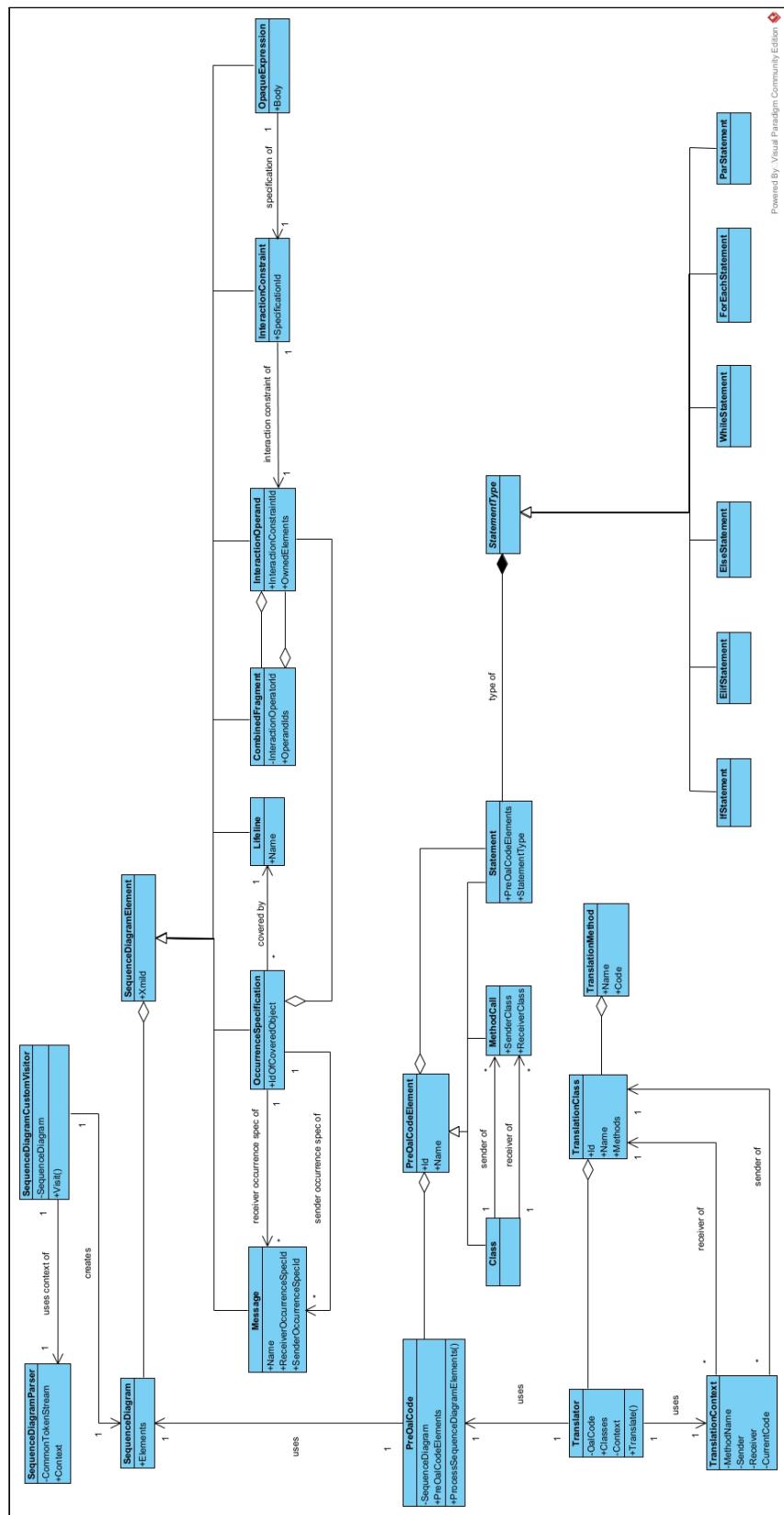
# Literatúra

- [1] SELIC, Bran. The pragmatics of model-driven development. *IEEE software*. 2003, 20(5), s. 19-25.
- [2] MELLOR, Stephen J.; CLARK, Anthony N. a FUTAGAMI, Takao. Model-driven development. *IEEE software*. 2003, 20(5), s. 14.
- [3] MELLOR, Stephen J. a BALCER, Marc J. Executable UML: a foundation for model-driven architecture. *Addison-Wesley Professional*. 2002.
- [4] XTUML. *Action Language (OAL) Tutorial*. [online]. XtUML. cop2023 [cit. 2023-11-03]. Dostupné na: <https://xtuml.org/learn/action-language-tutorial/>
- [5] UKIC, Nenad a TESLA, Ericsson Nikola. *XtUML model complexity metrics and the influence of their distribution on model understandability*. Dizertačná práca. Split: Fakultet elektrotehnike, strojarstva i brodogradnje u Splitu, 2016.
- [6] GEEKSFORGEEKS. *Lexical Analysis and Syntax Analysis*. [online]. GeeksforGeeks. 2023 [cit. 2023-11-01]. Dostupné na: <https://www.geeksforgeeks.org/lexical-analysis-and-syntax-analysis/>
- [7] NORTHWOOD, Chris. *Lexical and syntax analysis of programming languages*. [online]. 2020 [cit. 2023-11-04]. Dostupné na: <https://www.pling.org.uk/cs/lسا.html>
- [8] TOMASSETTI, Gabriele. *The ANTLR Mega Tutorial*. [online]. STRUMENTA. 2023 [cit. 2023-11-05]. Dostupné na: <https://tomassetti.me/antlr-mega-tutorial/>
- [9] JIANG, Tao et al. Formal grammars and languages. In: *Algorithms and Theory of Computation Handbook*. Volume 1. Chapman and Hall/CRC, 2009, s. 549-574.
- [10] PARR, Terence. *The Definitive ANTLR 4 Reference*. 1. The Pragmatic Bookshelf, 2013. ISBN 9781934356999.

- [11] THONGMAK, Mathupayas a MUENCH AISRI, Pornsiri. Design of rules for transforming sequence diagrams into java code. *Ninth Asia-Pacific Software Engineering Conference*. 2002, s. 485-494. IEEE.
- [12] ELKASHEF, Nermeen a HASSAN, Yasser F. Mapping UML Sequence Diagram into the Web Ontology Language. *International Journal of Advanced Computer Science and Applications*. 2020, 11(5).
- [13] KULKARNI, D. R. a SRINIVASA, C. K. Novel approach to transform UML Sequence diagram to Activity diagram. *Journal of University of Shanghai for Science and Technology*. 2021, 23(7).
- [14] PANTHI, Vikas a MOHAPATRA, Durga P. Automatic test case generation using sequence diagram. In: *Proceedings of International Conference on Advances in Computing*. Springer India, 2012, s. 277-284.
- [15] FRANKO, Marián Ján. *Podpora vývoja softvéru pomocou editovania modelu v rozšírenej realite*. Diplomová práca. Bratislava: Fakulta informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave, 2021.
- [16] RADOSKÝ, Lukáš. *Optimization and Reuse in Development of Large Software Systems*. Minimová práca PhD. štúdia. Bratislava: Fakulta matematiky, fyziky a informatiky Univerzity Komenského v Bratislave, 2023.
- [17] XTUML. *Object action language reference manual*. [online]. 2009 [cit. 2024-11-20]. Dostupné na: <http://www.oaool.com/docs/OAL08.pdf>
- [18] NOVÁK, Filip. *Vizualizácia softvérových architektúr a generovanie zdrojového kódu*. Diplomová práca. Bratislava: Fakulta matematiky, fyziky a informatiky Univerzity Komenského v Bratislave, 2022.
- [19] FOWLER, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2018. ISBN 013486512X.
- [20] VOORHEES, Ellen M. Variations in relevance judgments and the measurement of retrieval effectiveness. In: *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 1998, s. 315-323.
- [21] GOUTTE, C. a GAUSSIER, E. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In: *European conference on information retrieval*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, s. 345-359.

## **Príloha A: diagram tried nášho riešenia**

Na obrázku A.1 sa nachádza kompletný diagram tried nášho riešenia.



Obr. A.1: Kompletnejší diagram tried nášho riešenia.