Slovenská technická univerzita
Fakulta informatiky a informačných technológií
FIIT-5212-85962

Simona Miková

Vizualizácia a editovanie diagramov viacrozmerného UML

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	ÚISI, FIIT STU Bratislava
Vedúci práce:	doc. Ing. Ivan Polášek PhD.

Bratislava, 2018

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

ZADANIE BAKALÁRSKEHO PROJEKTU

Meno študenta:	**Miková Simona**
Študijný odbor:	Informatika
Študijný program:	Informatika
Názov projektu:	**Vizualizácia a editovanie diagramov viacrozmerného UML**

Zadanie:
Analýza a návrh softvérových systémov sú natoľko komplexné a zložité procesy, že si vyžadujú podporu špecifických metód, jazykov a nástrojov. Jedným z nich je aj jazyk UML, ktorý poskytuje zachytenie modelu softvérových systémov grafickou notáciou formou série doplňujúcich sa a nadväzujúcich diagramov. Pomocou pri orientácii v takýchto diagramoch by mohla byť transformácia zobrazenia do trojrozmerného priestoru počítačovej grafiky.
Pokračujte v tvorbe funkcionality (pridávanie, odoberanie a zmena vrstiev a elementov) nových prototypov 3D UML diagramu tried, diagramu aktivít alebo sekvenčného diagramu, ktoré opisujú scenáre prípadov použitia a modulovú štruktúru celého systému. Navrhnite možnosti zobrazenia diagramov aj s pomocou nových kolmých vrstiev alebo inými vhodnými prístupmi, napríklad vrstvenie alternatívnych alebo paralelných vetví interakcií. Overte Vašu metódu a prototyp zobrazovania v trojrozmernom priestore UML.

Práca musí obsahovať:

Anotáciu v slovenskom a anglickom jazyku
Analýzu problému
Opis riešenia
Zhodnotenie
Technickú dokumentáciu
Zoznam použitej literatúry
Elektronické médium obsahujúce vytvorený produkt spolu s dokumentáciou

Miesto vypracovania: Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU, Bratislava
Vedúci projektu: doc. Ing. Ivan Polášek, PhD.

Termín odovzdania práce v zimnom semestri : 11. 12. 2018
Termín odovzdania práce v letnom semestri : 7. 5. 2019

SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4
1

Bratislava 17. 9. 2018

prof. Ing. Pavol Návrat, PhD.
riaditeľ ÚISI

Čestné vyhlásenie

Čestne vyhlasujem, že som túto prácu vypracovala samostatne, na základe konzultácií a s použitím uvedenej literatúry.

V Bratislave, 7. 5. 2019 .....

Simona Miková

Poďakovanie

Touto cestou by som sa chcela poďakovať vedúcemu bakalárskej práce doc. Ing. Ivanovi Polášekovi PhD., za konzultácie, trpezlivosť a odborné rady pri jej vypracovávaní.

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Informatika 3
Autor: Simona Miková
Bakalárska práca: Vizualizácia a editovanie diagramov viacrozmerného UML
Vedúci práce: doc. Ing. Polášek Ivan PhD.
December 2018

Modelovací jazyk UML je v súčasnosti najpoužívanejším jazykom pre tvorbu modelov popisujúcich systém a jeho fungovanie. S rastúcou zložitosťou programov sa ale tieto modely stávajú neprehľadné a ťažké na pochopenie. Práca opisuje dôležitosť modelovania softvéru ako takého a výhody, ktoré by prinieslo pridanie ďalšieho rozmeru do diagramov. Prináša pohľad aj na už existujúce riešenia vizualizácie diagramov v trojrozmernom priestore. Ďalej opisuje diagram sekvencií, jeho syntax a vysvetľuje metamodel tohto diagramu. Okrem toho práca analyzuje existujúci prototyp vizualizácie sekvenčného diagramu v trojrozmernom priestore, ktorá prebieha v reálnom čase počas ladenia programu. Tento prístup prináša množstvo výhod a uľahčí navigáciu v zdrojovom kóde. Na tomto prototype navrhuje a implementuje ďalšie rozšírenia vizualizácie ako aj editovania vo forme detekcie cyklov v programe a ich následnom obalení fragmentom. Na detekovanie cyklu využíva kombináciu dynamickej a statickej analýzy, vďaka tomu ponúka zaujímavé a jednoduché riešenie.

Naša práca prináša zaujímavé a inovatívne riešenie využitia tretieho rozmeru pri modelovaní softvéru spojené s možnosťou vizualizácie zdrojového kódu v reálnom čase.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Informatics
Author: Simona Miková
Bachelor thesis: Visualization and editing diagrams of multidimensional UML
Supervisor: doc. Ing. Polášek Ivan PhD.
December 2018

Unified modeling language is currently the most widely used language for creating the models describing the system and its functioning, but with the increasing complexity of programs, these models are becoming disarranged and hard to understand. The work describes importance of software modelling per se and the benefits which adding a new dimension to the diagrams would bring. It gives an insight into the existing solutions for visualising diagrams in three-dimensional space. It describes the sequence diagram and its syntax and explains the metamodel of this diagram. In addition, work analyzes an existing prototype of the sequence diagram visualization in three-dimensional space that runs in real time during the program debugging. This approach brings many benefits and it makes navigation in the source code easier. Other visualisation extensions as well as editing in the form of cycle detection in the program are designed and implemented on this prototype. It uses a combination of dynamic and static analysis to detect the cycle, offering an interesting and simple solution.

Our work brings an interesting and innovative solution for the use of the third dimension in the modeling software connected with the possibility of the visualisation of the source code in real time.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

V súčasnosti sú softvérové systémy čoraz viac komplexné a to zvyšuje potrebu ich modelovania a vizualizácie. Najpoužívanejší jazyk v tejto oblasti je v dnešnej dobe Unified Modeling Language, skrátene UML. Jeho cieľom je poskytnúť štandardnú notáciu pre tvorbu diagramov, ktoré je možné využívať v každej fáze vývoja systému.

Takto rozsiahle systémy môžu obsahovať aj milióny riadkov zdrojového kódu. Diagramy tried alebo sekvencií takého systému bývajú väčšie ako celá obrazovka počítača a preto navigácia v nich môže byť zložitá a rýchle pochopenie súvislostí takmer nemožné. Problém nastáva práve vtedy, ak sa v takomto systéme vyskytne chyba, ktorú je potrebné rýchlo opraviť, alebo pri oboznamovaní niekoho nového so systémom. V týchto prípadoch by prácu programátora uľahčilo práve trojrozmerné zobrazenie týchto diagramov v reálnom čase vykonávania programu, napríklad počas jeho ladenia. Takéto zobrazenie by mu pomohlo jednoducho pochopiť v akom poradí sa jednotlivé časti kódu vykonávajú a ako sa správajú.

Prechod od 2D zobrazenia k 3D by nám teda mohol priniesť množstvo výhod. Pridanie ďalšej dimenzie by znamenalo napríklad nové možnosti rozmiestnenia jednotlivých prvkov diagramu v priestore, čím by sa zvýšila jeho čitateľnosť a prehľadnosť. Pre človeka je taktiež oveľa prirodzenejšie pochopenie súvislostí v trojdimenzionálnom priestore a preto by zavedenie ďalšej dimenzie v UML diagramoch bolo logickým krokom k vylepšeniu a zjednodušeniu vizualizácie systémov.

V tejto práci sa budeme zaoberať práve využitím viacrozmerného priestoru pri vizualizácii UML diagramov, konkrétne budeme pracovať s diagramom sekvencií. Výsledkom práce bude zobrazenie sekvečného diagramu z dynamickej analýzy programu v trojrozmernom priestore.

V druhej kapitole sa budeme venovať bližšiemu opisu dôležitosti modelovania softvéru, modelovaciemu jazyku UML, zobrazovaniu diagramov v 2D priestore a možnými výhodami, ktoré by mohlo priniesť 3D zobrazenie. Už existujúce riešenia v danej oblasti sú taktiež krátko opísané v tejto kapitole. Kapitola 3 pokrýva návrh riešenia, bliž-

šie sa zameriava na dôvody vytvárania prototypu a uvádza požiadavky na vytváraný prototyp. V kapitole 4 je opísaná implementácia nami navrhnutého prototypu a jeho základné vlastnosti. V kapitole 5 sa budeme venovať testovaniu funkčnosti nami vytvoreného prototypu. Kapitola 6 prináša zhrnutie našej práce, opis splnenia či nesplnenia jednotlivých požiadaviek a návrhy na ďalší možný vývoj. Na konci práce sa nachádzajú prílohy, ktoré obsahujú: technickú dokumentáciu (Príloha A), používateľskú príručku (Príloha B), denník práce za oba semestre (Príloha C) a obsah elektronického média (Príloha D).

Kapitola 2

Analýza

V tejto kapitole sa budeme venovať analýze súčasných riešení vizualizácie zdrojového kódu pomocou diagramov UML a ich transformáciou do 3D. V prvej časti sa bližšie zameriame na definície diagramov, konkrétne sekvenčného diagramu, o ktorého 3D vizualizáciu sa budeme vrámci bakalárskej práce usilovať. Ďalšou časťou tejto kapitoly je analýza už existujúcich riešení v porovnaní s naším návrhom.

2.1 Modelovanie softvéru

Modelovanie softvéru nám prináša abstrakciu, ktorá môže pomôcť pri zredukovaní zložitosti tvorby softvérov. Namiesto práce so softvérom ako celkom sa tak zaoberáme len jeho určitými časťami [8]. Taktiež nám modelovanie pomáha lepšie pochopiť vytváraný softvér a vyjadriť naše myšlienky. Podľa Object Modeling Group je modelovanie navrhovanie softvérových aplikácií pred samotnou implementáciou [21]. Lepšie porozumenie softvéru môže byť dosiahnuté práve pohľadom z rôznych perspektív, ako napríklad modelmi požiadaviek a statickými či dynamickými modelmi daného softvéru [5]. Tieto modely najčastejšie vizualizujeme prostredníctvom dvojdimenzionálnych (2D) diagramov. Jedným z najznámejších jazykov používaných pri modelovaní softvéru je Unified Modeling language, UML.

2.1.1 Unified Modeling Language

Unified Modeling Language je štandardizovaný modelovací jazyk, určený na tvorbu softvérových modelov opisujúcich systém a jeho architektúru či samotnú funkčnosť [21]. Pozostáva zo súboru diagramov vyvinutých s cieľom pomôcť špecifikovať a vizualizovať softvér či vytvárať a dokumentovať artefakty týchto systémov. Predstavuje veľmi dôležitú časť vývoja objektovo orientovaného softvéru. Používa väčšinou grafické notácie na vyjadrenie návrhu systémov. Cieľom UML je poskytnúť štandardnú notáciu, ktorú môže využívať široká škála systémov v každej fáze vývoja softvéru [21].

2.2 Sekvenčný diagram

V našej práci budeme bližšie pracovať práve so sekvenčným diagramom. Diagram sekvencií je najbežnejším typom interakčného diagramu, ktorý zobrazuje interakcie objektov v čase [27]. V terminológii UML interakcia predstavuje jednotku správania, ktorá sa zameriava na pozorovateľnú výmenu informácií medzi jednotlivými elementami diagramu (napríklad medzi objektami) v podobe správ [27]. Sekvenčný diagram túto interakciu opisuje tým, že sa zameriava na postupnosť správ, ktoré sú vymieňané medzi objektami, spolu s ich korešpondujúcim výskytom na čiarach života [13].

2.2.1 Syntax diagramu sekvencií

Syntax definovaná v špecifikácii UML pozostáva z konkrétnej syntaxe, ktorá definuje používanú grafickú notáciu, a abstraktnej syntaxe. Abstraktná syntax je daná metamodelom definujúcim vzťahy medzi jednotlivými prvkami [19].

2.2.2 Konkrétna syntax

Grafickú notáciu sekvenčného diagramu si ukážeme na konkrétnom príklade uvedenom na obrázku 2.1. Bližšie popíšeme len časti, ktorými sa budeme zaoberať v našej práci.

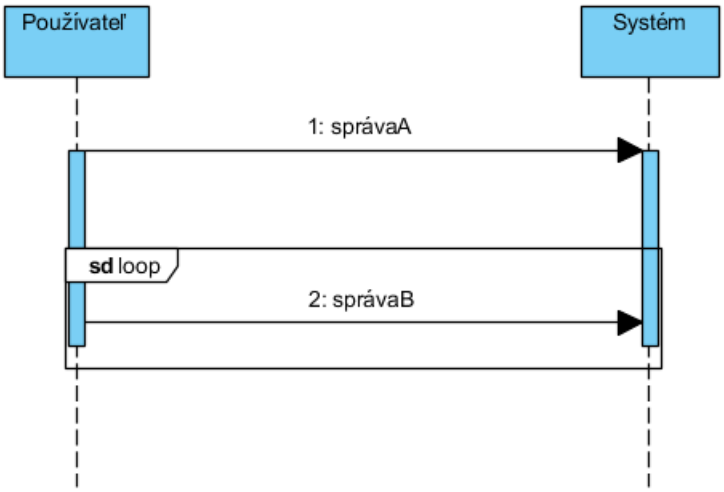
Čiary života reprezentujú životný cyklus jednotlivých účastníkov interakcie. Účastníci predstavujú objekty, medzi ktorými dochádza k interakcii. V našom príklade sa jedná konkrétne o používateľa a systém. Interakcia je znázornená pomocou správ. Úsek, počas ktorého je objekt aktívny sa označuje zvislým obdĺžnikom nazývaným aktivácia. Dôležitou časťou sekvenčného diagramu sú fragmenty. Fragment zoskupuje určité prvky diagramu do logického celku. Kombinovaný fragment je obdĺžnikový prvok, ktorý v ľavom hornom rohu obsahuje operand interakcie. Operand určuje typ fragmentu a upravuje vykonávanie správ, ktoré sa v ňom nachádzajú.

Najpoužívanejšie typy operandov sú napríklad:

- Alt - ponúka niekoľko možností alternatívy, vykoná sa práve tá, ktorej podmienka je v danom momente splnená.

- Opt - podobne ako pri operátore alt sa vykoná možnosť, ktorej podmienka je splnená. Neponúka však žiadne alternatívy, teda buď sa fragment vykoná celý, alebo sa nevykoná vôbec.

- Loop - definuje slučku, fragment sa vykonáva, pokiaľ je uvedená podmienka splnená.

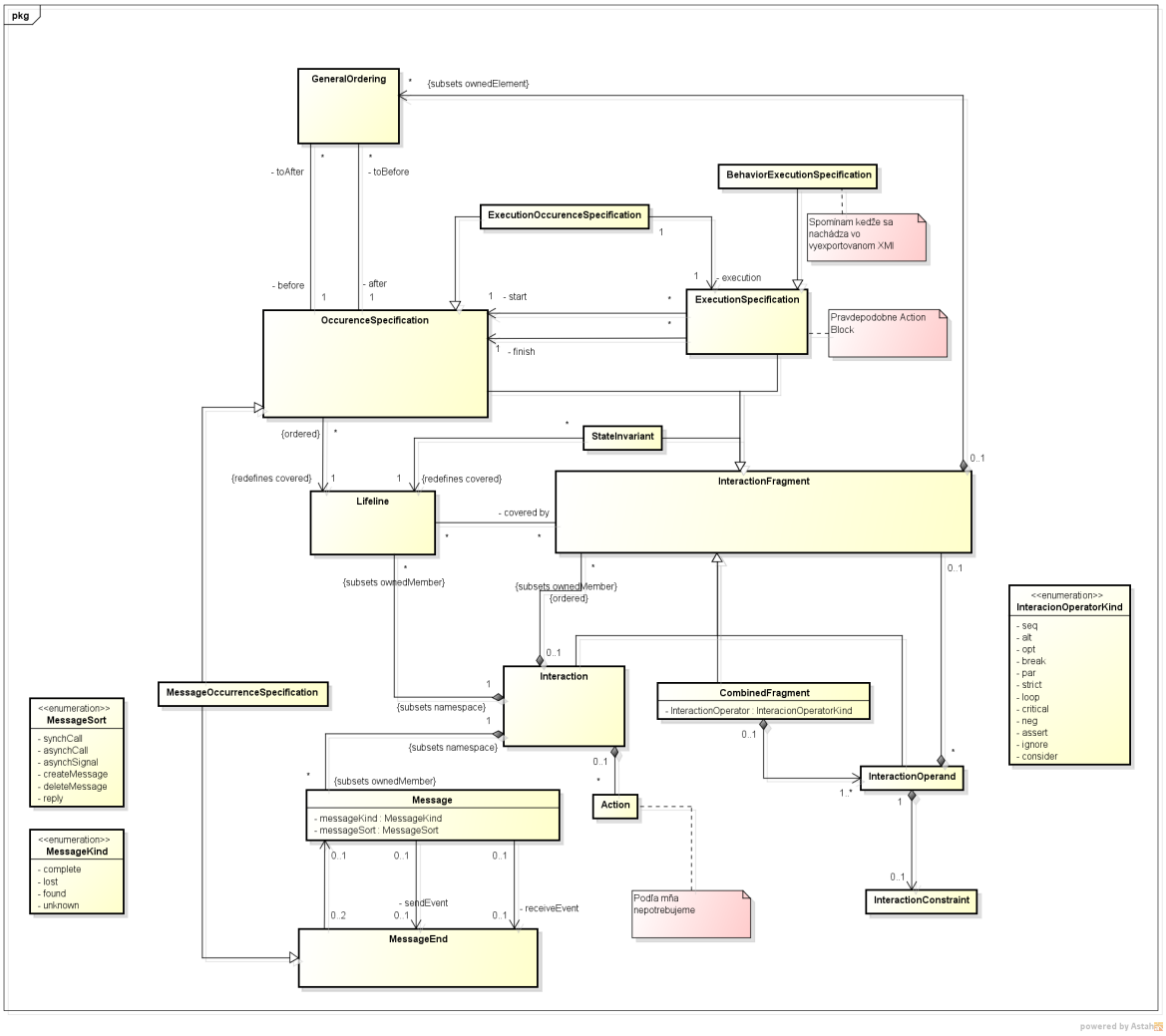


Obr. 2.1: Metamodel sekvenčného diagramu [20]

2.2.3 Abstraktná syntax a metamodel

Abstraktná syntax je definovaná metamodelovaním. Pod pojmom metamodel môžeme teda chápať akúsi abstrakciu, ktorá definuje štruktúru modelu UML. Potom každý UML model je konkrétnou inštanciou metamodelu tohto typu. Metamodel na obrázku 2.2 je prevzatý z UML Superstructure Specification dostupnej na [20].

V danom metamodeli môžeme vidieť triedy, ktoré by predstavovali jednotlivé prvky sekvenčného diagramu alebo ich časti, spolu s ich kardinalitami. Každá interakcia sa skladá z čiar života, správ a môže taktiež obsahovať fragment, alebo akciu. Trieda InteractionFragment predstavuje abstraktnú triedu pre triedy CombinedFragment a ExecutionSpecification, ktorá znamená vykonanie operácie alebo správania. V ďalšej časti sa pozrieme na konkrétne príklady metamodelov sekvenčného diagramu, kde bližšie opíšeme aj ďalšie jeho časti.



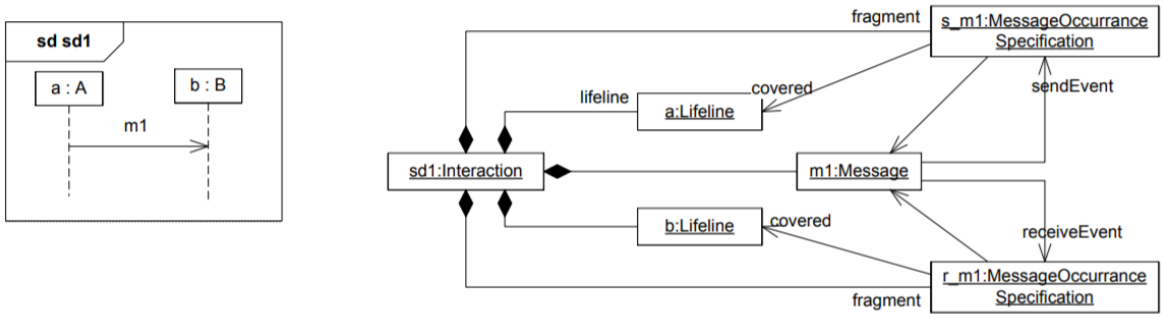
Obr. 2.2: Metamodel sekvenčného diagramu [20]

2.2.4 Konkrétne príklady sekvenčného diagramu a jeho metamodelu

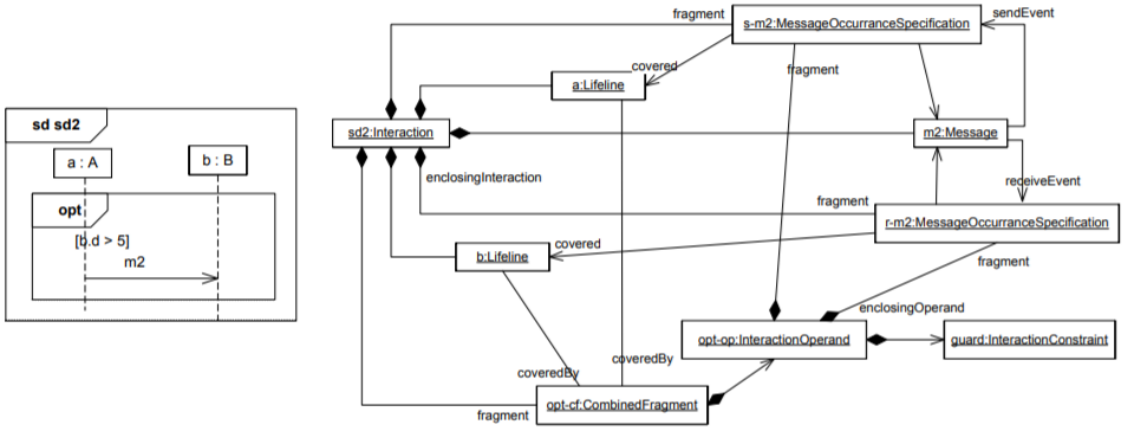
Na pravej strane obrázku 2.3 môžeme vidieť metamodel sekvenčného diagramu sd1. Trieda Interaction predstavuje konkrétnu interakciu, ktorú daný sekvenčný diagram zachytáva. Interakcia obsahuje typy Lifeline, Message a MessageOccurenceSpecification. Presnejšie bude interakcia obsahovať čiaru života a aj b a správu m1, ktorá sa posiela. Ako vidíme, správa musí mať definovaný bod začiatku a konca na konkrétnej čiare života, na čo nám slúžia konkrétne inštancie triedy MessageOccuranceSpecification.

Ak by sa v sekvenčnom diagrame nachádzal aj fragment, ktorý by prekrýval obe čiary života a aj b a obsahoval by správu m1, do metamodelu by nám pribudli ďalšie tri triedy, ktorý by daný fragment definovali, ako môžeme vidieť na obrázku 2.4. Trieda CombinedFragment obsahuje typ InteractionOperand a slúži na to, aby sme vedeli definovať typ operátora. Trieda interactionOperand zase obsahuje typ Interacti-

onConstraint a táto zase určuje podmienku, v našom prípade [b.d > 5].



Obr. 2.3: Príklad konkrétnej a abstraktnej syntaxe v jednoduchom diagrame z [19]



Obr. 2.4: Príklad konkrétnej a abstraktnej syntaxe v komplexnejšom diagrame z [19]

2.3 Potreba vizualizácie softvéru

Objektovo orientované programovanie sa môže začínajúcim aj skúseným programátorom javiť ako náročné. Zavádza netriviálne pojmy ako dedenie, zapuzdrenie, abstrakciu alebo polymorfizmus. Programátor má často problém pri pohľade na komplexný kód pochopiť jeho vykonávanie alebo vzťahy medzi objektami. Preto sa zvýšil záujem o využívanie rôznych techník vizualizácie softvéru. Vizualizácia môže práve vďaka zobrazovaniu informácií v inej podobe pomôcť pochopiť súvislosti, ktoré sú pri obyčajnom prezeraní kódu ťažko viditeľné [24]. Programátori vyžadujú podporu vizualizácie pomocou modelov, ktoré im pomôžu efektívnejšie implementovať a navrhovať jednotlivé komponenty systému [7]. Taktiež požadujú možnosť dynamickej vizualizácie, ktorá by im pomohla lepšie pochopiť vykonávané programy. Ideálne by tieto statické a dynamické prístupy vizualizácie mali byť konzistentné [7].

Pri vizualizácii softvéru s cieľom jeho lepšieho pochopenia sa ako veľmi sľubný ukázal práve jazyk UML. Ako uvádzajú autori v [24], hlavnou motiváciou výberu je jeho popularita a jednoduchosť použitia. Okrem toho je UML možné používať v každej etape vývoja softvéru.

2.4 Prechod od 2D zobrazenia k 3D

Pri modelovaní softvéru najčastejšie využívame 2D diagramy. Dôvod prečo sa používajú práve 2 dimenzie môže byť ten, že myšlienka načrtnutia diagramu na papier alebo jeho vytvorenie v konkrétnom nástroji, sa javí byť jednoduchšia ako tvorba 3D modelu [14]. Prevládajúce používanie 2D diagramov môžeme priradiť aj skutočnosti, že väčšina týchto diagramov je založená práve na grafoch, ako opisujú v [4].

Na vizualizáciu modelov v 2D bolo vynaloženého mnoho úsilia. Avšak stále viac komplexné požiadavky a zložitosť navrhovaných softvérov si vyžaduje zlepšenie vizualizácie, a tým lepšie pochopenie vytváraných modelov pre lepšiu spoluprácu dizajnérov a ich tímov pri návrhu a tvorbe softvéru [2]. Dve dimenzie pri komplexnejších systémoch nemusia stačiť na dostatočne prehľadné znázornenie všetkých vzťahov, samotné diagramy môžu vyzerať zložitejšie ako samotný zdrojový kód [12]. Preto by práve zavedenie ďalšej dimenzie prinieslo množstvo výhod [11].

2.4.1 Výhody 3D zobrazenia

UML ako najrozšírenejší modelovací jazyk súčasnosti prezentuje jednotlivé modely iba v 2D podobe. Grafické metódy pre modelovanie softvérových návrhov zdôrazňujú štruktúru a vzťahy medzi jednotlivými prvkami, avšak využívanie len dvoch dimenzií môže mať za následok ťažkosti pri vizualizácii modelov, ako uvádzajú v [1]:

- Keďže zložitosť softvérov stále rastie, diagramy reprezentujúce tieto softvéry sa rozširujú tiež, niekedy až exponenciálne, a preto pochopenie štruktúry ako celku z daných diagramov môže byť ťažšie.
- Keďže diagramy naberajú na veľkosti a zložitosti, usporiadanie sa stáva čoraz ťažšie a časovo náročnejšie a prináša mnohé problémy.
- V hierarchických štruktúrach sa komponenty z tej istej vrstvy musia zobraziť v hierarchii na rovnakej úrovni. V 2D to znamená lineárne usporiadanie pre každú úroveň, ktoré ale rýchlo spotrebuje priestor. Tieto úrovne sa ale v 3D priestore dajú jednoducho rozložiť na jednotlivé vrstvy.

Presunutím diagramov UML zo štandardného 2D do 3D priestoru tak umožníme jednoduchšiu a prehľadnejšiu vizualizáciu aj zložitých diagramov v modernej trojrozmernej grafike. Využívaním výhod tretej dimenzie môžeme dosiahnuť oveľa čitateľnejšie

a jednoduchšie modely aj pri komplexných systémoch. Navyše s nárastom používania virtuálnej a rozšírenej reality sa zdá, že v budúcnosti bude modelovanie systémov v 3D ešte viac rozšírené [2].

2.5 Analýza zdrojového kódu

Analýzu zdrojového kódu môžeme chápať ako spôsob automatizovaného testovania s cieľom odhaliť chyby či iné problémy softvéru [10]. Získame tak model (alebo modely) systému, s ktorými môžeme ďalej pracovať. Takýto model môže byť vytvorený pomocou rôznych automatizovaných nástrojov určených na analýzu zo zdrojového kódu alebo z vykonávaného programu [10]. Na základe toho rozdeľujeme analýzu na statickú a dynamickú.

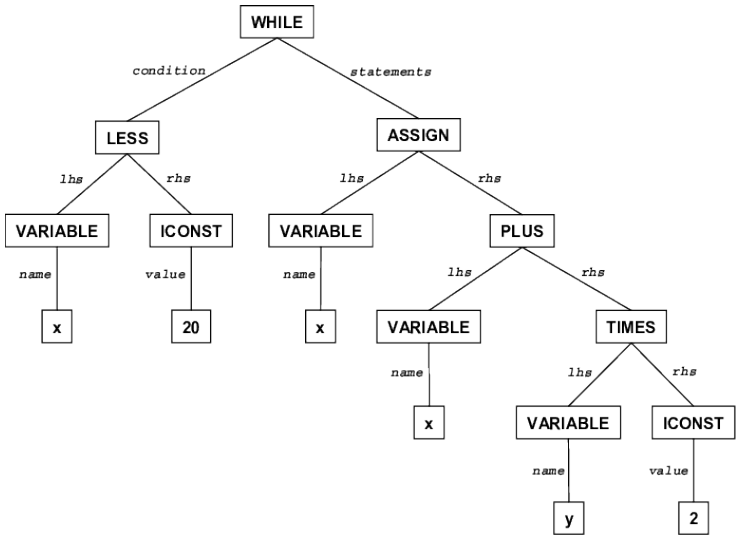
- Statická analýza - Analýza sa vykonáva skúmaním kódu bez skutočného vykonania programu. To môže pomôcť odhaliť chyby a pochopiť jednotlivé súvislosti medzi objektami už v rannej fáze vývoja systému.
- Dynamická analýza - Analýza sa vykonáva v reálnom čase vykonávania programu alebo počas jeho ladenia (angl. debugovania) s cieľom odhaliť možné, menej viditeľné chyby alebo zložitejšie súvislosti programu.

2.6 Parsovanie zdrojového kódu

Parsovanie alebo syntaktická analýza je proces, v ktorom analyzujeme príkazy alebo reťazce symbolov v programovacích jazykoch alebo dátových štruktúrach zodpovedajúce pravidlám formálnej gramatiky, teda súboru pravidiel, ktorý opisuje syntax jazyka [22].

Výstupom parsera je organizovaná dátová štruktúra, zvyčajne strom. Stromy sa využívajú práve preto, že predstavujú jednoduchý a prirodzený spôsob práce s kódom a rôznou úrovňou jeho detailov. Môže sa jednať o syntaktický strom (ang. Parse Tree) alebo abstraktný syntaktický strom (ang. Abstract Parsing Tree), skrátene AST. Tieto dve štruktúry sú si veľmi podobné. Obidve sú stromy, ale líšia sa v tom, ako presne a do akých detailov sú schopné reprezentovať daný zdrojový kód. Technicky vzato by sa Parse Tree mohol nazývať aj Concrete Syntax Tree, skrátene CST, teda konkrétny syntaktický strom, pretože by mal konkrétnejšie odrážať skutočnú syntax v porovnaní s AST [22].

CST obsahuje všetky tokeny, ktoré sa objavili v programe. Naopak AST predstavuje akoby vylepšenú verziu, v ktorej sa odstránia všetky informácie, ktoré by mohli byť odvodené alebo nie sú dôležité pre pochopenie kódu. V našej práci je teda logickejšie využívať pri parsovaní štruktúru AST.



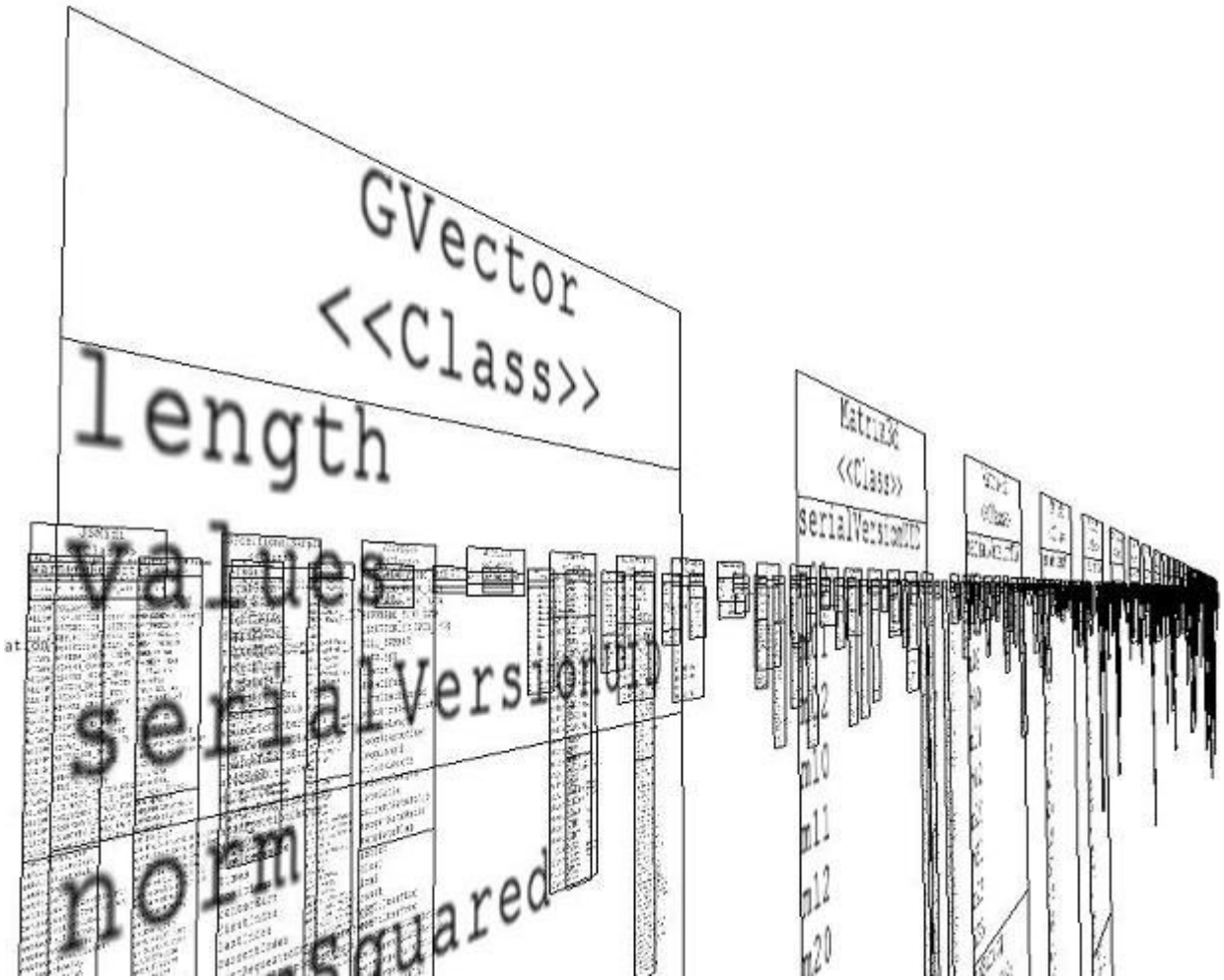
Obr. 2.5: Ukážka abstraktného syntaktického stromu z [3]

2.7 Existujúce riešenia

V súčasnosti existuje viacero rôznych riešení na vizualizáciu UML diagramov vo viacrozmernom priestore. Navyše sa toto odvetvie stále rýchlejšie rozvíja a objavujú sa taktiež funkčné riešenia vizualizácie s využítím virtuálnej či rozšírenej reality.

2.7.1 X3D-UML

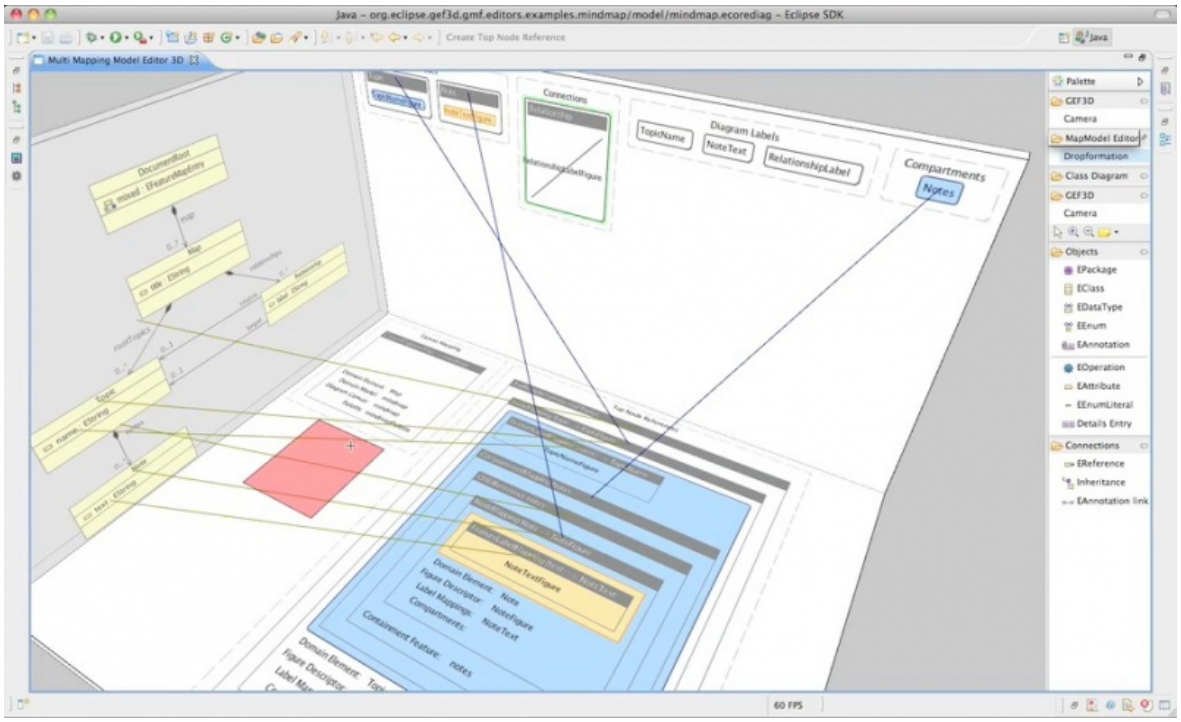
Extensible 3D (X3D) umožňuje používateľom sťahovať, zobrazovať a interagovať s 3D obsahom prostredníctvom modulu webového prehliadača alebo samostatného prehliadača. X3D poskytuje bohatú sadu funkcií navrhnutých pre vytváranie 3D svetov, ktoré je možné aplikovať pri modelovaní softvéru. Touto myšlienkou sa zaoberal aj Paul McIntosh, ktorý vo svojej práci [16] uvádza možné výhody X3D-UML (spojenia X3D a UML) a využíva práve X3D na vizualizáciu stavového diagramu ako aj diagramu tried v [15].



Obr. 2.6: Príklad diagramu tried z práce [15]

2.7.2 GEF 3D

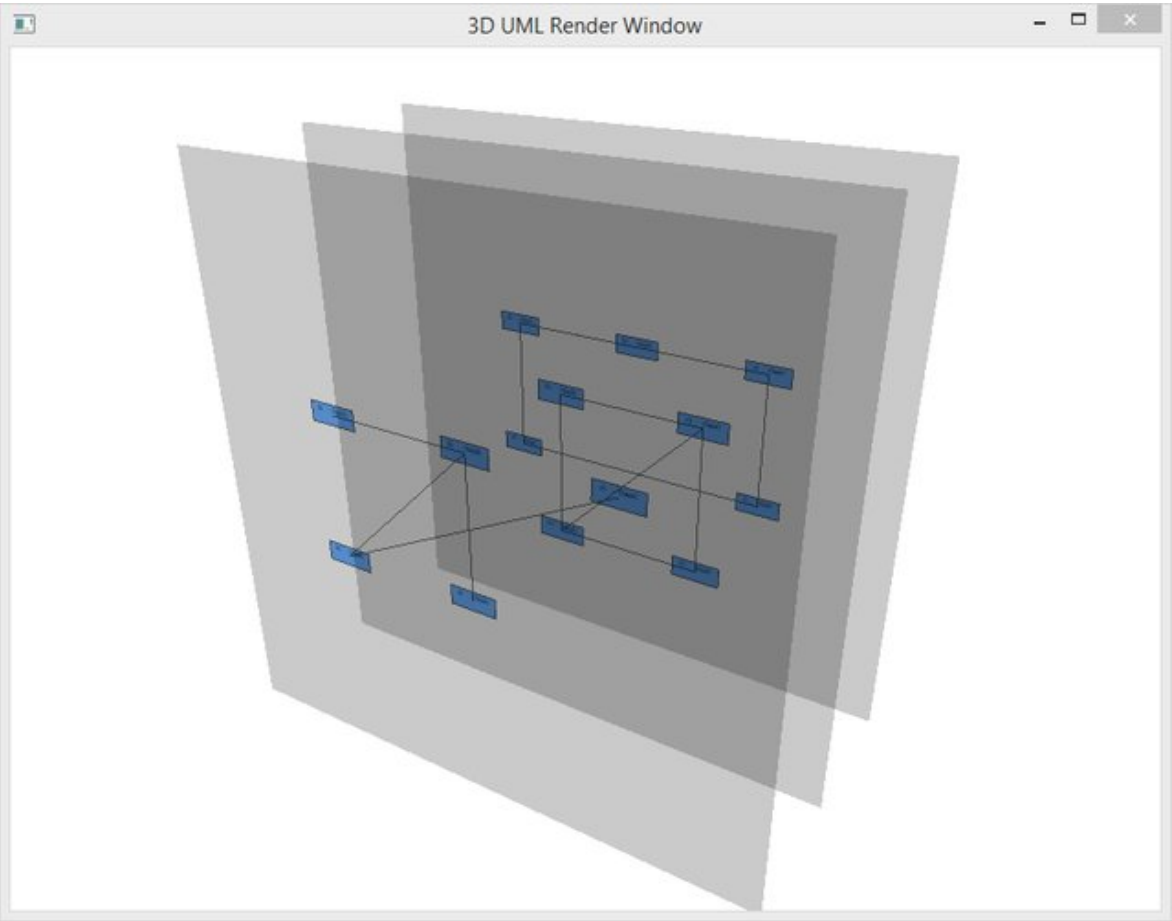
GEF 3D je rozšírenie Eclipse, ktoré je založené na známom dvojrozmernom grafickom editovacom nástroji Eclipse GEF. Umožňuje nám vytvárať 3D diagramy, 2D diagramy a taktiež kombinovať 3D a 2D diagramy. V konečnom dôsledku sa skôr využíva na vizualizáciu 2D diagramov v 3D priestore, nie na samotnú tvorbu 3D diagramov [28]. Táto vizualizácia sa dosahuje umiestnením 2D diagramov do vrstiev a následne môžu byť 3D prvky použité na zobrazenie spojení medzi jednotlivými vrstvami alebo medzi jednotlivými elementami na vrstvách.



Obr. 2.7: Príklad diagramu vytvoreného v GEF3D [28]

2.7.3 Ogre 3D prototyp

Spomínaný prototyp je vyvíjaný na pôde FIIT STU. Ogre 3D (Open Source 3D Graphics Engine) je engine podporujúci vykresľovanie 3D objektov v reálnom čase. Tento prototyp vizualizuje sekvenčný diagram na viacerých vrstvách rozmiestnených v 3D priestore a následne z tohto diagramu vie zostaviť aj diagram tried, ako uvádzajú autori v [6].



Obr. 2.8: Sekvenčný diagram zobrazený na viacerých vrstvách za použitia prototypu z [6]

2.7.4 Zhrnutie existujúcich riešení a ich zhodnotenie

V súčasnosti je existujúcich riešení na danú problematiku stále viac, no v našej práci sme spomenuli len tie najznámejšie. Väčšina z nich využíva 3D priestor na zobrazenie 2D UML diagramov na viacerých vrstvách. Toto umožňuje zjednodušenie komplexných diagramov a lepšiu orientáciu v nich. Taktiež tým, že je zachované 2D zobrazenie jednotlivých prvkov diagramov, je pre používateľa jednoduché pochopiť dané modely aj pri ich zobrazení v trojdimenzionálnom priestore. Niektoré riešenia v danej oblasti

naopak transformujú prvky 2D diagramov na trojrozmerné objekty, čím však môžu zmiasť používateľov pri prvom stretnutí s takýmto typom diagramu.

2.7.5 Prototyp používaný v práci

Jedným z ďalších možných riešení je prototyp navrhnutý a implementovaný Ing. Jurajom Vincúrom na pôde FIIT STU. V rámci našej práce budeme s týmto prototypom pracovať a všetky doplnenia funkcionality budú navrhované a implementované do tohto prototypu.

Kapitola 3

Návrh riešenia

Prvú časť tejto práce sme venovali analýze problémovej oblasti. Zaoberali sme sa najmä vizualizáciou UML diagramov v trojrozmernom priestore a už existujúcimi riešeniami v tejto oblasti a došli sme k záveru, že transformácia zobrazenia do trojrozmerného priestoru by mohla pomôcť pri orientácii v komplexnejších UML diagramoch.

Na základe poznatkov získaných z analýzy môžeme pokračovať v návrhu riešenia, ktoré bude následne implementované do prototypu.

3.1 Ciele práce

Cieľom práce je vytvoriť prototyp vizualizácie sekvenčného diagramu zo zdrojového kódu v trojrozmernom priestore. Naša práca sa trochu líši od už existujúcich riešení v tom, že diagram je možné zo zdrojového kódu vytvárať počas jeho ladenia. Za kladný výsledok práce budeme považovať vizualizáciu sekvenčného diagramu v trojrozmernom priestore s možnosťou jeho editovania. Editovanie diagramu budeme vykonávať prostredníctvom detekovania cyklov v programe a ich následným obalením do fragmentu. Trojrozmerný priestor využijeme vhodným usporiadaním viacerých diagramov na vrstvách.

3.2 Požiadavky

Na základe analýzy súčasného stavu a už existujúcich riešení sme sformulovali nasledujúce požiadavky:

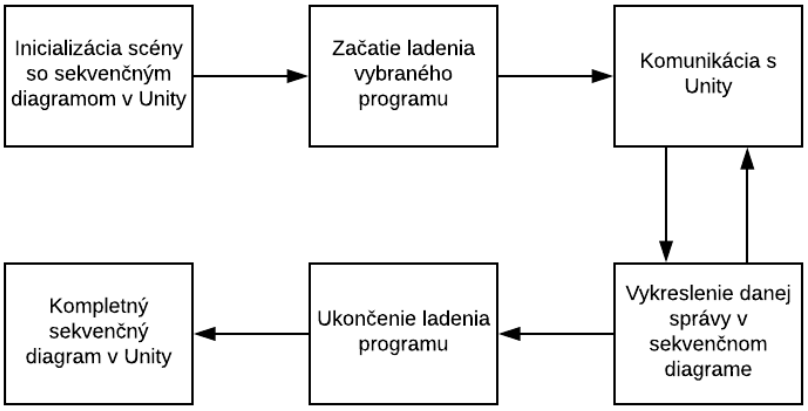
- Vizualizácia sekvenčného diagramu má byť možná počas ladenia programu
- Vizualizácia má prebiehať v trojrozmernom priestore. Je nutné využiť vlasnosti, ktoré nám 3D zobrazenie prináša a dostatočne využiť možnosti rozmiestnenia prvkov v priestore.

- Pri vizualizácii je potrebné dodržať zaužívanú notáciu jednotlivých prvkov diagramu podľa štandardnej UML notácie pre daný typ diagramu.
- Počas vizualizácie popri ladení programu je nutné, aby používateľ vedel s kódom interagovať a pridávať jednotlivé body prerušenia (anglicky breakpoint).
- Vizualizovaný diagram bude možné editovať detekciou opakujúcich sa volaní (teda cyklov v programe) a ich obalením do fragmentu.

3.3 Opis používaného prototypu

Daný prototyp vytvára 3D sekvenčný diagram počas ladenia programu. Takýto princíp vizualizácie zdrojového kódu môže byť výhodný pri hľadaní chyby v programe alebo pri potrebe lepšie sa oboznámiť s kódom. Samotný 3D sekvenčný diagram je vizualizovaný v programe Unity a podporovaný programovací jazyk, z ktorého vieme diagramy vytvárať, je Python.

Na obrázku nižšie môžeme vidieť ako prebieha vykresľovanie diagramu od začiatku ladenia. Ako prvé z ladiaceho okna programu Visual Studio Code inicializujeme sekvenčný diagram v Unity odoslaním inicializačnej správy. Následne môžeme začať ladenie programu. Pri krokovaní (spôsob ladenia) zdrojového kódu sa stále pri narazení na bod prerušenia odošle správa pomocou komunikačného servera medzi Visual Studio Code a Unity obsahujúca potrebné informácie na vykreslenie daného volania v diagrame sekvencií v Unity. Správa je reprezentovaná dátovým typom Json a nesie informácie o triede, funkcii, ktorá danú interakciu vyvolala, riadku a pod. Komunikácia Visual Studio Code s Unity a následné vykreslenie daného prvku sa opakuje až do ukončenia ladenia programu, kedy už môžeme v Unity vidieť scénu s kompletným sekvenčným diagramom aký vidíme napríklad na obrázku 3.3.



Obr. 3.1: Proces vizualizácie diagramu v prototype

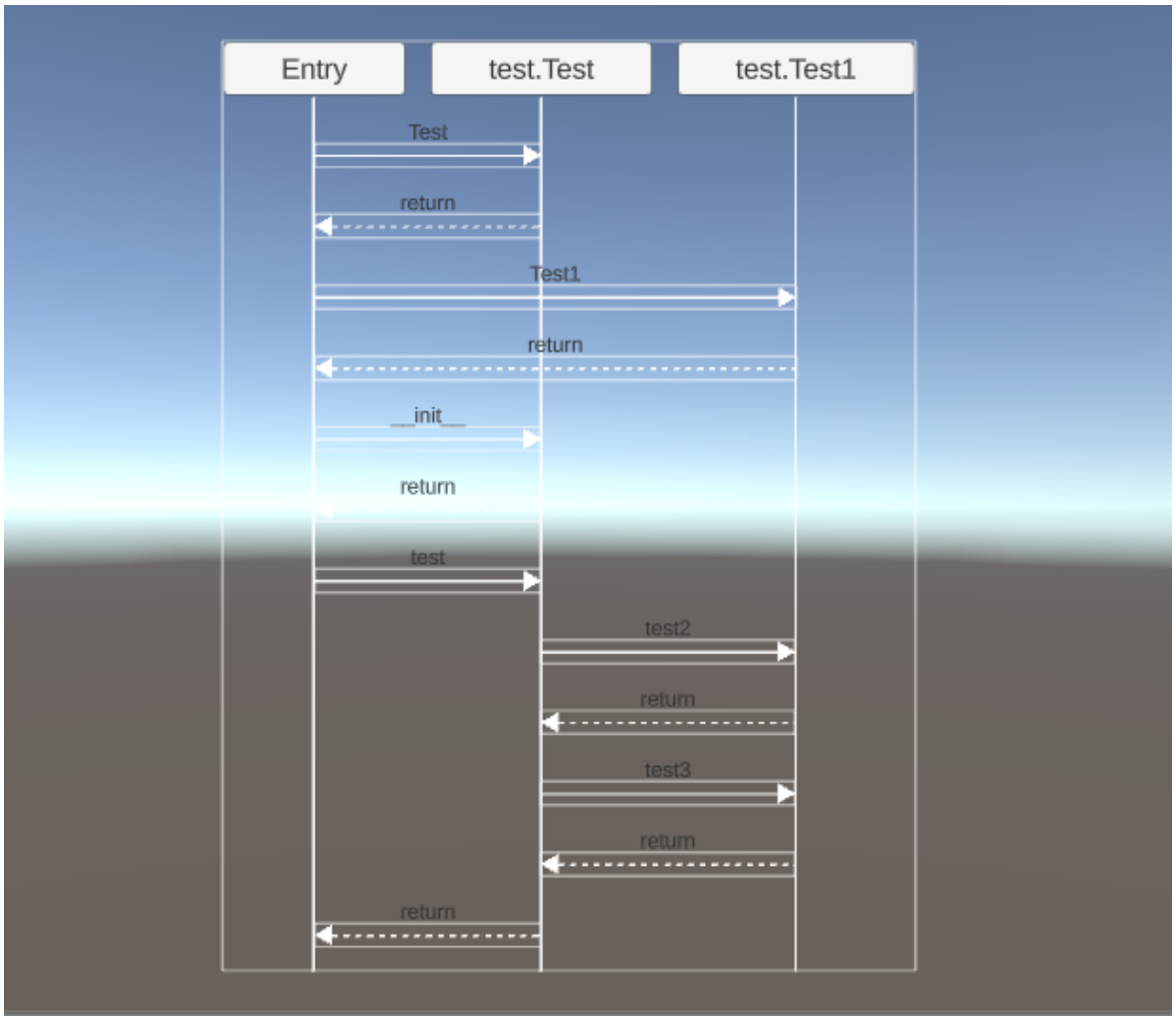
3.3.1 Ukážka vizualizovaného sekvenčného diagramu

Na obrázku 3.3 môžeme vidieť sekvenčný diagram, ktorý vznikol pri ladení zdrojového kódu z obrázku 3.2. Do sekvenčného diagramu sú pridané len tie funkcie, resp. volania, ktoré sú vo vývojovom prostredí počas ladenia označené bodom prerušenia.

```
1   class Test(object):
2
3       def __init__(self):
4           self.t = Test1()
5
6       def test(self):
7           self.t.test2()
8           self.t.test3()
9
10  class Test1(object):
11
12      def test2(self):
13          print 2
14
15      def test3(self):
16          print 3
17
18  if __name__=="__main__":
19      Test().test()
20
```

Obr. 3.2: Ukážka použitého zdrojového kódu

Daný kód obsahuje 2 triedy s názvami Test a Test1. Triedy sú vo výslednom sekvenčnom diagrame zobrazované ako biele obdĺžniky s názvom danej triedy. Tieto triedy predstavujú jednotlivých účastníkov interakcie. Z týchto tried následne vychádzajú čiary života, ktoré sú vedené vertikálne pozdĺž celej scény a reprezentujú plynutie času tak, ako je to v diagrame sekvencií zaužívané. Správy sú zobrazované jednoduchými šípkami vedenými v smere interakcie so začiatkom a koncom umiestneným na čiare života korešpondujúcej triedy v časovom slede podľa ich volania počas vykonávania programu. Návratové správy sú zobrazené prerušovanou šípkou smerujúcou v opačnom smere ako pôvodné volanie. Nad oboma typmi správ môžeme vidieť názov metódy, ktorá interakciu vyvolala.



Obr. 3.3: Sekvenčný diagram vizualizovaný použitým prototypom

3.4 Použité vývojové prostredia a nástroje

Na implementáciu vizualizácie budeme používať Unity. Unity je multiplatformový herný engine vyvinutý spoločnosťou Unity Technologies. V súčasnosti je Unity široko využívané, keďže môže byť použité na vytváranie dvojrozmerných ale aj trojrozmerných hier, ako aj rôznych simulácií pre mnohé platformy v reálnom čase [26]. Unity je taktiež preferovaným vývojovým nástrojom pre tvorcov XR (virtuálnej a rozšírenej reality).

Ďalším vývojovým prostredím, ktoré budeme v našej práci používať, je Visual Studio Code. Visual Studio Code je editor zdrojového kódu vyvinutý spoločnosťou Microsoft pre systémy Windows, Linux a MacOS [18]. Obsahuje mnoho funkcií, vďaka ktorým predstavuje výkonný vývojársky nástroj.

Visual Studio Code nám taktiež umožňuje používanie mnohých rozšírení tretích strán ako aj vytvorenie vlastného rozšírenia podľa potreby. Taktiež nám umožňuje vytvoriť vlastné rozšírenie ladenia programu, ktoré potrebujeme, aby sme vedeli komunikovať s Unity, čo je aj jedným z hlavných dôvodov, prečo sme si dané prostredie

vybrali.

3.5 Technológie použité v prototype

V tejto časti si priblížime technológie, ktoré sú v prototype používané a teda v ich používaní budeme aj naďalej pokračovať.

- C# - Je objektovo orientaný programovací jazyk vytvorený spoločnosťou Microsoft. Microsoft si za základ pre tento nový jazyk zobral jazyky C++ a Java [17]. Práve jazyk C# sa používa na vytváranie skriptov v Unity. Pomocou týchto skriptov môžeme ovládať vizualizované objekty na scéne.

- Javascript - Je objektovo orientovaný programovací jazyk. V našom prototype je využitý pri implementácii rozšírenia vo Visual Studio Code, ktoré slúži na výmenu správ medzi vývojovým prostredím a Unity.

- Python - Python je vyvíjaný ako open source projekt. Podporuje objektovo orientované, štruktúrované aj funkcionálne programovanie. Je to dynamický typový jazyk, podporuje veľké množstvo vysokoúrovňových dátových typov [23].

- Pylint - Predstavuje nástroj na analýzu kódu v jazyku Python. V našej práci je veľmi dôležitý, využívame ho na parsovanie zdrojového kódu, ktorý následne vizualizujeme.

3.6 Komunikácia s Unity

Na sprostredkovanie komunikácie s Unity slúžia správy v dátovom formáte Json. Json predstavuje textový spôsob zápisu dát, ktorý je nezávislý na počítačovej platforme [9]. Tento formát bol zvolený najmä preto, že je jednoducho čitateľný, pochopiteľný a pre počítače ľahko spracovateľný. V programe rozlišujeme 2 typy možných správ. Tieto správy v sebe obsahujú informácie potrebné na vykreslenie jednotlivých elementov diagramu.

Prvý typ slúži na inicializáciu súboru (diagramu) a obsahuje dáta potrebné pre celý objekt opisujúce základnú štruktúru.

Štruktúra daného typu vyzerá nasledovne:

```
{
"action":"G_CLS_DIA",
"payload":{
"relationships":[],
"loops":[],
```

```
"classes":{},
"globals":{},
"fileMap":{}
}
```

Opis dôležitých parametrov daného typu:

- relationships - pole vzťahov medzi triedami, v ktorých dochádza k interakcii
- loop - pole obsahujúce začiatočné a koncové pozície cyklov v programe
- classes - pole obsahujúce všetky triedy v programe a základné informácie o nich
- filemap - pole obsahujúce zoznam súborov

Druhý typ sa posiela vždy, keď sa v programe narazí na bod prerušenia. Obsahuje informácie o interakcii, ktorá sa bude následne zobrazovať v diagrame.

Štruktúra daného typu vyzerá nasledovne:

```
{
"action":"S_DRW_TRC",
"payload":{
"calls":[]
}
```

3.7 Úpravy prototypu

V tejto časti sa bližšie zameriame na úpravy, ktoré budeme následne implementovať do prototypu.

3.7.1 Vizualizácia viacerých diagramov

Prvou úpravou, ktorú budeme musieť implementovať, je možnosť vizualizácie viacerých diagramov. Súčasný stav prototypu umožňuje vykreslenie len jedného diagramu, no naším cieľom je zobraziť viacero diagramov na vrstvách.

Prototyp upravíme tak, aby pri každej inicializácii vytvoril novú inštanciu vizualizovaného diagramu, ktorej sa pridelí unikátne identifikačné číslo (ID), aby sme tieto diagramy vedeli od seba odlíšiť a aj naďalej s nimi pracovať.

3.7.2 Odstránenie spätných správ

Ako sme mohli vidieť na obrázku 3.3, po každom volaní funkcie, ktoré je zobrazené v podobe asynchrónnej správy v sekvenčnom diagrame, sa nám zobrazuje aj spätná

správa. Aby sme udržali diagram prehľadnejší, rozhodli sme sa tieto správy z diagramu odstrániť a pri ďalšom vývoji ich už nepoužívať.

3.7.3 Pridanie ohraničenia k diagramu

Keďže je naším cieľom zobrazovať viacero diagramov, je dôležité, aby sa dali jednoducho od seba odlíšiť. Preto pridáme každému diagramu ohraničenie, ktoré ho obalí a jednoznačne určí, ktoré prvky danému diagramu patria.

Pri každom pridaní objektu tak budeme musieť skontrolovať šírku a výšku diagramu a ak sa zmení, budeme musieť upraviť aj ohraničenie. Pseudokód možného riešenia:

```
function upravOhranicenie(diagram) {
    sirka, vyska
    sirka <- diagram.ziskajAktualnuSirku()
    vyska <- diagram.ziskajAktualnuVysku()
    if (bolPridanyObjekt(diagram)) then
        diagram.aktualizujOhranicenie(sirka, vyska)
    end if
}
```

3.7.4 Pridanie pozadia

Aby bol diagram ľahšie čitateľný, rozhodli sme sa za každý vizualizovaný diagram pridať jednofarebné pozadie. Pri výbere sme zvolili sivú farbu, ktorá kontrastuje so zobrazenými správami a tie sú tak lepšie viditeľné.

Pozadie sa pridá každému diagramu už pri jeho inicializácii a následne sa bude prispôsobovať veľkosti diagramu podobne ako ohraničenie.

3.7.5 Pridanie aktivácií na čiary života

Ak chceme dodržať zaužívanú notáciu modelovania sekvenčných diagramov, mali by sme pridať na čiary života jednotlivé bloky aktivácií. Tie by boli zobrazené ako obdĺžniky v mieste výskytu správy.

Počas vykresľovania by sme sa museli zamerať na to, či sa na danej čiare života už správa nachádza a je potrebné blok aktivácie len predĺžiť, alebo sa jedná o prvý výskyt a je potrebné ho vytvoriť. Zároveň je potrebné určiť aj to, či nastáva interakcia medzi rovnakými čiarami života ako v predošlom kroku, alebo sa jeden z aktérov zmenil.

3.8 Návrh pridanej funkcionality

V tejto časti sa budeme sústrediť na detailnejší návrh funkcionality, ktorú budeme následne implementovať.

3.8.1 Zobrazovanie diagramov na vrstvách

Keďže sa nachádzame v trojrozmernom priestore, môžeme využiť na vizualizáciu diagramu viacero vrstiev. Vrstvy v Unity umožňujú vykresľovanie alebo osvetlenie iba časti scény. Taktiež sa používajú na selektívne usporiadanie objektov, napríklad kvôli zabráneniu kolíziám. V tejto práci budeme vrstvy využívať na zobrazenie viacerých sekvenčných diagramov vedľa seba alebo za sebou. Takýmto spôsobom môžeme vizualizovať viacero diagramov v prostredí naraz, pričom všetky budú dobre viditeľné a čitateľné.

V Unity si vieme vytvoriť viacero vrstiev, ku ktorým vieme následne počas vizualizácie pristupovať a umiestňovať na nich jednotlivé objekty, čo v našom prípade bude sekvenčný diagram a jeho prvky. Sekvenčné diagramy budeme rozlišovať pomocou identifikačného čísla, ktoré nám pomôže každý diagram a jeho prvky správne umiestniť na danú vrstvu. Aby sa zmena vrstvy prejavila na všetkých prvkoch diagramu, je nutné ich rekurzívne prejsť a umiestniť ich na príslušnú vrstvu.

Pseudokód možného riešenia sa nachádza nižšie. Pre jednoduchosť uvedieme len časť funkcie pridajDiagram(), ktorá sa týka úpravou vrstiev:

```
function pridajDiagram() {
    vrstva
    if (prvyPridany() == true) then
        zmenVrstvu(diagram, 1)
        vrstva += 1
    else
        zmenVrstvu(diagram, vrstva)
        diagram.pozicia = prev.pozicia + posunutie
        vrstva += 1
    end if
}

function zmenVrstvu(diagram, vrstva) {
    diagram.nastavVrstvu(vrstva)
    nastavVrstvuKomponentom(diagram, vrstva)
}
```

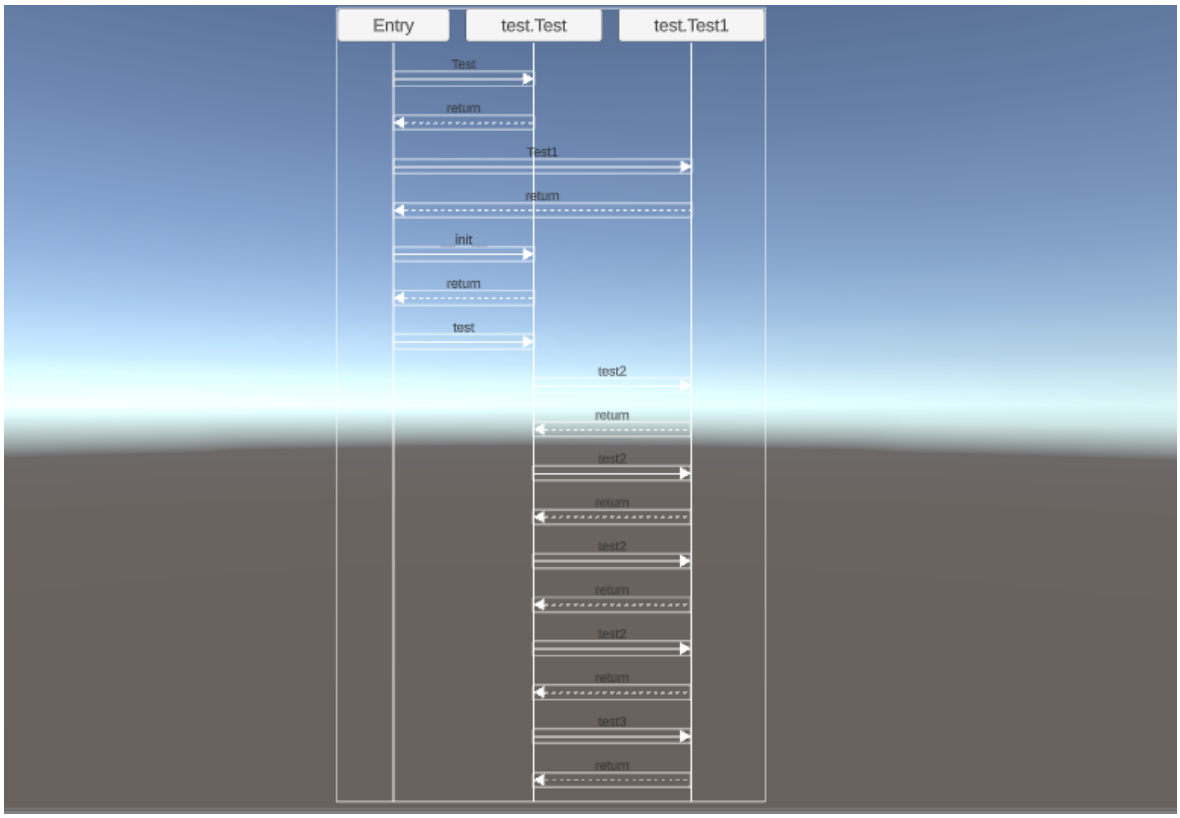
3.8.2 Detekcia jednoduchého cyklu

Ako môžeme vidieť na obrázku 3.4 medzi triedou Test a Test1 sa viackrát opakuje volanie tej istej metódy. To znamená, že v programe sa nachádza cyklus a tieto správy by mali byť správne zaobalené do fragmentu.

Prvé triviálne riešenie, ktoré môžeme skúsiť pre cyklus s jedným volaním, je porovnávanie aktuálnej a predchádzajúcej správy. Ak sú správy zhodné, znamená to, že sa v programe nachádza cyklus.

Pseudokód detekovania jednoduchého cyklu by v takom prípade vyzeral nasledovne:

```
function detekujCyklus(predoslaSprava, aktualnaSprava) {
    existujeCyklus
    if (porovnaj(predoslaSprava, aktualnaSprava)) then
        existujeCyklus <- true
    else
        existujeCyklus <- false
    end if
    return existujeCyklus
}
```



Obr. 3.4: Ukážka opakovaných volaní

3.8.3 Detekcia zložitejšieho cyklu

Je jasné, že vyššie uvedené riešenie nebude fungovať pre zložitejší cyklus, v ktorom sa nachádza viac volaní metód, ktoré sa medzi sebou líšia. V takom prípade je nutné uchovávať všetky doterajšie volania a nie len to posledné. Pri každej novej interakcii by bolo nutné porovnať pridávanú správu so všetkými predchádzajúcimi a brať do úvahy všetky možné dvojice, trojice a pod. predošlých správ a zisťovať či sa nezhodujú, teda či sa nejedná o cyklus.

Dané riešenie je ale výpočtovo aj pamäťovo náročné, keďže je nutné prechádzať pri každom pridaní správy zoznamom predošlých správ až niekoľko krát a môže spôsobiť značné spomalenie programu. Taktiež môže nastať problém, že bude ako cyklus označená sekvencia volaní, ktorá sa skutočne v cykle ani nenachádza, čo môže spôsobiť používateľovi značné zmätenie.

3.8.4 Detekcia vnorených cyklov

Predošlé riešenie je náročné samo o sebe a teda jeho rozšírenie na podporu detekcie vnorených cyklov je prakticky nemožné, ak chceme, aby program nebol pomalý a fungoval správne.

Riešením by mohla byť práve kombinácia vlastností statickej a dynamickej analýzy. Diagram budeme stále vizualizovať z dynamickej analýzy, ale informácie o cykloch vieme získať statickou analýzou, pri parsovaní zdrojového kódu, kedy môžeme v pomocnej metóde testovať či sa vo vytváranom strome volaní nachádza uzol obsahujúci kľúčové slovo for alebo while. Túto informáciu o výskyte cyklu, začiatočnom a konečnom riadku, si vieme uložiť do Jsonu, pomocou ktorého komunikujeme s Unity a následne by sme zabezpečili editáciu sekvenčného diagramu zaobalením príslušných volaní do fragmentu. Umiestňovanie správ na čiary života v danej časovej postupnosti by ostalo nezmenené.

Možné riešenie detekcie cyklov statickou analýzou:

```
function najdiCykly(subor) {
    pozicie, strom
    strom <- subor.parsuj()
    for uzol v strome
        if jeInstancia(uzol, for) ||
        jeInstancia(uzol, while) then
            pozicie.pridaj(urcPozicie(uzol)
        end if
    end for
```

```
    return pozicie
}
```

3.8.5 Obalenie fragmentom

Ak sa správa nachádza v cykle, je potrebné ju obaliť fragmentom. Pri vnorených cykloch potrebujeme dané fragmenty do seba vnárať. Po pridaní ďalšej správy sa hranice fragmentu automaticky rozšíria.

Pseudokód možnej implementácie by vyzeral nasledovne:

```
function obalFragmentom() {
    vlavo, dole, vpravo, spravy

    spravy <- vsetky sprvy patriace do cyklu
    for sprava v spravy
        if jeVCykle(sprava) then
            vlavo <- najviacVlavo(sprava)
            hore <- najviacDole(sprava)
            vpravo <- najviacVpravo(sprava)
        end if
    end for

    nastavPoziciuVlavo(vlavo)
    nastavPoziciuDole(dole)
    nastavPoziciuVpravo(vpravo)

    nastavVysku(spravy.pocet * vyskaSpravy)
}
```

Kapitola 4

Implementácia funkcií do prototypu

V našej práci budeme pracovať s prototypom už spomínaným v návrhu riešenia. Všetky navrhované funkcie sú implementované do tohto prototypu. Pri implementácii sme modifikovali triedy SequenceDiagram.cs, Message.cs a TCPService.cs. Taktiež vznikli nami vytvorené triedy s názvami Loop.cs, Background.cs, SequenceDiagramObject.cs . Pri implementácii detekcie cyklov sme upravili aj projekt Diapy a obsah odosielaného Jsonu určeného na komunikáciu. Na porovnanie je priečinok obsahujúci pôvodný prototyp odovzdaný spolu s touto prácou.

4.1 Triedy a objekty v prototype

Hlavné triedy vyskytujúce sa v prototype a k nim prislúchajúce objekty sú:

- SequnceDiagram - predstavuje samotný objekt vizualizovaného diagramu,
- Message - predstavuje šípku, ktorá sa vykreslí v prípade interakcie,
- Loop - reprezentuje cyklus, teda fragment, ktorým sú správy nachádzajúce sa v cykle obalené,
- Background - predstavuje pozadie sekvenčného diagramu,
- TCPService - predstavuje triedu na spracovanie správ poslaných medzi Visual Studio Code a Unity,

4.2 Implementované funkcie

V ďalšej časti sa bližšie pozrieme na funkcie implementované do používaného prototypu.

4.2.1 Vizualizácia diagramov a ich pridanie na vrstvy

Pri pridaní nového diagramu je vytvorená inštancia triedy SequenceDiagram. Následne je diagram umiestený na príslušnú vrstvu. Implementácia umiestňovania je založená na pôvodnom návrhu. Po priradení diagramu na vrstvu je nutné prejsť rekurzívne všetky jeho komponenty a taktiež ich priradiť na danú vrstvu. Na to používame funkciu zobrazenú na obrázku A.2.

```
GameObject tmp = Instantiate( seqTest , diagramObj);
tmp.name = "#" + this.diagramCount; // set name of generated diagram


if (diagramCount == 0)
{
    tmp.transform.position = new Vector3(0, 0, 0);
    tmp.layer = 1;
    SetLayerRecursively(tmp, 1);
    prevLayer = tmp.layer;
}
else
{
    RectTransform rt = (RectTransform)prev.transform;
    float width = rt.rect.width;
    tmp.transform.position = prev.transform.position + new Vector3(width + 50, 0, 0);
    tmp.layer = LayerMask.NameToLayer((prevLayer + 1).ToString());
    SetLayerRecursively(tmp, tmp.layer);
}
```

Obr. 4.1: Ukážka zdrojového kódu na umiestenenie diagramu na vrstvu

```
public static void SetLayerRecursively(GameObject go, int layerNumber)
{
    foreach (Transform trans in go.GetComponentsInChildren<Transform>(true))
    {
        trans.gameObject.layer = layerNumber;
    }
}
```

Obr. 4.2: Rekurzívna funkcia na pridelenie vrstvy komponentom

4.2.2 Pridanie pozadia a ohraničenia k diagramu

Pozadie je súčasťou každej inštancie sekvenčného diagramu a ohraničenie je jeho komponentom (UI LineRenderer). Ohraničenie sa automaticky zväčšuje podľa aktuálnej veľkosti diagramu.

```
public void refreshBounds()
{
    if (this.obj != null) {

        int spacing = 25;

        float x = this.obj.GetComponent<RectTransform>().sizeDelta.x; // get diagram width
        float y = this.obj.GetComponent<RectTransform>().sizeDelta.y; // get diagram height

        float posX = this.obj.GetComponent<RectTransform>().position.x; // get diagram width
        float posY = this.obj.GetComponent<RectTransform>().position.y; // get diagram height

        //draw lines by calculatin their positions

        var line = this.obj.GetComponent<LineRenderer>();

        line.SetPosition(0, new Vector3(posX +  0            , posY +  y/2 + spacing, 0));
        line.SetPosition(1, new Vector3(posX + -x/2 - spacing, posY +  y/2 + spacing, 0));
        line.SetPosition(2, new Vector3(posX + -x/2 - spacing, posY + -y/2 - spacing, 0));
        line.SetPosition(3, new Vector3(posX +  x/2 + spacing, posY + -y/2 - spacing, 0));
        line.SetPosition(4, new Vector3(posX +  x/2 + spacing, posY +  y/2 + spacing, 0));
        line.SetPosition(5, new Vector3(posX +  0            , posY +  y /2 + spacing, 0));

        line.material = new Material(Shader.Find("Sprites/Default"));
        line.startColor = Color.black;
        line.endColor = Color.black;

        line.startWidth = 5;
        line.endWidth = 5;

    }
```

Obr. 4.3: Vytváranie ohraničenia diagramu

4.2.3 Detekovanie cyklov v programe a ich následné obalenie fragmentom

Počas pridávania správy prebieha kontrola, či sa daná správa nachádza v cykle. Kontrola prebieha tak, že skontrolujeme či sa riadok správy nachádza v rozmedzí cyklov získaných statickou analýzou. Pokiaľ sa nachádza, tak musíme danú správu obaliť fragmentom. Pri vytváraní fragmentu sa následne kontroluje, či daný fragment už existuje. Ak nie, tak sa vytvorí nový fragment a pridá sa doň objekt danej správy, ktorá do cyklu patrí. Ak fragment už existuje, tak sa doň pridá daná správa a nový fragment sa nevytvára.

```
if (!existingLoop)
{
        var lop = Instantiate(loopPrefab, loopsTransform);
        int currLayer = (GameObject.Find("#" + (GameObject.Find("TCPService").GetComponent<TCPService>().getDiagCount() - 1))).layer;
        lop.layer = currLayer;


        loopsListAll.Add(lop);

        // get loop object script
        var lopScript = lop.GetComponent<Loop>();

        // init Object
        lopScript.setBounds(tmpBounds);
        lopScript.addLoopObject(tmpObject);

        for (int i = 0; i < subloop; i++)
        {
            lopScript.decreaseLevel();
        }
}
else
{
    tmpLoop.addLoopObject(tmpObject);
}
```

Obr. 4.4: Pridávanie správ do fragmentu

Kapitola 5

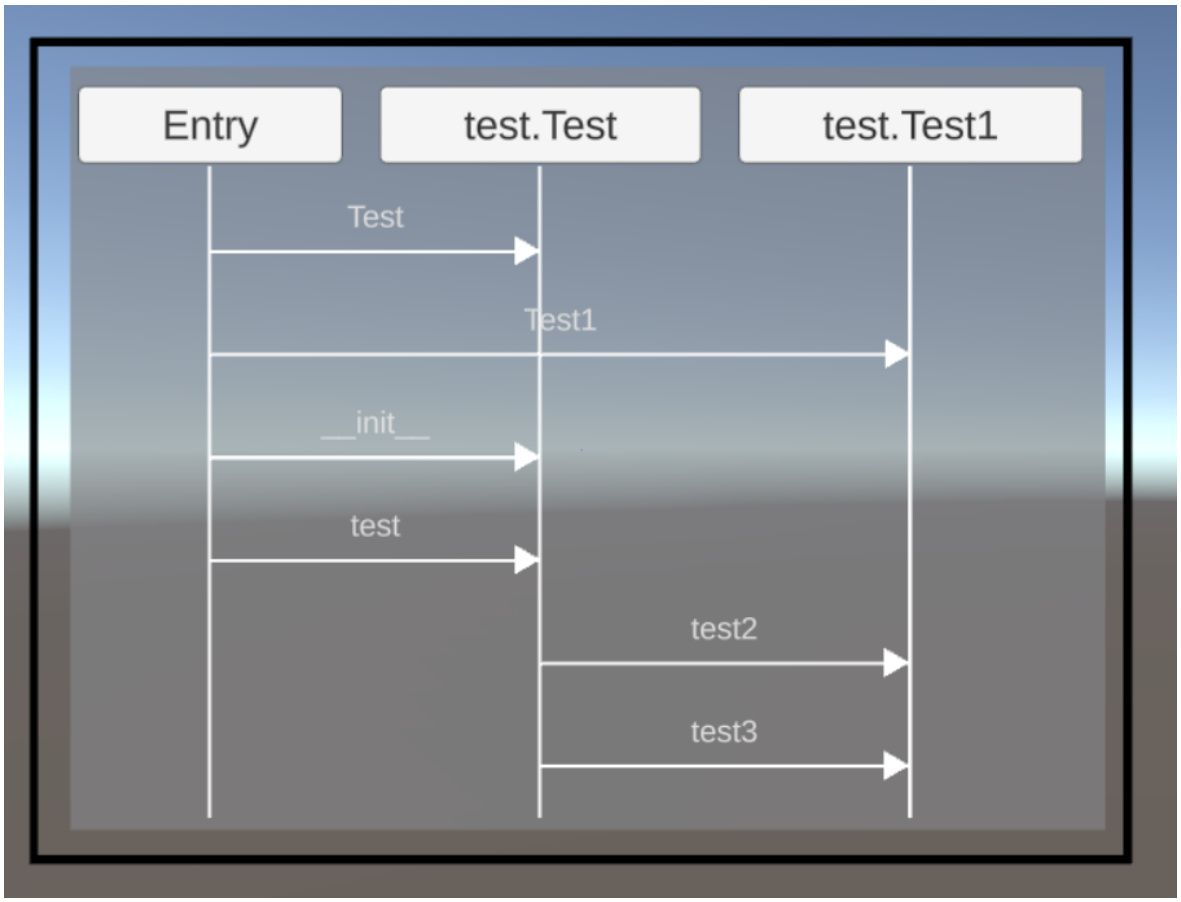
Testovanie prototypu

V tejto kapitole sa budeme venovať testovaniu nami implementovaných funkcií v prototype. Zdrojové kódy ku každému spomenutému testu sú opísané v prílohe obsahujúcej opis digitálneho média a budú odovzdané spolu s bakalárskou prácou.

5.1 Vizualizácia jednoduchého sekvenčného diagramu

Týmto testom sa uistíme, že všetky implementované funkcie upravujúce zobrazenie diagramu fungujú správne. Na obrázku 5.1 môžeme vidieť jednoduchý diagram zodpovedajúci zdrojovému kódu zobrazenému nižšie. Môžeme vidieť, že ohraničenie aj pridané pozadie sa zobrazujú správne a jasne určujú hranice daného diagramu.

```
class Test(object):
    def __init__(self):
        self.t = Test1()

    def test(self):
        self.t.test2()
        self.t.test3()

class Test1(object):
    def test2(self):
        print 2

    def test3(self):
        print 3

if __name__ == "__main__":
    Test().test()
```



Obr. 5.1: Ukážka jednoduchého diagramu

5.2 Zobrazenie sekvenčných diagramov na vrstvách

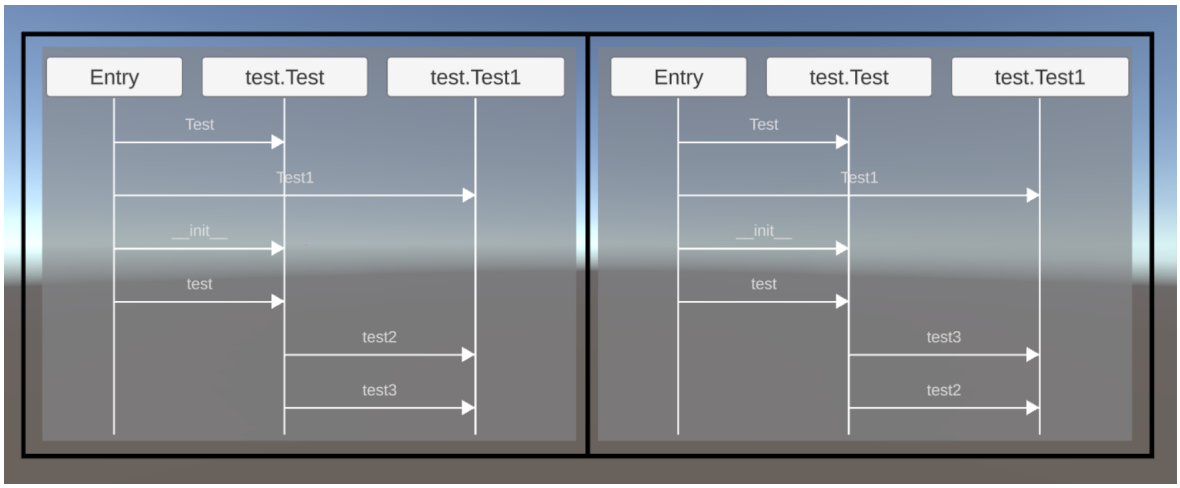
Daným testom zistíme, či funkcia na zobrazovanie diagramov na vrstvách funguje správne. Cieľom je vizualizovať 2 jednoduché diagramy, ktoré majú byť zobrazené vedľa seba na viacerých vrstvách. Pri tomto teste znova použijeme zdrojový kód z prvého testu, kde medzi jednotlivými volaniami len zmeníme poradie volaní funkcií test2() a test3(). Výsledok testu môžeme vidieť na obrázku 5.2.

5.3 Vizualizácia diagramu obsahujúceho cyklus

Tento test slúži na overenie správnosti detekcie cyklu a jeho následnej vizualizácie v diagrame. Funkciu test() z prvého testu upravíme nasledovne:

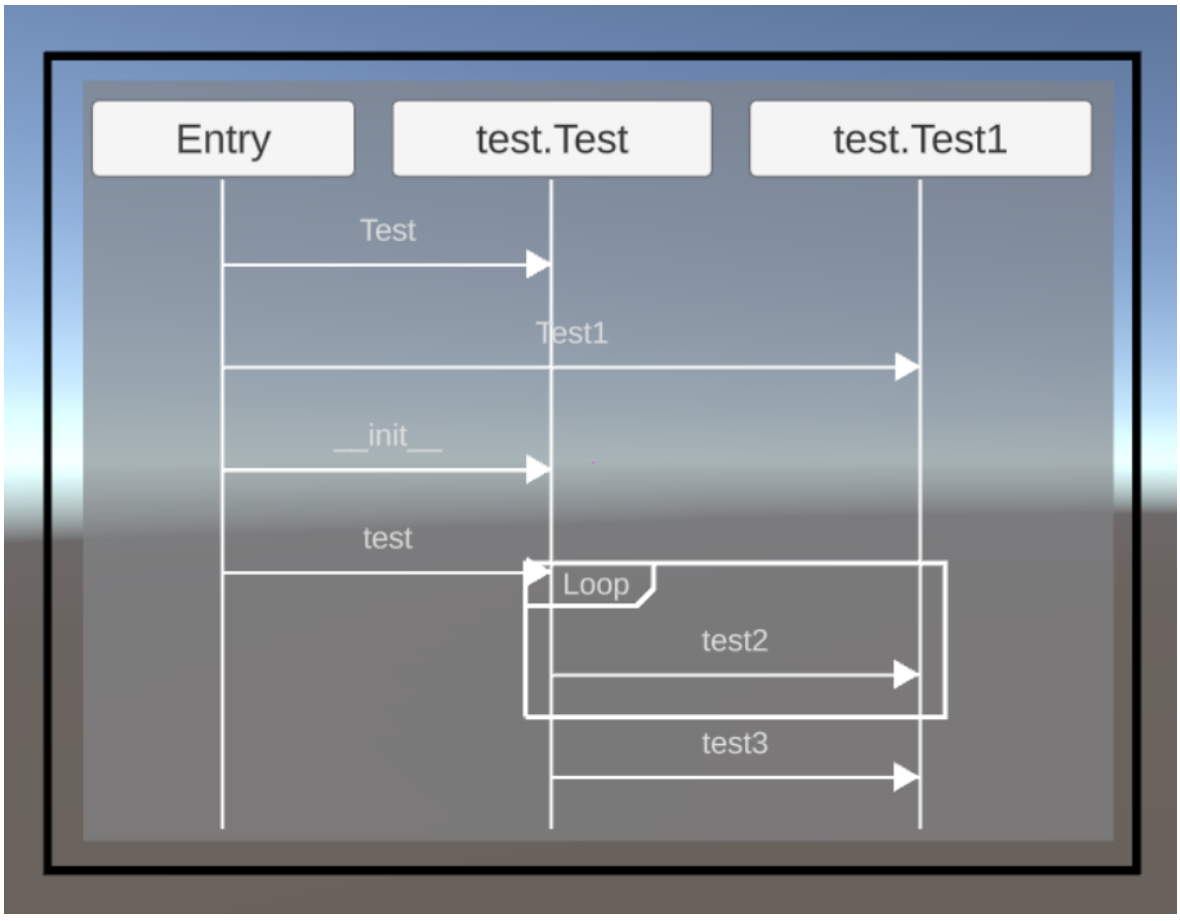
```
def test(self):
    for x in range(4):
        self.t.test2()
    self.t.test3()
```

Výsledok vizualizácie môžeme vidieť na obrázku 5.3. Vidíme, že správa zobrazujúca



Obr. 5.2: Ukážka zobrazenia na vrstvách

volanie funkcie test2() bola obalená fragmentom, zatiaľ čo test3() sa už nachádza mimo fragmentu tak ako to je v zdrojovom kóde.



Obr. 5.3: Ukážka detekcie jednoduchého cyklu

5.4 Vizualizácia viacerých cyklov v diagrame

Výsledok ďalšieho testu overí správanie prototypu, ak sa v zdrojovom kóde nachádza viacero cyklov ako napríklad v tomto prípade po úprave funkcie test():

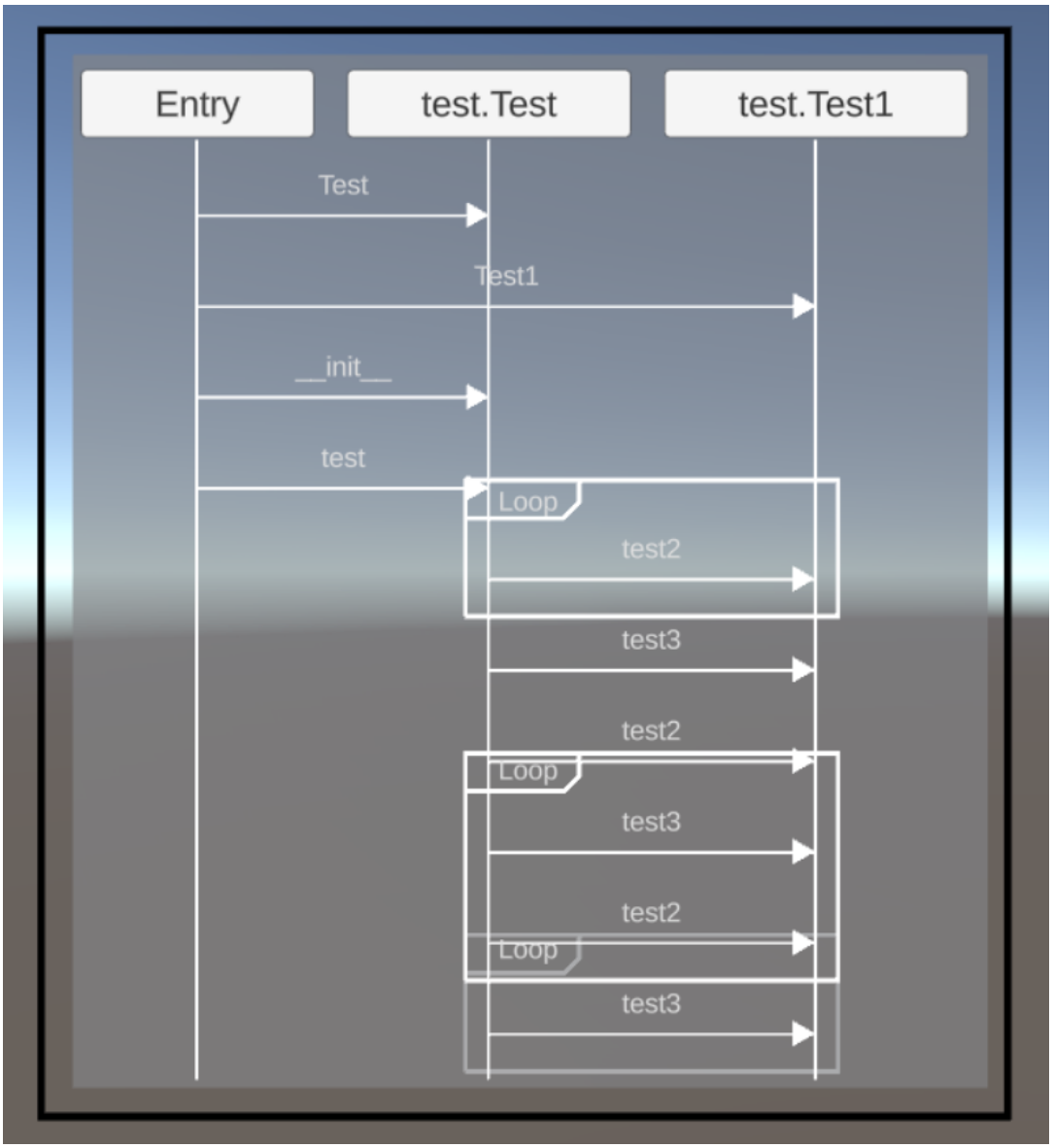
```
def test(self):
    for x in range(4):
        self.t.test2()
    self.t.test3()
    self.t.test2()
    for x in range(4):
        self.t.test3()
        self.t.test2()
    i = 1
    while i < 6:
        self.t.test3()
        i += 1
```

5.5 Vizualizácia vnorených cyklov v diagrame

Týmto testom overíme, či implementácia detekcie vnorených cyklov funguje správne. Znova si upravíme funkciu test() nasledovne:

```
def test(self):
    for x in range(4):
        self.t.test2()
        for x in range(2):
            self.t.test3()
            for x in range(2):
                self.t.test2()
            self.t.test3()
        self.t.test2()
    self.t.test3()
```

Na obrázku 5.5 môžeme vidieť, že v diagrame sú všetky cykly správne obalené fragmentami, vnútorný cyklus stále obaľuje vonkajší a fragmenty správne zahŕňajú správy, ktoré sa nachádzajú v daných cykloch.

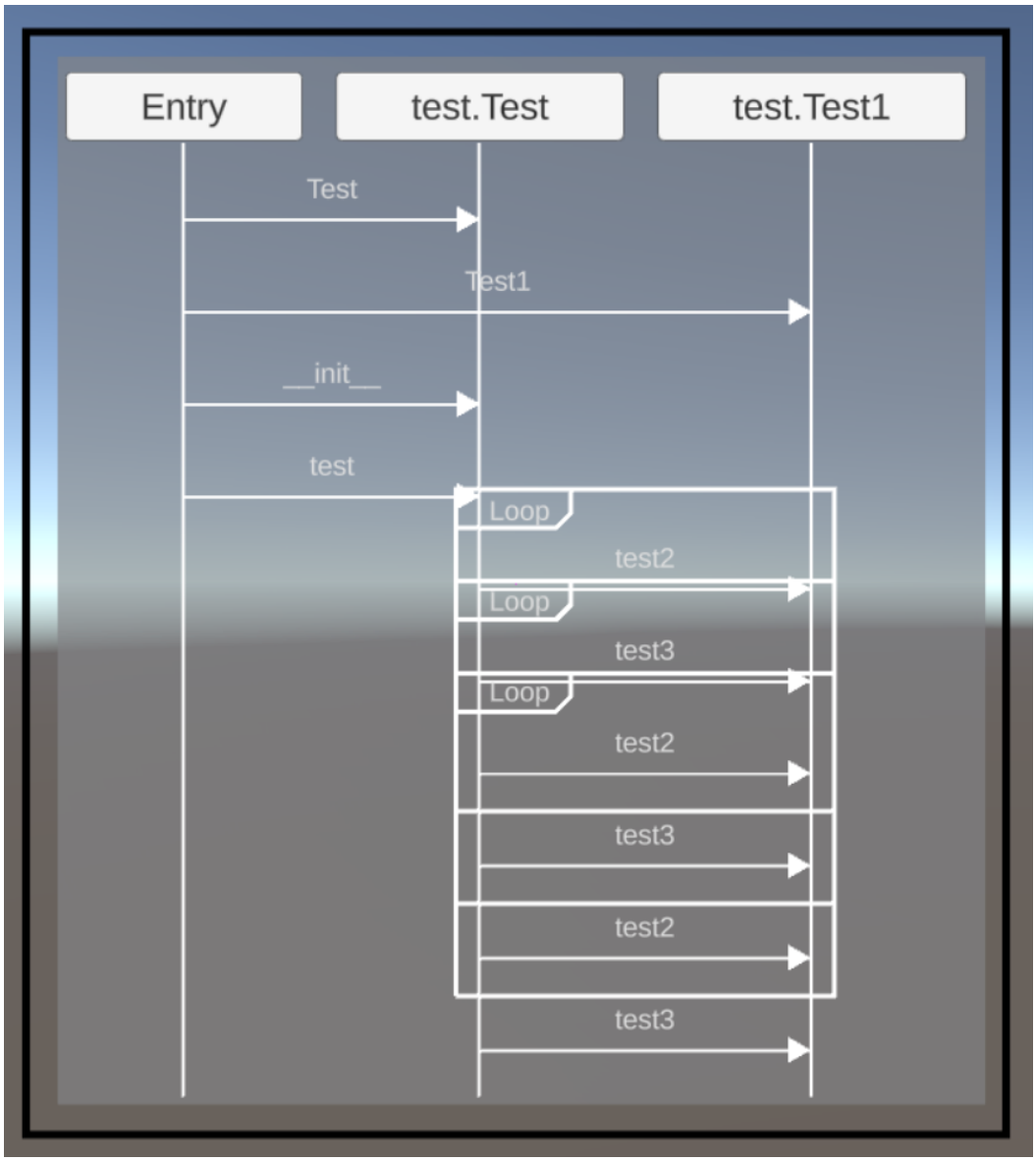


Obr. 5.4: Ukážka detekcie viacerých cyklov

5.6 Vizualizácia zložitejšieho diagramu

Posledným testom overíme schopnosť implementácie nášho prototypu zobrazovať zložitejší diagram. Na vizualizáciu budeme tentokrát používať nasledujúci zdrojový kód:

```
class Test(object):
    def __init__(self):
        for x in range(3):
            self.t = Test1()
            self.t2 = Test2()

    def test(self):
        for x in range(2):
```

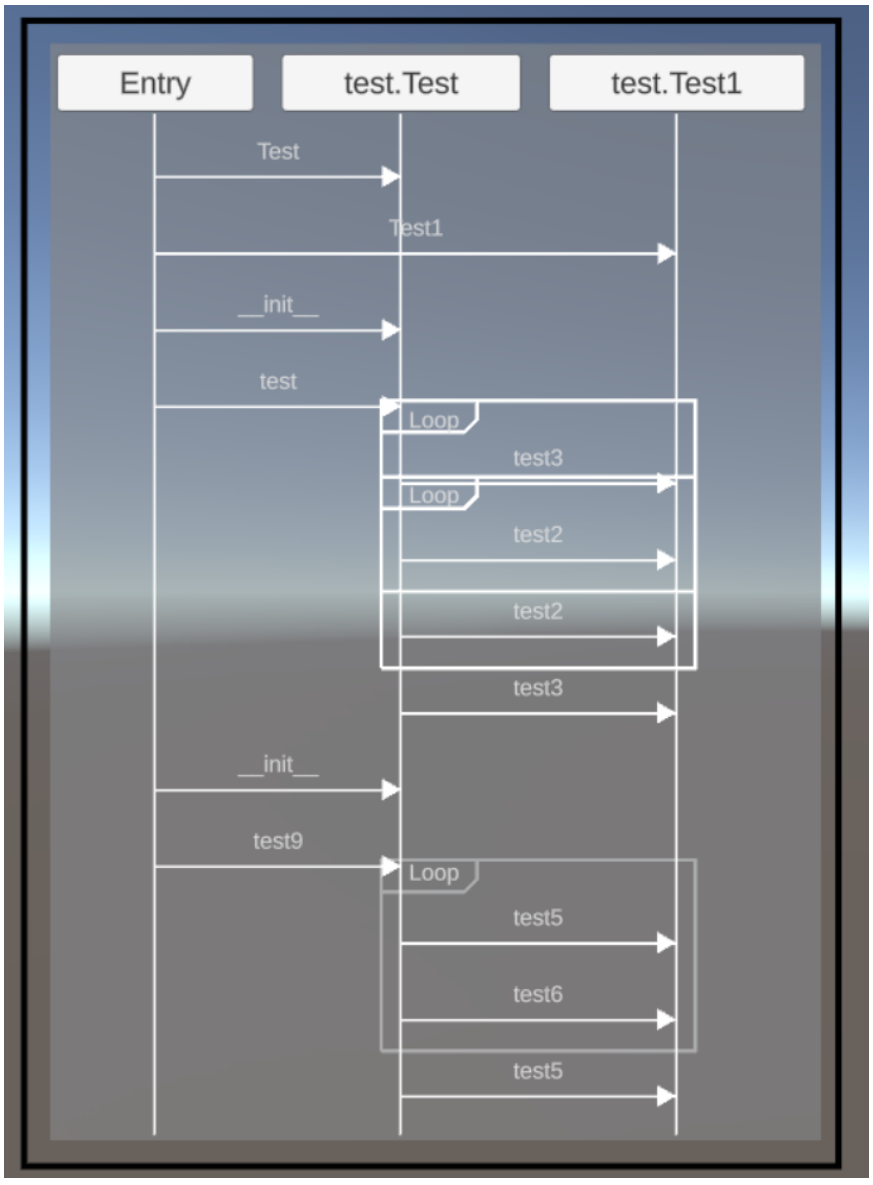


Obr. 5.5: Ukážka detekcie vnorených cyklov

```
self.t.test2()
self.t.test3()
for x in range(2):
    self.t.test3()
    self.t.test2()
self.t2.test5()
for x in range(2):
    self.t.test2()
    self.t2.test5()
self.t.test3()
self.t.test2()
```

```
        self.t.test3()

class Test1(object):
    def test2(self):
        print 3

    def test3(self):
        print 3

class Test2(object):
    def test5(self):
        print 5

if __name__=="__main__":
    Test().test()
```

Na obrázku 5.6 môžeme vidieť diagram zobrazujúci jednotlivé správy korešpondujúce s volaniami jednotlivých funkcií v našom zdrojovom kóde správne obalené fragmentami v prípade výskytu cyklu.



Obr. 5.6: Ukážka zložitejšieho algoritmu

Kapitola 6

Zhodnotenie

V tejto kapitole sa budeme venovať zhodnoteniu nami vytvoreného prototypu, splneniu či nesplneniu požiadaviek na prototyp a návrhom ďalších funkcií.

Cieľom práce bolo navrhnúť a implementovať ďalšie funkcie do prototypu vytvoreného Ing. Jurajom Vincúrom na vizualizáciu sekvenčného diagramu v trojrozmernom priestore.

Správne fungovanie a použitie finálneho prototypu sme testovali v kapitole 5. Výsledkom našej práce je fungujúci prototyp vizualizácie a editovania sekvenčného diagramu implementovaný podľa návrhu funkcií vychádzajúcich z analýzy problémovej oblasti a samotnej analýzy prototypu.

6.1 Splnené požiadavky

Počas implementácie sa nám podarilo splniť všetky požiadavky, ktoré sme si stanovili v návrhu. Z funkcií, ktoré sme chceli do prototypu implementovať, sa podarilo splniť nasledovné:

- Možnosť vizualizácie viacerých diagramov
- Odstránenie zobrazenia spätných správ
- Pridanie ohraničenia k diagramu
- Pridanie pozadia k diagramu
- Zobrazenie diagramov na viacerých vrstvách
- Detekovanie jednoduchého cyklu
- Detekovanie zložitejšieho cyklu
- Detekovanie vnorených cyklov
- Obalenie správ fragmentom

6.2 Nesplnené požiadavky

Zo spomenutých funkcií sa nám nepodarilo splniť:

- Pridanie aktivácií na čiary života

Keďže táto požiadavka sa zdala ako menej dôležitá a pomerne jednoduchá na implementáciu, nechávali sme to na úplný koniec. To však spôsobilo, že návrh prototypu sa už nedal jednoducho zmeniť, aby sme získavali pri vykresľovaní správ aj ďalšie informácie o predošlých správach a ich výskyte na čiarach života. Preto sa nám danú požiadavku nepodarilo splniť.

Okrem toho bolo v pláne umožnenie aj statickej analýzy zdrojového kódu. Návrh prototypu a jeho následné mapovanie jednotlivých prvkov diagramu sekvencií tento druh analýzy nepodporuje. Jednoduchšie ako zmena implementácie na statickú analýzu, by bolo vytvorenie úplne nového prototypu, na čo ale už v tak pokročilom čase vypracovávania práce nebol dostatok času.

6.3 Návrh možných vylepšení a ďalšej práce

Ako jedno z možných vylepšení navrhujeme napríklad vizualizáciu diagramov sekvencií jednotlivých metód v zdrojovom kóde. Prispeli by sme tak k ešte väčšej prehľadnosti a zjednodušeniu pochopenia programu. Ďalším vylepšením by mohla byť dodatočná implementácia požiadaviek nesplnených v tejto práci.

Na základe vyššie spomenutého sa možnosť aj statickej analýzy javí ako významné vylepšenie nášho riešenia, no jeho dosiahnutie používaný prototyp neumožňoval. Bolo by možné využiť štýl súčasnej vizualizácie diagramu v samostatnej implementácii statickej analýzy, čo by vytvorilo zaujímavé riešenie s veľkým potenciálom.

Kapitola 7

Záver

Zadaním tejto bakalárskej práce bolo vizualizovať vybraný UML diagram v trojrozmernom priestore. Ako konkrétny diagram, s ktorým sme pracovali, sme zvolili sekvenčný diagram, ktorý zachytáva interakciu objektov v čase. Práve postupnosť volaní jednotlivých funkcií zobrazená v diagrame sekvencií, môže uľahčiť programátorom prácu pri snahe pochopiť zdrojový kód či ladení programu, preto sme sa sústredili práve na vizualizáciu diagramu počas ladenia programu.

V rámci analýzy sme sa zaoberali dôležitosťou modelovania softvéru a možnými problémami, ktoré môžu nastať pri vizualizácii diagramov v dvojrozmernom priestore. Z toho dôvodu začali vznikať rôzne riešenia vizualizácie modelov vo viacrozmernom priestore.

Na základe analýzy problémovej oblasti sme zhodnotili, že vytvorenie prototypu na vizualizáciu diagramu v trojrozmernom priestore môže byť v súčasnosti prínosné. Používali sme už vytvorený prototyp vizualizácie jednoduchého sekvenčného diagramu, ktorého jednotlivé vlastnosti a funkčnosť sme analyzovali a na základe toho rozhodli o potrebe implementácie ďalšej funkcionality.

Ako prvé sme doimplementovali možnosť vizualizácie viacerých diagramov a následne ich rozmiestnenie na viacerých vrstvách v priestore kvôli väčšej prehľadnosti. Možnosť vizualizácie viacerých diagramov v priestore sme považovali za veľmi dôležitú, keďže programátori veľmi zriedka pracujú s jednoduchým súbormi a zobrazenie viacerých diagramov v priestore by pre nich bolo veľmi prínosné.

Následne sme sa snažili vylepšiť zobrazenie daného diagramu. Z dôvodu možnosti zobrazenia viacerých diagramov bolo nutné uvažovať o jeho ohraničení, ktoré by jednoznačne vytýčilo, ktoré prvky do diagramu patria a samotná vizualizácia by sa tak stala viac prehľadnou. Taktiež sme zvolili pridanie tmavšieho pozadia za každý diagram, aby sme zabezpečili ľahšiu čitateľnosť.

Implementácia týchto funkcionalít zabrala viac času ako sme prvotne predpokladali, najmä vďaka potrebe pochopenia fungovania daného prototypu a nutnosti prispôsobe-

nia návrhov už existujúcej implementácii. Najväčšie problémy však nastali pri implementácii detekovania cyklov a následnom editovaní správ v diagrame ich obalením do fragmentu. Dlhší čas trvalo vôbec navrhnúť nejaké možné riešenie a tak sme najprv skúsili detekovať len jednoduché cykly obsahujúce jedno volanie. Následne sme daný algoritmus rozšírili na detekovanie aj zložitejších cyklov. Tento princíp bol však veľmi výpočtovo náročný, vzhľadom na to koľko správ bolo potrebné pri každej interakcii porovnávať. Spomalenie programu bolo tak výrazné, až sa nedal plnohodnotne používať. Z toho dôvodu sme museli vymyslieť iné riešenie, ktorým bolo spojenie statickej a dynamickej analýzy. Toto riešenie prinieslo požadovaný výsledok bez akéhokoľvek výrazného zaťaženia programu a umožnilo detekovať aj vnorené cykly v programe.

Funkčnosť nášho prototypu sme testovali pomocou viacerých testov spomínaných v tejto práci. Ich náročnosť sa postupne stupňovala a pomocou nich sa nám úspešne podarilo zistené chyby v prototype opraviť. Výsledkom je tak plne funkčný prototyp podporujúci funkcie implementované v tejto práci. V našom riešení vidíme potenciál na pridávanie ďalších funkcií v budúcnosti. Zaujímavým riešením by bola tiež napríklad vizualizácia zdrojového kódu zo statickej analýzy, ktorá umožňuje o programe zistiť vlastnosti, ktoré nám z dynamickej analýzy nie sú známe. V takom prípade je ale potrebné vytvoriť novú implementáciu celého prototypu, pretože nami používaný prototyp statickú analýzu nepodporuje a jeho úprava by bola veľmi náročná.

Literatúra

[1] Tim Dwyer. Three dimensional uml using force directed layout. In *Proceedings of the 2001 Asia-Pacific Symposium on Information Visualisation - Volume 9*, APVis '01, pages 77–85, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[2] M. Ferenc, I. Polasek, and J. Vincur. Collaborative modeling and visualization of software systems using multidimensional uml. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 99–103, Sept 2017.

[3] Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a text generation template language for modelica. 09 2009.

[4] Joseph Gil and Stuart Kent. Three dimensional software modelling. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 105–114, Washington, DC, USA, 1998. IEEE Computer Society.

[5] Hassan Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

[6] Lukáš Gregorovič, Ivan Polasek, and Branislav Sobota. *Software Model Creation with Multidimensional UML*. Springer International Publishing, 2015.

[7] John Grundy and John Hosking. High-level static and dynamic visualisation of software architectures. In *In Proceedings of 2000 IEEE Symposium on Visual Languages*, pages 18–20. IEEE CS Press, 2000.

[8] Mária Bieliková Jakub Šimko, Marián Šimko. *Softvérové inžinierstvo v otázkach a odpovediach*. Slovenská technická univerzita v Bratislave, 2017.

[9] JSON manual. *Introducing JSON*. https://www.json.org/.

[10] Radoslav Kirkov and Gennady Agre. Source code analysisâ“an overview. 05 2019.

[11] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205, July 2000.

[12] Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. 07 1999.

[13] Xiaoshan Li, Zhiming Liu , and H Jifeng. A formal semantics of uml sequence diagram. volume 2004, pages 168– 177, 02 2004.

[14] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 27–ff, New York, NY, USA, 2003. ACM.

[15] Paul McIntosh, Margaret Hamilton, and Ron van Schyndel. X3d-uml: Enabling advanced uml visualisation through x3d. In *Proceedings of the Tenth International Conference on 3D Web Technology*, Web3D '05, pages 135–142, New York, NY, USA, 2005. ACM.

[16] Paul McIntosh, Margaret Hamilton, and Ron van Schyndel. X3d-uml: 3d uml state machine diagrams. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, pages 264–279, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[17] Microsoft. *Introduction to the C Language and the .NET Framework*, Júl 2015. https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

[18] Microsoft. *About Visual Studio Code*, August 2018. https://code.visualstudio.com/docs/editor/whyvscode.

[19] Zoltan Micskei and Hélène Waeselynck. Uml 2.0 sequence diagrams' semantics. 12 2018.

[20] Object Management Group. *UML 2.1.2 Superstructure Specification*, November 2007. http://www.omg.org/cgi-bin/doc?formal/07-11-02.

[21] Object Management Group. *OMG Unified Modeling Language*, Máj 2015. http://www.omg.org/spec/UML/2.5/.

[22] Moreau Pierre-Etienne, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In Görel Hedin, editor, *Compiler Construction*, pages 61–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] Python. *What is Python*. https://www.python.org/doc/essays/blurb/.

[24] Steven P. Reiss and Manos Renieris. Jove: Java as it happens. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 115–124, New York, NY, USA, 2005. ACM.

[25] James Rumbaugh. *Object-Oriented Modeling and Design with UML.* Technical Publication Pune, 1991.

[26] Unity. *The world's leading real-time engine*, 2018. https://unity3d.com/unity.

[27] Antinisca Di Marco Vittorio Cortellessa, Paola Inverardi. *Model-Based Software Performance Analysis.* Sringer, 2011.

[28] Jens von Pilgrim and Kristian Duske. Gef3d: A framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 95–104, New York, NY, USA, 2008. ACM.

Kapitola A

Technická dokumentácia

V technickej dokumentácii bližšie opíšeme použitý prototyp a jeho funkcie.

A.1 Použitý prototyp

V práci pracujeme s prototypom Ing. Juraja Vincúra na vizualizáciu sekvenčného diagramu. Všetky funkcie opísané nižšie boli implementované práve do daného prototypu.

A.2 Zobrazovanie diagramov na vrstvách

```
GameObject tmp = Instantiate( seqTest , diagramObj);
tmp.name = "#" + this.diagramCount; // set name of generated diagram


if (diagramCount == 0)
{
    tmp.transform.position = new Vector3(0, 0, 0);
    tmp.layer = 1;
    SetLayerRecursively(tmp, 1);
    prevLayer = tmp.layer;
}
else
{
    RectTransform rt = (RectTransform)prev.transform;
    float width = rt.rect.width;
    tmp.transform.position = prev.transform.position + new Vector3(width + 50, 0, 0);
    tmp.layer = LayerMask.NameToLayer((prevLayer + 1).ToString());
    SetLayerRecursively(tmp, tmp.layer);
}
```

Obr. A.1: Ukážka zdrojového kódu na umiestenenie diagramu na vrstvu

A.3 Umiestnenie komponentov na vrstvy

```
public static void SetLayerRecursively(GameObject go, int layerNumber)
{
    foreach (Transform trans in go.GetComponentsInChildren<Transform>(true))
    {
        trans.gameObject.layer = layerNumber;
    }
}
```

Obr. A.2: Rekurzívna funkcia na pridelenie vrstvy komponentom

A.4 Vytváranie ohraničenia diagramu

```
public void refreshBounds()
{
    if (this.obj != null) {

        int spacing = 25;

        float x = this.obj.GetComponent<RectTransform>().sizeDelta.x; // get diagram width
        float y = this.obj.GetComponent<RectTransform>().sizeDelta.y; // get diagram height

        float posX = this.obj.GetComponent<RectTransform>().position.x; // get diagram width
        float posY = this.obj.GetComponent<RectTransform>().position.y; // get diagram height

        //draw lines by calculatin their positions

        var line = this.obj.GetComponent<LineRenderer>();

        line.SetPosition(0, new Vector3(posX +  0            , posY +  y/2 + spacing, 0));
        line.SetPosition(1, new Vector3(posX + -x/2 - spacing, posY +  y/2 + spacing, 0));
        line.SetPosition(2, new Vector3(posX + -x/2 - spacing, posY + -y/2 - spacing, 0));
        line.SetPosition(3, new Vector3(posX +  x/2 + spacing, posY + -y/2 - spacing, 0));
        line.SetPosition(4, new Vector3(posX +  x/2 + spacing, posY +  y/2 + spacing, 0));
        line.SetPosition(5, new Vector3(posX +  0            , posY +  y /2 + spacing, 0));

        line.material = new Material(Shader.Find("Sprites/Default"));
        line.startColor = Color.black;
        line.endColor = Color.black;

        line.startWidth = 5;
        line.endWidth = 5;

    }
```

Obr. A.3: Vytváranie ohraničenia diagramu

A.5 Detekovanie cyklu v programe

```
public void AddLoop(IList tmpBounds,GameObject tmpObject)
{
    bool existingLoop = false;
    Loop tmpLoop = null;

    //subloop scaling resize;
    double subloop = 0;
    Transform subloopObject = null;

    if(loopsTransform.childCount > 0)
    {
        for(int i = 0; i < loopsTransform.childCount; i++)
        {
            Transform item = loopsTransform.GetChild(i);

            IList itemBounds = item.GetComponent<Loop>().getBounds();

            if(    (System.Convert.ToInt32(itemBounds[0].ToString()) == System.Convert.ToInt32(tmpBounds[0].ToString()))
                && (System.Convert.ToInt32(itemBounds[1].ToString()) == System.Convert.ToInt32(tmpBounds[1].ToString())))
            {
                existingLoop = true;

                tmpLoop = item.GetComponent<Loop>();
            }


            if (!existingLoop)
            {
                bool fromBottom = ( (System.Convert.ToInt32(itemBounds[0].ToString()) <= System.Convert.ToInt32(tmpBounds[0].ToString())));
                bool fromTop = ((System.Convert.ToInt32(itemBounds[1].ToString()) >= System.Convert.ToInt32(tmpBounds[1].ToString())));


                if (fromBottom && fromTop)
                {
                    subloop++;

                    subloopObject = item;
                }
            }


        }

    }
```

Obr. A.4: Funkcia na detekciu cyklu č.1

```
if (!existingLoop)
{
        var lop = Instantiate(loopPrefab, loopsTransform);
        int currLayer = (GameObject.Find("#" + (GameObject.Find("TCPService").GetComponent<TCPService>().getDiagCount() - 1))).layer;
        lop.layer = currLayer;


        loopsListAll.Add(lop);

        // get loop object script
        var lopScript = lop.GetComponent<Loop>();

        // init Object
        lopScript.setBounds(tmpBounds);
        lopScript.addLoopObject(tmpObject);

        for (int i = 0; i < subloop; i++)
        {
            lopScript.decreaseLevel();
        }
}
else
{
    tmpLoop.addLoopObject(tmpObject);
}

Debug.Log("Loop added");
```

Obr. A.5: Funkcia na detekciu cyklu č.2

```
void updateLoop()
{
    List<GameObject> loopObjectsAll = new List<GameObject>();

    loopObjectsAll.Clear();

    loopObjects.ForEach((item) =>
    {
        loopObjectsAll.Add(item);
    });


    if (transform.parent.parent.GetComponent<SequenceDiagram>().loopsListAll.Count > 0)
    {
        foreach(GameObject item in transform.parent.parent.GetComponent<SequenceDiagram>().loopsListAll)
        {
            IList itemBounds = item.GetComponent<Loop>().getBounds();

            if (    ((System.Convert.ToInt32(itemBounds[0].ToString()) != System.Convert.ToInt32(this.getBounds()[0].ToString())))
                &&   (System.Convert.ToInt32(itemBounds[1].ToString()) != System.Convert.ToInt32(this.getBounds()[1].ToString())))

                &&
                   ((System.Convert.ToInt32(itemBounds[0].ToString()) >= System.Convert.ToInt32(this.getBounds()[0].ToString())
                &&  (System.Convert.ToInt32(itemBounds[1].ToString()) <= System.Convert.ToInt32(this.getBounds()[1].ToString()))))))
            {

                foreach( GameObject subItem in item.GetComponent<Loop>().loopObjects)
                {
                    loopObjectsAll.Add(subItem);
                }
            }
        }
    }
```

Obr. A.6: Prispôsobenie fragmentu

Kapitola B

Používateľská príručka

V tejto časti sa budeme venovať bližšiemu opisu spustenia a následnej práce s naším prototypom.

B.1 Softvérové požiadavky

- Visual Studio Code, verzia 1.24.1
- Python, verzia 2.7.15
- Unity
- node.js, aktuálna verzia

B.2 Prvé spustenie

Pred prvým spustením je nutné mať nainštalovaný potrebný softvér a vykonať nasledovné kroky:

- V priečinku VSCodeDebug otvoriť príkazový riadok a spustiť príkaz npm install. Tento príkaz nainštaluje všetky potrebné závislosti (balíky) na fungovanie programu.
- V priečinku C:\Users\"meno používateľa"\.vscode\extensions\ ms-python.python-2018.3.1\out\client\debugger\otvoriť súbor Main.js a pridať doň daný kód:

```
sendEvent(event) {
    super.sendEvent(event);
    event.event = "custom";
    super.sendEvent(event);
}
```

B.3 Spustenie a inicializácia prototypu

Pre spustenie vizualizácie sekvenčného diagramu z dynamickej analýzy je potrebné vykonať nasledujúce kroky:

1. Spustiť program DiagramServer.py nachádzajúci sa v priečinku Diapy.
2. Otvoriť príslušný projekt v Unity a spustiť aktuálnu scénu.
3. Vo Visual Studio Code otvoriť priečinok DebugExtension a spustiť ladenie programu stlačením klávesy F5.
4. Po vykonaní predošlého kroku sa otvorí okno daného rozšírenia, kde je potrebné otvoriť akýkoľvek Python projekt dostupný v priečinku Testovanie, alebo vlastný súbor.
5. Po otvorení súboru použiť klávesovú skratku Ctrl+P a do otvoreného vyhľadávacieho poľa napísať >Init Sequence Diagram
6. V tomto momente je už sekvenčný diagram inicializovaný a pripravený na vizualizáciu zdrojového kódu.

Používateľ môže otvoriť v danom rozšírení v programe Visual Studio Code vlastný projekt napísaný v jazyku Python alebo využiť jeden z testovacích projektov, ktoré sú odovzdané s touto prácou. Po tomto kroku môže pridať na akékoľvek riadky body zastavenia (breakpointy), na ktorých sa program pri ladení zastaví. V danom okamihu prebehne komunikácia s Unity a do vizualizovaného diagramu je pridaná konkrétna správa. Krokovanie programu prebieha jednoduchým stlačením klávesy F5. Vizualizáciu je možné sledovať počas celého procesu ladenia v Unity.

Kapitola C

Plán práce

C.1 1. semester

- 1. týždeň – oboznámenie sa s témou, vyhľadánie materiálov
- 2. – 3. týždeň - oboznámenie sa s prototypom a jeho analýza
- 4. - 6. týždeň – študium zdrojov, tutoriály programovania v Unity
- 7. – 9. týždeň – implementácia zobrazovania diagramov na viacerých vrstvách, analýza danej oblasti
- 10. – 12. týždeň – návrh implementácie detekovania cyklov v zdrojovom kóde, písanie analýzy

C.1.1 Plnenie plánu v zimnom semestri

Pri práci na bakalárskom projekte v zimnom semestri sa nám prvú polovicu semestra darilo dodržiavať stanovené termíny, no v druhej polovici už plnenie plánu bolo pomalšie ako naše očakávania. Prispel k tomu najmä nedostatok času a náročnejšia implementačná časť. Spomínané meškanie v plnení plánu v prvom semestri zmiernime intenzívnejšou prácou na implementácii daných riešení do prototypu v druhom semestri.

C.2 2. semester

- 1. týždeň – implementácia detekovania jednoduchých cyklov v zdrojovom kóde z dynamickej analýzy
- 2. – 4. úprava implemenemtácie na zložitejšie cykly a ich obaľovanie fragmentom

- 5. – 8. implementácia grafických úprav diagramu, zmena implementácie detekovania cyklov v programe, obaľovania fragmentom a vizualizácie vnorených fragmentov
- 9. týždeň – dokončenie implementácie, refaktoring
- 10. – 12. týždeň – testovanie implementácie, dokončenie práce

C.2.1 Plnenie plánu v letnom semestri

V tomto semestri sa nám podarilo plán práce plniť úspešnejšie, aj napriek tomu, že implementácia detekovania cyklov bola značne náročnejšia ako sme očakávali. Implementáciu sme museli niekoľkorát zmeniť, až kým sme nedosiahli požadovaný výsledok. Ostatné implementované funkcie boli menej náročné, čo mám pomohlo udržať tempo v stanovenom pláne.

Kapitola D

Opis digitálnej časti práce

Evidenčné číslo: FIIT-5212-85962
Do Akademického informačného systému budú odovzdané nasledovné súbory:

- priečinok so zdrojovými súbormi programu Unity
- priečinok Diapy so zdrojovým kódom aplikácie slúžiacej na parsovanie zdrojového kódu
- priečinok DebugExtension so zdrojovými súbormi rozšírenia (pluginu) pre ladenie programu vo Visual Studio Code
- testovacie zdrojové kódy v jazyku Python
- elektornickú verziu tejto práce
- pôvodnú verziu používaného prototypu v priečinku PoužitýPrototyp