Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Editovanie vrstiev tried a objektov v modeloch xUML

Bakalárska práca

2023
Dávid Laurovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Editovanie vrstiev tried a objektov v modeloch xUML

Bakalárska práca

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

Bratislava, 2023
Dávid Laurovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Dávid Laurovič
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Editovanie vrstiev tried a objektov v modeloch xUML
Editing class and object layers in xUML models

Anotácia: Pre pochopenie rozsiahlych softvérových systémov je užitočné poznať nielen ich štruktúru ale aj dynamickú povahu, scenáre funkcionalít a prípadov použitia, ako aj interakcie medzi prvkami softvérovej architektúry.
Analyzujte vybraté metódy modelovania v softvérovom inžinierstve (napríklad Executable UML a Object Action Language), interaktívnu grafiky v Unity a náš prototyp animácie modelu v xUML z roku 2021/2022.
Platforma Unity umožňuje prácu v 2D a 3D priestore ale aj migrovať do virtuálnej (VR) alebo rozšírenej reality (AR).

Cieľ: Cieľom práce je obohatiť existujúci prototyp modelovania softvérovej architektúry a jej funkcionality.
Navrhnite metódu jednoduchého editovania prepojených vrstiev tried a objektov v našom novom prototype fúzie štruktúry a dynamiky xUML modelu.
Bakalárska práca bude súčasťou rozbiehaného výskumu podpory kolaboratívneho modelovania a vizualizácie vo VR/AR priestore.

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.

Gregorovic L., Polasek I. Analysis and design of object-oriented software using multidimensional UML. In Proceedings of the 15th international conference on knowledge technologies and data-driven business, i-KNOW 2015, 21 - 22 October 2015, Graz, Austria. ACM New York, 2015. ISBN 978-1-4503-3721-2.

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.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátum zadania: 31.05.2022

Dátum schválenia: 19.09.2022

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestné prehlásenie: Čestne prehlasujem, že som túto prácu vypracoval samostatne iba s použitím uvedených zdrojov a na základe konzultácií so školiteľom.

. . . . . .

Poďakovanie: V prvom rade chcem poďakovať môjmu školiteľovi Ivanovi Poláškovi, za jeho cenné rady, konzultácie a poznámky.

Ďalej by som chcel poďakovať Lukášovi Radoskému a Olge Charnej za uvedenie do problematiky, zoznámenie s prototypom AnimArch po funkčnej aj implementačnej časti ako aj za pripomienky k samotnému editoru.

V neposlednej rade by som chcel poďakovať Martinovi Vankovi za revízie kódu, trpezlivosť pri prepájaní vyvíjaných funkcionalít ako aj všeobecné praktické rady k rôznym oblastiam práce na projekte.

Taktiež by som chcel poďakovať mojej rodine a kamarátom za ich podporu aj v tých najhorších časoch.

. . . . . .

Abstrakt

Návrh a dokumentácia architektúr softvérových systémov vie viesť ku komplexným štruktúram, ktoré nemusia byť ľahko pochopiteľné. Jedným zo spôsobov ako popisovať softvérové systémy je pomocou UML diagramov. Triedne a objektové diagramy sú však z pravidla štrukturálne, statické a chýba im dynamická povaha, čo môže viesť k nižšej zrozumiteľnosti. Cieľom našej práce je rozšíriť existujúci nástroj AnimArch, ktorý sa pomocou animácií snaží priniesť dynamiku do týchto diagramov. Toto prostredie umožňuje okrem animovania týchto diagramov vo viacerých vrstvách, aj kolaboratívne softvérové modelovanie a generovanie zdrojového kódu priamo z modelu. Východiskom tejto práce je obohatenie tohto softvéru o vytváranie, editovanie a mazanie všetkých komponentov triednych diagramov a upravenie objektového diagramu takým spôsobom, ktorý zachová spätnú kompatibilitu so všetkými ostatnými funkcionalitami tohto prostredia a zároveň bude slúžiť ako robustný základ pre budúci vývoj nad týmto nástrojom. V rámci práce rozoberieme základné pojmy nevyhnutné na porozumenie práce v tomto prostredí. Ďalej rozoberieme jeho súčasnú architektúru, nami navrhnuté zmeny pamäťového modelu, ako aj parsovania a ukladania diagramov vytvorených v AnimArchu a vylepšenie samotného používateľského rozhrania.

Kľúčové slová: Vývoj riadený modelom, Editovanie UML, xUML, OAL, AnimArch

Abstract

The design and documentation of software system architectures can lead to complex structures that may not be easily understood. One way of describing software systems is by using UML diagrams. However, class and object diagrams are by rule structural, static and lack a dynamic nature, which can lead to lower understandability. The goal of our work is to extend an existing tool, AnimArch, which attempts to bring dynamic to these diagrams by using animations. In addition to animating these diagrams in multiple layers, this environment allows for collaborative software modeling and the generation of source code directly from the model. The basis of this work is to enrich this software by creating, editing and deleting all the components of the class diagrams and modifying the object diagram layer in a way that maintains backward compatibility with all the other functionalities of this environment, while serving as a robust foundation for future development on top of this work. As part of this thesis, we will discuss the basic concepts necessary to understand how to work in this environment. We will also discuss its current architecture, our proposed changes to the memory model as well as the parsing and storage of diagrams created in AnimArch, and improvements to the user interface itself.

Keywords: Model-driven development, UML editor, Executable UML, OAL, AnimArch

Obsah

Zoznam obrázkov

Úvod

Pri práci na veľkých softvérových projektoch vie veľmi ľahko dojsť k neporozumeniu, v akom vzťahu sú jednotlivé komponenty systému, ako spolu komunikujú a aké majú závislosti. Na zjednodušenie a zobrazenie architektúry sa preto používa vizualizácia. Jednou zo základných typov vizualizácií, s ktorou sa stretne každý programátor je formou UML diagramu. Existuje veľké množstvo typov UML diagramov, ale štandardné 2D modely spája jedna spoločná vlastnosť - sú statické. Na obohatenie štandardných diagramov o dynamiku sa preto využívajú tzv. xUML (executable UML) diagramy. Rozdiel oproti štandardným UML diagramom je zrejmý už z názvu, ide o spustiteľné diagramy. Vďaka ich dynamickej náture je preto jednouchšie modelovať komunikáciu medzi komponentami a samotné volania a závislosti medzi nimi sú jasnejšie a prehľadnejšie.

Táto bakalárska práca je súčasťou rozsiahlejšieho výskumu prebiehajúceho na projekte s názvom AnimArch. V súčastnosti ide o prototyp na báze spomínaných xUML diagramov vo viacerých vrstvách naraz. Ide o rozsiahly projekt, ktorý okrem vizualizácie umožňuje aj generovanie programového kódu priamo z modelu, ako aj kolaboratívne softvérové modelovanie pomocou sieťového prepojenia.

V súčasnej verzii tohto projektu však nie je možné štruktúru triednych diagramov vytvárať ani editovať. Cieľom tejto bakalárskej práce je preto navrhnúť vhodný spôsob ako pridať spomínané funkcionality, lepšie odabstrahovať a rozpliesť vysokú mieru prepojenia medzi triednou a objektovou vrstvou a implementovať tieto zmeny tak, aby nenarušili ostatné funkcionality. Naša práca na prototype musí byť taktiež dostatočne modálna pre potreby budúceho výskumu na projekte.

V tejto práci preto postupne predstavíme problematiku vývoja riadeného modelom, opíšeme UML a xUML diagramy, ich rozdiely, výhody a súčasný stav prototypu AnimArch. Ďalej budeme prezentovať návrh zmien týkajúcich sa architektúry, pamäťového modelu, používateľského rozhrania ako aj štruktúry ukladania diagramov do súborov. V poslednej kapitole podrobne opíšeme implementáciu okien na úpravu diagramov, proces ukladania diagramov do rôznych typov súborov, využitie návrhového vzoru State pri pridávaní hrán ako aj priebeh hlbokého editovania. Na záver zhodnotíme novo pridanú funkcionalitu, rozoberieme nedostatky prototypu a načrtneme budúcu prácu, ktorá bude prebiehať na projekte.

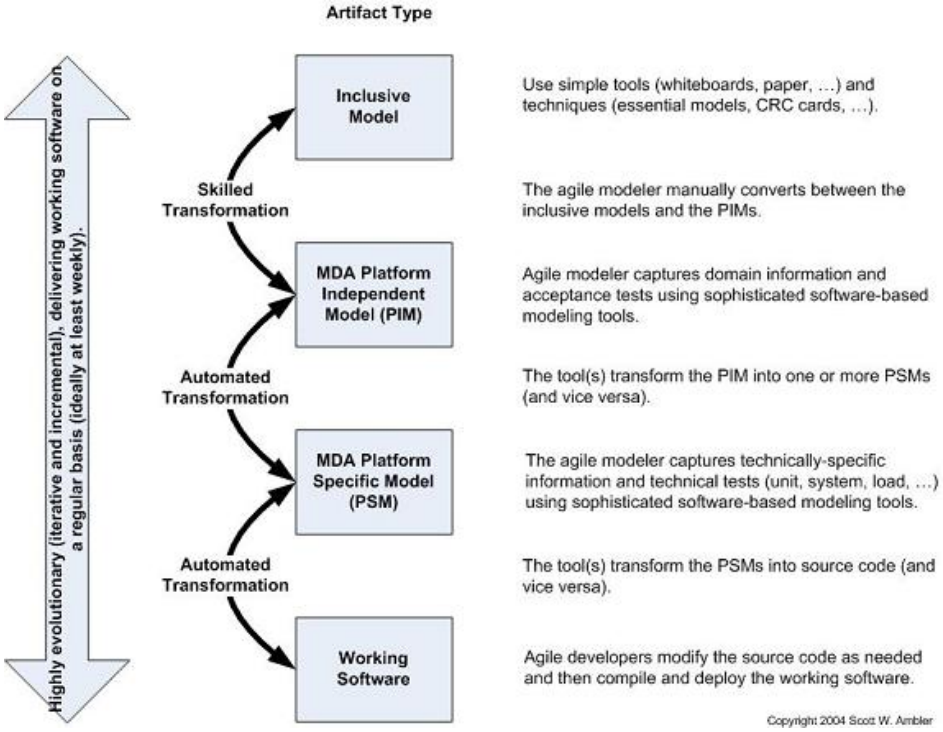
Kapitola 1

Úvod do problematiky

V tejto kapitole sa budeme venovať základným pojmom, ktoré sú významné pre našu prácu. V prvom rade vysvetlíme teoretické východiská nevyhnutné na pochopenie nášho výskumu a ďalej si rozoberieme súčasný stav prototypu AnimArch.

1.1 Vývoj riadený modelom

S navrhovaním softvéru je priamo prepojené modelovanie. Či už ide o dátový model, alebo architektúru, modely sa v dokumentácií vyskytujú všade, lebo sú častokrát jednoduchšie a pochopiteľnejšie, pretože prinášajú level abstrakcie samotnému kódu. Častokrát bývajú však modely zanedbávané a nie vždy reflektujú finálny stav programu, pretože ich neustála úprava stojí čas a peniaze.



Obr. 1.1: Vývoj pomocou MDD, prevzaté z [5]

Do tejto problematiky priamo vstupuje vývoj riadený modelom, alebo po anglicky Model-driven development (MDD). Jeho hlavnou myšlienkou je, že výsledným produktom nie je zdrojový kód ale model, z ktorého bude tento kód priamo vygenerovaný a naopak, ako je vidieť aj na ilustračnom obr. 1.1. Modely vedia byť menej viazané na konkrétnu implementáciu, a jednoduchšie na udržiavanie, vďaka čomu vedia byť viac odolné do budúcnosti.

Pre úspech MDD sú dôležité dva kľúčové koncepty. Prvým konceptom je kompletná automatizácia generovania zdrojového kódu z modelov a naspäť zo zdrojového kódu do modelov. Dosiahnutím kompletnej automatizácie by vývojár musel do výsledného vygenerovaného kódu zasahovať iba minimálne. Aj keď vznikali prvé kompilátory, bol z počiatku odpor od programátorov, že výsledný kód bude menej efektívny než strojový kód písaný ručne. V súčasnosti však bežný kompilátor vie rutinne prekonať väčšinu programátorov. Podobný vývoj si pri dnešnej sile umelej inteligencie v kombinácií s bežnými generátormi vieme tiež predstaviť aj v generátoroch kódu.

Ďalším kľúčovým konceptom je štandardizácia. V súčasnosti existuje veľa implementačných štandardov, napríklad SOAP, REST, ako aj modelovacích štandardov, napríklad UML, čo by malo v teoretickej rovine uľahčiť vývoj riadený modelom. Avšak nie všetky tieto štandardy sú dokonalé. Napríklad enormná veľkosť štandardu UML 2.0 s jeho rôznymi výnimkami nie je v čistej forme najvhodnejšia na úspešné prevedenie MDD a preto je priestor skúšať tento štandard obohatiť. [13, 9]

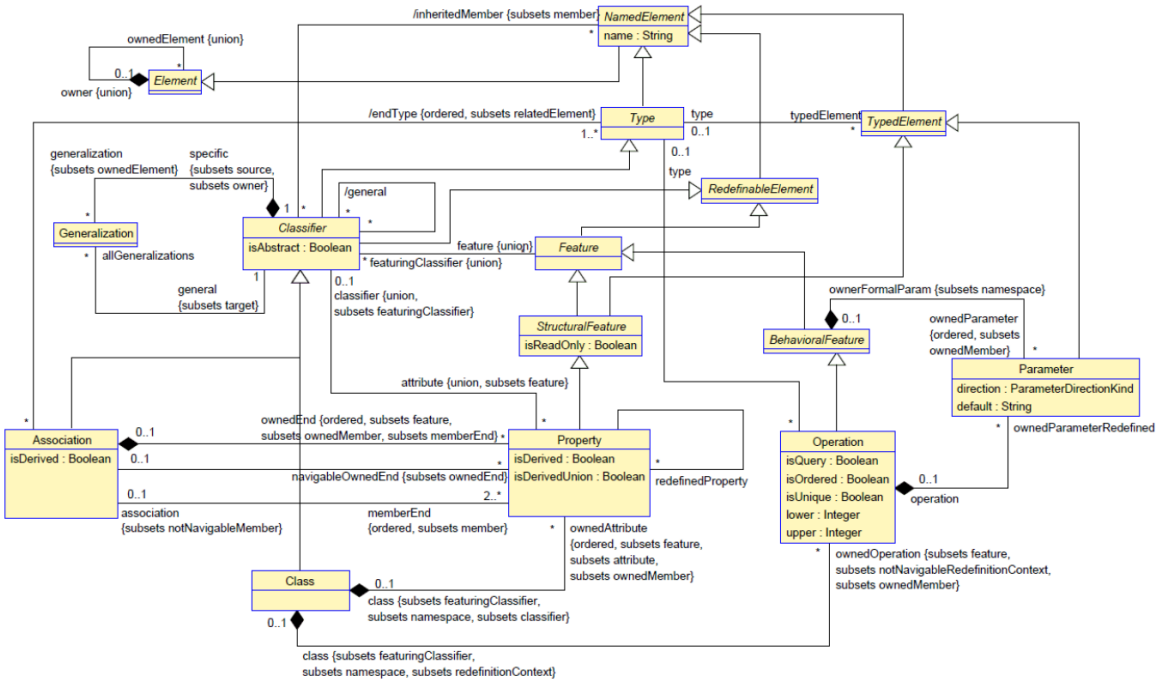
Pojem vývoja riadeného modelom úzko súvisí aj s prototypom AnimArch, na ktorého vývoji sme sa podielali. V rámci prototypu sa snažíme implementovať a rozšíriť jeden z existujúcich štandardov, xUML. V AnimArchu je možné k triednemu diagramu vytvoriť animáciu podľa jednoduchého skriptovacieho kódu OAL a z kombinácie OAL a triedneho diagramu môže používateľ jednoducho vygenerovať zdrojové súbory v jazyku Python.

1.2 UML metamodel

Vzhľadom na to, že pri našej práci navrhujeme a zobrazujeme UML diagramy, vzniká potreba mať konkrétnu vnútornú reprezentáciu rôznych prvkov diagramu. Na otázku ako reprezentovať diagramy sa dostávame k UML metamodelu, tzv. modelu o modeli (meta z gréc. za/po). UML metamodel je navrhnutý a špecifikovaný organizáciou Object Management Group a veľa rôznych nástrojov na prácu s UML taktiež využíva tento metamodel, vďaka čomu vieme zabezpečiť ľahšiu integráciu s inými prostrediami v budúcnosti. V rámci AnimArchu používame metamodel ako vzor pri OAL reprezentácií diagramu.

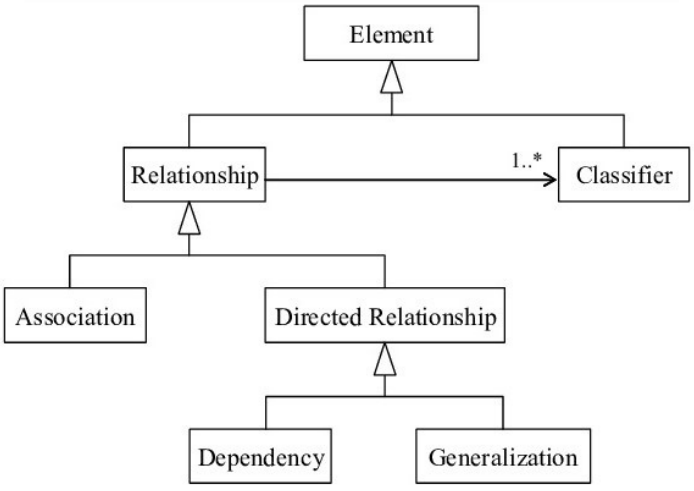
Keďže rozsiahlosť štandardu UML je obrovská, jeho kompletný metamodel je veľmi komplexný, preto pre potreby tejto práce opíšeme iba zjednodušenú verziu časti relevantnej pre našu prácu, a to konkrétne reprezentáciu tried a vzťahov popísanej diagramami obr. 1.2 a obr. 1.3

Základným prvkom, ktorý tvorí UML diagram je element, od ktorého priamo dedí NamedElement, teda pomenovaný element. Od NamedElement dedí Type, TypedElement a Classifier. Type predstavuje typ a TypedElement predstavuje element, ktorému bol konkrétny



Obr. 1.2: Zjednodušený UML metamodel, prevzaté z [10]

typ pridelený. Classifier je abstraktná trieda, ktorá zastrešuje množinu inštancií, ktoré majú spoločné vlastnosti, napríklad vzťah, trieda alebo aj interface. Vlastnosť je reprezentovaná triedou Feature a tie sa delia na štrukturálne a behaviorálne vlastnosti, reprezentované StructuralFeature a BehavioralFeature, ktoré dedia od Feature. Pod štrukturálne vlastnosti patrí Property, ktoré priamo reprezentujú atribúty triedy. Property taktiež špecifikujú asociácie medzi triedami. Pod behaviorálne vlastnosti patria Operation, ktoré priamo reprezentujú metódy tried. Operation môže ďalej obsahovať parametre reprezentované triedou Parameter. Triedy, reprezentované triedou Class, potom agregujú Property a Operation.



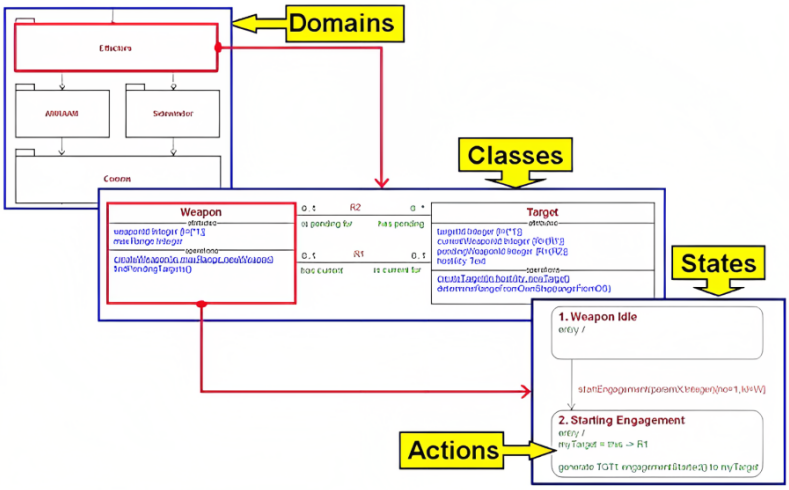
Obr. 1.3: Dedenie vzťahov v UML metamodeli

Vzťahy sú, rovnako ako ostatné prvky diagramu, taktiež elementami. Relationship, ktorá

reprezentuje abstraktný vzťah je napojená na Classifier a nie priamo na triedu. Rozlišujeme dva typy vzťahov, neorientované a orientované. Pod neorientované patrí asociácia, ktorá dedí priamo od Relationship. Asociácie sú ďalej špecifikované prepojením na Property ako je znázornené na obr. 1.3. Orientované vzťahy, teda také, ktoré majú zdroj a cieľ, sú reprezentované abstraktnou triedou DirectedRelationship a dedia od nej Dependency a Generalization. Reprezentácia agregácií nie je znázornená ani na jednom z diagramov a je riešená ako atribút triedy Property. [4, 6, 8]

1.3 Spustiteľné UML - xUML

Spustiteľné UML, alebo po anglicky executable UML (xUML), je rozšírením štandardu UML o schopnosť ich spúšťať, ako je už z názvu zrejmé, respektíve modelovať ich chod v prevádzke. V porovnaní s klasickými UML diagramami prinášajú level abstrakcie, ktorý je bližšie k samotnej implementácii pomocou programovacích jazykov zatiaľ čo neuberajú možnosť vidieť ako spolu jednotlivé komponenty komunikujú za behu. Samotné xUML modely sú nezávislé od výslednej platformy pre akú bude softvér vyvíjaný, avšak umožňujú podstatne jednoduchšiu konverziu do programovacieho jazyka, keďže im nechýba dynamika.



Obr. 1.4: Princípy xUML, prevzaté z [1]

Spustiteľné UML pracuje so štyrmi základnými konceptmi. Prvým konceptom je rozoznanie domén v rámci softvéru, ktorý vyvíjame. To v praxi znamená nemodelovať celý softvér naraz pomocou jedného obrovského modelu, ale rozdeliť ho na menšie celky a tieto domény následne samostatne modelovať pomocou xUML. Rozdelenie softvéru na domény zaručuje ich väčšiu modularitu pri vývoji a vďaka tomu vedia byť vytvorené s väčšou mierou nezávislosti. Komunikácia medzi doménami je zaručená pomocou takzvaných mostov, ang. bridges. Vďaka tomu vedia domény klásť požiadavky na ostatné domény na základe dohodnutých pravidiel.

Ďalším konceptom využívaným v xUML sú štandardné UML triedne a stavové, resp. sekvenčné diagramy. Entity v rámci domény predstavujú špecifické triedy s atribútmi a me-

tódami, ktoré sú spolu prepojené reláciami. Komunikácia medzi týmito triedami je zachytená v sekvenčnom alebo stavom diagrame.

Posledným dôležitým konceptom v xUML je takzvaný jazyk akcií (Action Language). Všeobecne jazyk akcií predstavuje jednoduchý skriptovací jazyk, pomocou ktorého vieme oživiť a znázorniť komunikáciu uvedenú v stavovom, resp. sekvenčnom diagrame do triedneho diagramu. Pomocou tohto jazyka vieme vytvárať inštancie, vykonávať operácie alebo upravovať atribúty. Práve zapracovanie skriptovacieho jazyka je to, čo prináša spustiteľnosť do xUML a vďaka čomu je možná priamočiara konverzia týchto modelov do konkrétnych programovacích jazykov. [12] Viac o súčasnom stave a realizácií xUML je na [7].

AnimArch je prototyp, ktorý realizuje modelovanie na báze xUML. V súčasnosti obsahuje zobrazovanie triednych diagramov a ako jazyk akcií využíva existujúci jazyk OAL upravený pre naše požiadavky. Neobsahuje možnosť zobrazovať alebo modelovať stavové/sekvenčné diagramy, ktoré by prislúchali k triednym diagramom, avšak v budúcnosti je plánované pridať aj túto funkcionalitu.

1.4 Súčasný stav prototypu AnimArch

Cieľom našej práce je rozšíriť prototyp AnimArch o novú funkcionalitu vytvárania a editovania diagramov. Na lepšie pochopenie kontextu našej práce je preto nevyhnutné rozobrať aj súčasný stav tohto prototypu, jeho rôzne funkcie, ktoré ponúka, a jeho architektúru.

AnimArch je obsiahly nástroj vyvíjaný v prostredí Unity pomocou jazyka C#. V rámci tohto prototypu sa zaoberáme tromi hlavnými oblasťami výskumu a to vizualizácia a animácia UML diagramov, generovanie zdrojového kódu z kombinácie UML diagramu a OAL skriptu a kolaboratívnym softvérovým modelovaním pomocou sieťového prepojenia. Postupne si rozoberieme všetky tri oblasti.

1.4.1 Funkcie

Vizualizácia a animácia UML diagramov

Hlavnou funkcionalitou prototypu AnimArch je vizualizácia a animácia UML diagramov. Pod pojmom animácia si v našom konktexte predstavujeme postupné zvýrazňovanie dotknutých tried, metód a vzťahov medzi nimi na základe definovaného OAL skriptu. Animácia prebieha zároveň v dvoch vrstvách, triednej a objektovej a počas jej priebehu je preto možné sledovať ako volania prebiehajú v obidvoch vrstvách, ako aj inicializovanie objektov. V tomto prostredí vieme vytvárať animácie pomocou používateľského rozhrania, upravovať už existujúce animácie a taktiež je zabezpečené ich ukladanie do súborov ako aj ich spätné načítavanie.

Vizualizálne programovanie

Prostredie AnimArch sa taktiež venuje oblasti vizuálneho programovania. Cieľom vizuálneho programovania je vytvárať softvér pomocou diagramov, grafických a ilustračných elementov. Kombináciou triedneho diagramu a OAL skriptu pre animáciu vieme relatívne jednoducho

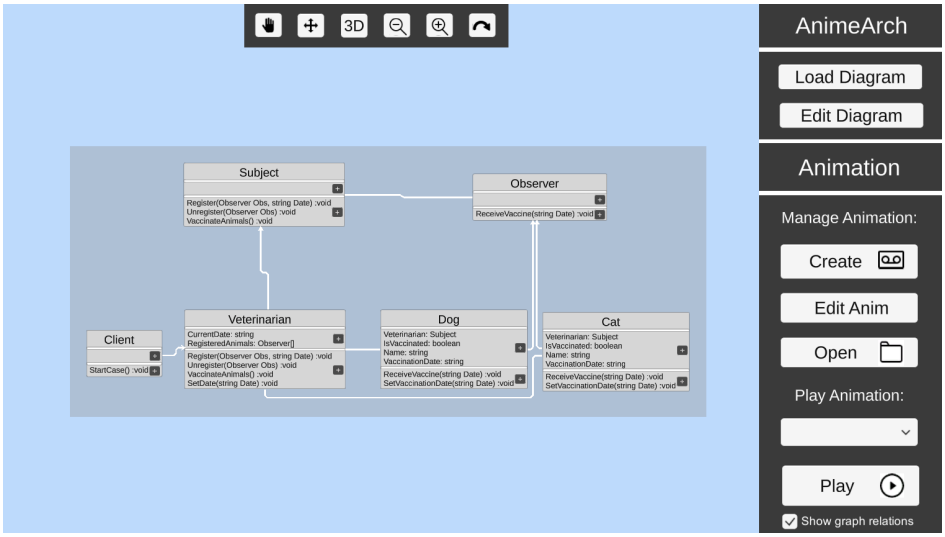
vygenerovať samotné zdrojové súbory v jazyku Python, ktoré reprezentujú akúkoľvek takto popísanú štruktúru tried. Na generovanie samotného zdrojového kódu sa využíva popísaná vlastná gramatika jazyka OAL, z ktorej sa následne vygeneroval Lexer a Parser pomocou nástroja ANTLR.

Kolaboratívne softvérové modelovanie

Najmladšou súčasťou nástroja AnimArch je sieťové prepojenie viacerých spustených inštancií AnimArchu naraz a vývoj v tejto oblasti ešte nebol dokončený v čase písania bakalárskej práce. Na sieťové prepojenie sa využíva architektúra takzvaného autoritatívneho servera. To znamená, že jeden server zdieľa všetkým klientom spoločný diagram, a tí môžu vidieť zmeny vykonávané na tomto diagrame v reálnom čase. Tento nový prístup umožní používateľom využívať AnimArch nielen ako nástroj na návrh, ale aj napríklad ako platformu na online vzdelávanie v oblasti návrhových vzorov. Nami navrhnutý spôsob editovania diagramov bude musieť byť dostatočne robustný, aby bolo jeho zakomponovanie do sieťového prepojenia čo najhladšie.

1.4.2 Používateľské rozhranie

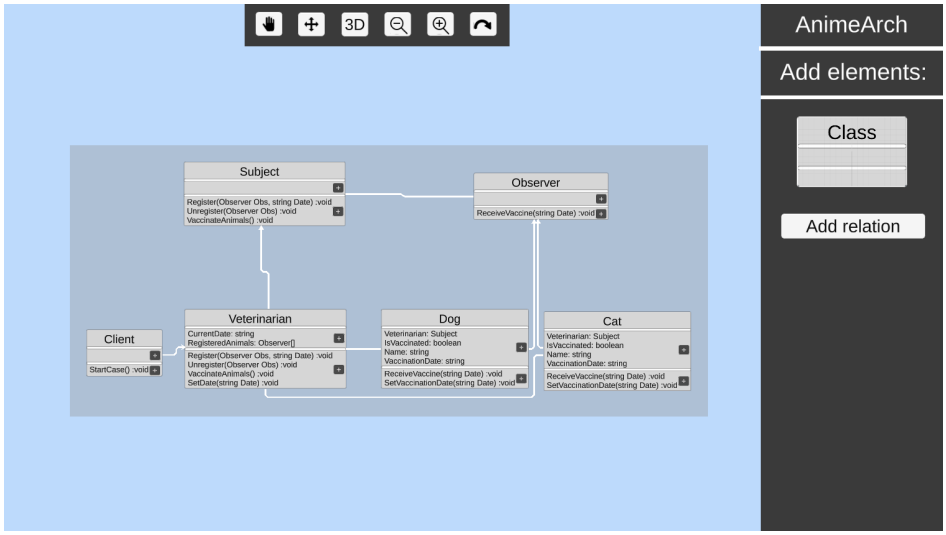
Po spustení aplikácie AnimArch sa dostaneme na hlavnú obrazovku, zobrazenú na obr. 1.5. Hlavná obrazovka obsahuje horný panel nástrojov na prácu s diagramom a bočný panel na načítanie diagramu, vytvorenie, načítanie a editovanie skriptu pre animáciu a taktiež tlačítko na spustenie zvolenej animácie.



Obr. 1.5: Pôvodná hlavná obrazovka

Na načítanie triedneho diagramu potrebujeme najprv diagram vytvoriť pomocou prostredia *Enterprise Architect*. Ide o komplexnú aplikáciu, v rámci ktorej je možné vytvárať veľké množstvo UML diagramov. Po exportovaní diagramu z tohto softvéru vo formáte *xmi* je možné ho importovať do AnimArchu pomocou tlačítka *Load Diagram* na hlavnej obrazovke prostredia. Ten načíta a zobrazí triedny diagram.

Po načítaní triedneho diagramu zo súbora vieme pomocou horného panela nástrojov hýbať celým diagramom, presúvať triedy v diagrame, pri čom sa automaticky prekresľujú aj relácie. Ďalej je pomocou tohto panela možné prepnutie do 3D režimu ako aj priblíženie a oddialenie samotného diagramu.



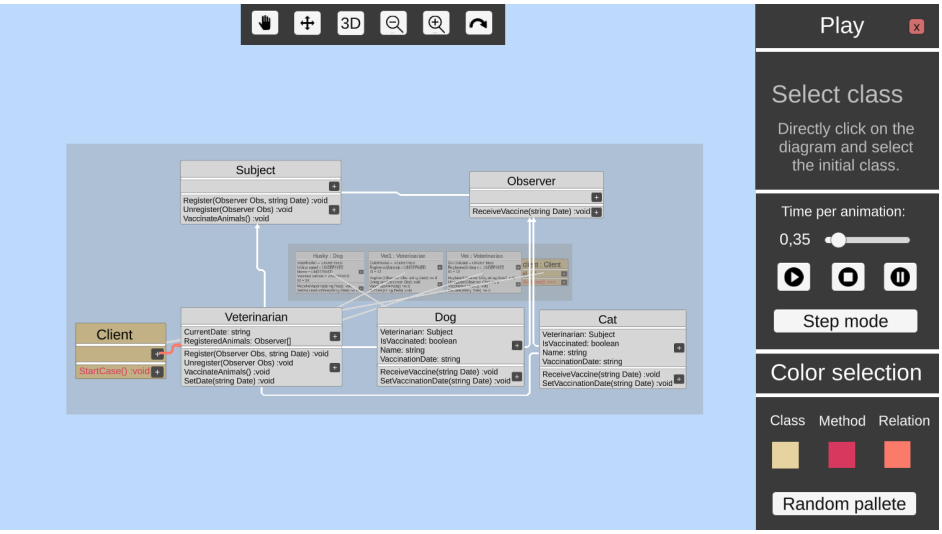
Obr. 1.6: Pôvodná obrazovka editovania

Tlačítko na editovanie diagramu nás dostane do editora, ktorého súčasný stav je vidieť na obr. 1.6. Tam sa nachádza tlačítko na pridanie tried a vzťahov medzi nimi. Každá trieda taktiež obsahuje tlačítko na pridanie metódy a tlačítko na pridanie atribútu. V súčasnosti však funguje iba pridávanie tried, ostatné tlačítka sú nefunkčné a taktiež sa nedá dostať späť na hlavnú obrazovku. Keďže vytvorenie funkčného editora nebolo doteraz prioritou, vzhľadom na to, že na vytváranie diagramov sa používalo prostredie *Enterprise Architect*, tieto funkcionality neboli nikdy dokončené. Naším cieľom je kompletne prerobiť editor a dostať ho do stavu, kedy umožňuje nie len pridávať nové triedy, vzťahy, atribúty a metódy, ale aj editovať už vytvorené komponenty diagramu a taktiež ich mazať.

Po načítaní diagramu je možné vytvoriť nový alebo načítať existujúci scenár pre animáciu. Animácia prebieha súčasne v dvoch vrstvách, v triednej a objektovej, medzi ktorými sa vieme prepínať pomocou hornej lišty s nástrojmi.

Do režimu vytvárania scenára pre animáciu sa dostaneme kliknutím na tlačítko *Create* na hlavnej obrazovke. V tomto režime vieme vytvárať skript pre animáciu klikaním na triedy a vypísané metódy. Pri vytváraní skriptu sa špecifikuje aká metóda sa zavolala a aká nasledujúca metóda sa má zavolať, či už z tej istej triedy alebo z inej triedy, ktorá je vo vzťahu s touto triedou. Počas klikania na komponenty diagramu sa automaticky generuje OAL kód, ktorý vieme manuálne upraviť v textovom poli v pravom dolnom rohu alebo rozšíriť o vytváranie objektov a taktiež validovať jeho korektnosť. Po vytvorení celej animácie ju vieme uložiť do formátu *json*. Prostredie taktiež umožňuje editovanie už vytvorených animácií, ktoré funguje rovnakým spôsobom.

Po načítaní skriptu sa vieme dostať do režimu prehrávania animácie pomocou tlačítka *Play* na hlavnej obrazovke. Po prepnutí sa do tohto režimu musíme zvoliť počiatočnú metódu,



Obr. 1.7: Pozastavená animácia v AnimArchu

od ktorej sa skript spustí, a až potom vieme animáciu spustiť. Po spustení skriptu sa zobrazí objektový diagram, ktorého komponenty sa inicializujú na základe definovaného OAL skriptu a budú sa postupne zvýrazňovať rôzne komponenty diagramu ako je vidieť na obr. 1.7. Keďže sa vieme prepínať medzi triednou a objektovou vrstvou, priebeh animácie vieme sledovať v oboch vrstvách naraz. Počas animácie vieme upravovať dĺžku jednotlivých krokov alebo animáciu pozastaviť a opäť spustiť.

1.4.3 Architektúra

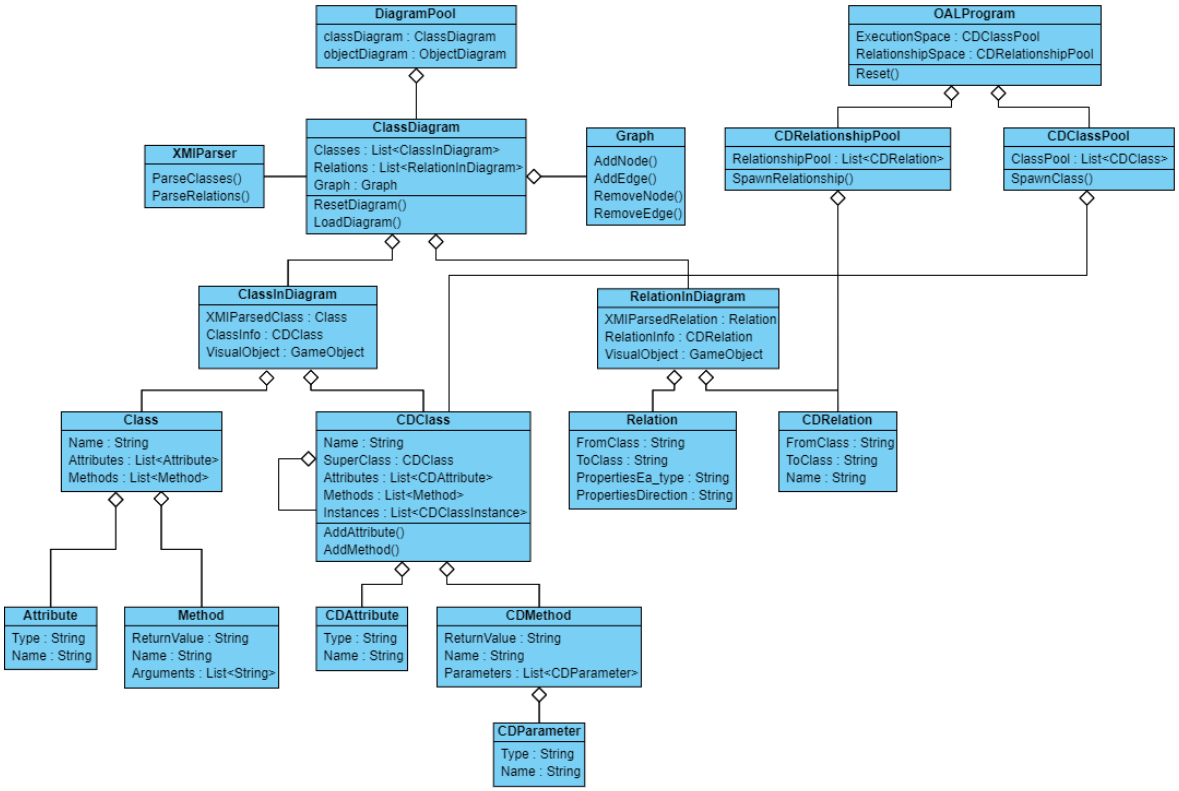
Kompletná architektúra prostredia AnimArch je pri jeho rôznych funkciách veľmi rozsiahla. Pre naše potreby preto opíšeme len relevantnú časť pre našu prácu a to konkrétne internú reprezentáciu triednej vrstvy a objektov s ňou spojených znázornenú triednym diagramom obr. 1.8.

Ako bolo spomenuté, v AnimArchu pracujeme v súčasnosti s triednou a objektovou vrstvou. Triedna vrstva je reprezentovaná triedou ClassDiagram a Objektová ObjectDiagram a ich inštancie sú uložené v triede DiagramPool.

Trieda ClassDiagram okrem uchovávania všetkých tried a vzťahov obsahuje metódy na inicializovanie diagramu zo súboru, vyčistenie celého diagramu pri načítavaní ďalšieho, ako aj metódy na vykreslenie diagramu.

Triedna vrstva existuje v AnimArchu súčasne v troch reprezentáciách. Prvou z nich je reprezentácia pomocou vizuálnych objektov prostredia Unity, s ktorými priamo interaguje používateľ. Vizuálnu reprezentáciu zastrešuje trieda Graph, ktorá okrem iného obsahuje metódy na korektnú inicializáciu a deštrukciu Unity GameObjectov pre triedy a vzťahy.

Ďalšou reprezentáciu sú zjednodušené triedy, ktoré sa priamo načítavajú zo súborov a obsahujú iba základné údaje. Poslednou reprezentáciou je reprezentácia tried spadajúcich pod OAL. Ide o triedy, s ktorým priamo pracuje Object Action Language a vďaka ktorým je možná jeho exekúcia. Táto reprezentácia je podobná metamodelu, každá trieda obsahuje referencie na svoje superclassy, obsahuje kontroly na typy a mnoho ďalšieho, avšak pre lepšiu



Obr. 1.8: Zjednodušený triedny diagram reprezentácie diagramov v AnimArchu

prehľadnosť nie sú na diagrame znázornené všetky tieto atribúty a metódy. Pomocou tejto reprezentácie taktiež vedia fungovať animácie.

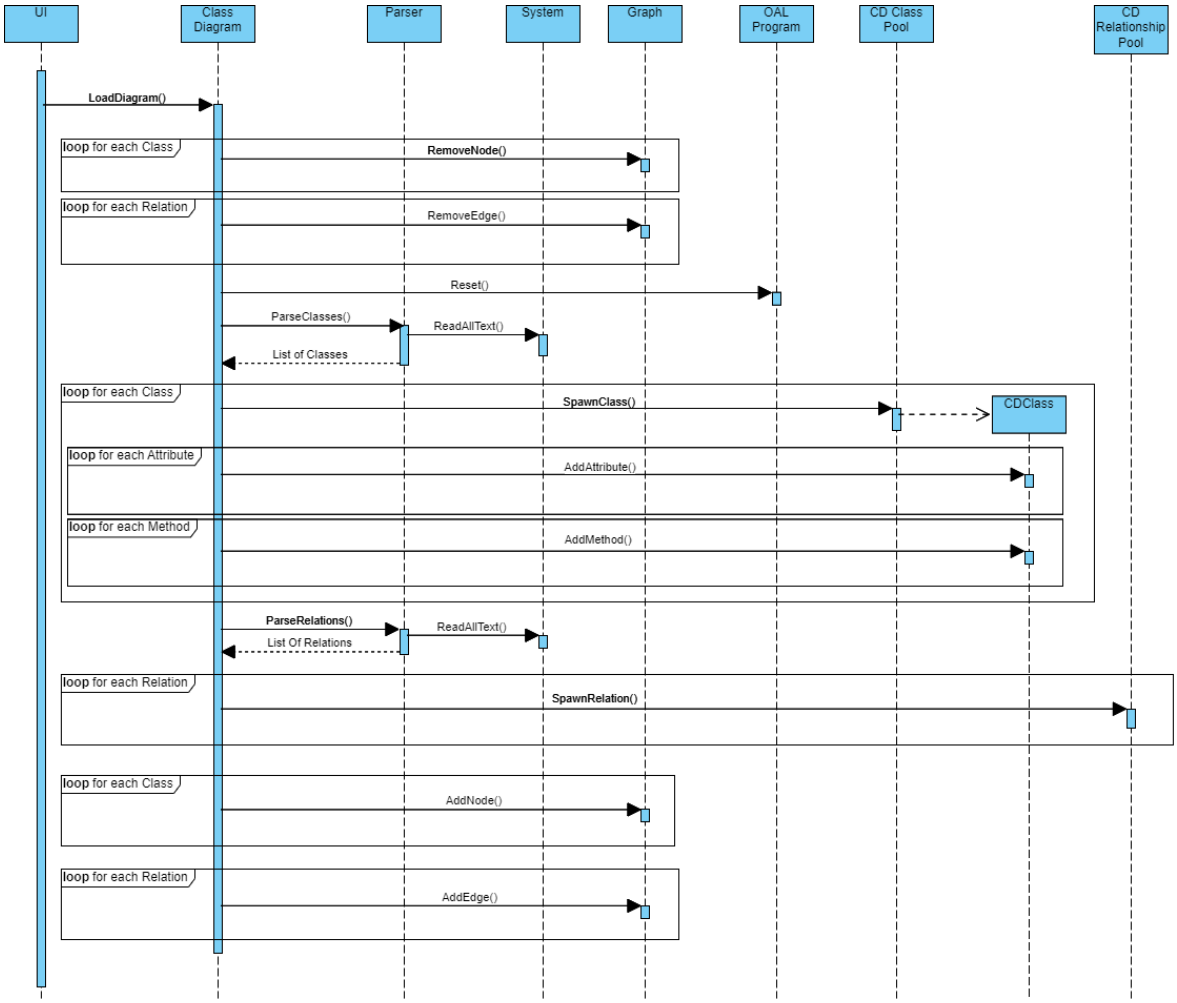
Všetky tieto tri reprezentácie sú prepojené v triede ClassInDiagram, ktorá slúži na zaobalenie jednej samostatnej triedy. Tá obsahuje tri atribúty. VisualObject, čo predstavuje Unity GameObject, čiže vizuálnu reprezentáciu triedy. Ďalej obsahuje XMIParsedClass, čo je spomínaná zjednodušená reprezentácia získaná priamo zo súborov. Využívajú sa na to triedy Class, Attribute a Method. Posledným atribútom je ClassInfo. Ide o reprezentáciu spadajúcu pod OAL, ktorá je realizovaná pomocou CDClass, CDAttribute, CDMethod a CDParameter.

Rovnako ako triedy aj vzťahy majú tri reprezentácie v rámci AnimArchu a sú prepojené v triede RelationInDiagram, ktorá zaobaľuje jednu samostatnú triedu. Je v nej uložený VisualObject, XMIParsedRelation ako aj RelationInfo.

Triedy poskytujúce OAL reprezentáciu tried, spadajú pod CDRelationshipPool a CDClassPool. Tieto obsahujú okrem iného metódy na ich korektnú inicializáciu a ich inštancie sú zlúčené v triede OALProgram, ktorá obsahuje metódu na ich kompletné prečistenie.

Pre lepšie pochopenie, ako medzi sebou jednotlivé triedy komunikujú, si popíšeme scenár pre načítavanie a inicializovanie diagramu zo súboru. Potom ako používateľ zvolí, že chce načítať diagram, sa z UI zavolá metóda LoadDiagram triedy ClassDiagram. V prvom rade je potrebné vymazať momentálne zobrazený diagram, čiže treba odstrániť jeho GameObjecty, resetovať OAL reprezentáciu a taktiež sa v rámci tejto triedy premažú zoznamy tried a metód.

Po premazaní diagramu zavolá ClassDiagram Parser, aby načítal zo súboru triedy a ten



Obr. 1.9: Sekvenčný diagram načítania diagramu zo súboru do AnimArchu

ich po korektnom sparsovaní vráti ako zoznam. Najprv sa pre každú triedu interne vytvorí nová ClassInDiagram, ktorej sa nastaví XMIParsedClass na triedu načítanú zo súboru. Potom sa prechádza celým zoznamom tried a pre každú sa postupne vytvorí CDClass reprezentácia a uloží do prislúchajúceho ClassInDiagram.

Následne sa opäť zavolá Parser, aby zo súboru načítal vzťahy a vrátil ich zoznam. Rovnako ako pri inicializovaní tried, sa najprv v rámci ClassDiagram vytvorí RelationInDiagram, ktorej sa nastaví XMIParsedRelation. Potom sa inicializuje CDRelation.

Na záver sa pre každú triedu a potom pre každý vzťah zavolá prislúchajúca metóda v triede Graph a takto získaná reprezentácia sa uloží do prislúchajúcej ClassInDiagram a RelationInDiagram.

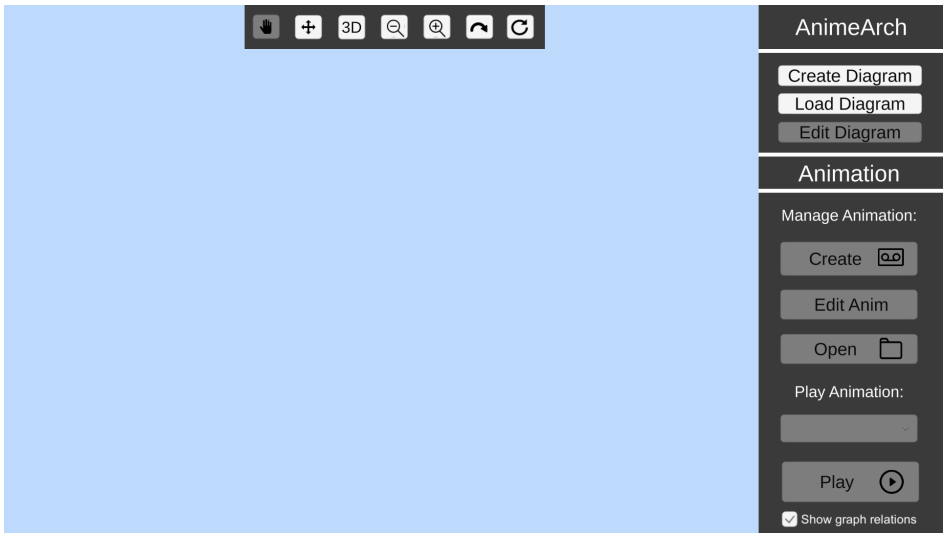
Kapitola 2

Návrh úprav AnimArchu pre potreby editora

V tejto kapitole popisujeme rozšírenie používateľského rozhrania, štruktúru tried slúžiacich na editovanie rôznych častí triedneho diagramu v prostredí AnimArch, výhody a nevýhody formátov súborov, do ktorých budeme diagramy ukladať, ako aj rozhodnutie ohľadom plytkého a hlbokého editovania.

2.1 Rozšírenia používateľského rozhrania

Pri rozšírení používateľského rozhrania aplikácie AnimArch o nové funkcionality sme sa snažili zachovať existujúci grafický dizajn, ale zároveň vylepšiť niektoré jeho nedostatky. Súčasťou nášho riešenia bola úprava hlavnej obrazovky, vytvorenie novej obrazovky na editovanie, navrhnutie okien na vytváranie, editovanie a mazanie častí diagramu a rozšírenie reprezentácie tried a vzťahov o tlačítka na ich editovanie a mazanie.



Obr. 2.1: Nová hlavná obrazovka

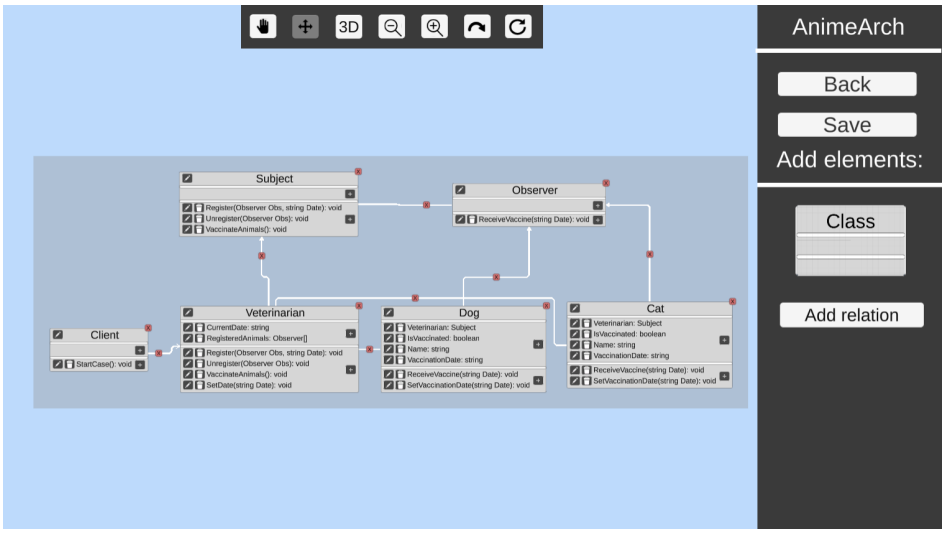
Hlavná obrazovka

Pôvodná hlavná obrazovka obsahovala bočný panel, ktorého súčast'ou bolo tlačítko na načítanie diagramu, nefunkčné tlačítko na editovanie diagramu a rôzne tlačítka na správu animácií. Taktiež obsahovala hornú lištu s rôznymi nástrojmi na prácu s diagramom, napríklad na pohyb s triedami, pohyb s celým diagramom, alebo oddialenie/priblíženie celého diagramu.

V rámci našej práce sme rozšírili bočný panel o tlačítko vytvorenia nového diagramu. Ďalej sme upravili správanie tlačítiek na správu animácie, aby sme zamedzili ich používaniu v prípade, že nemáme zobrazený žiadny diagram. Hornú lištu s nástrojmi sme tiež upravili tak, aby sa nástroje nedali používat' v prípade, že nemáme načítaný žiadny diagram. Lištu sme tiež obohatili o tlačítko *Reset*, ktoré vráti náš pohľad na diagram na pôvodnú pozíciu.

Obrazovka na editovanie diagramu

Po prekliknutí sa na obrazovku editovania diagramu, sa používateľovi zobrazia na komponentoch diagramu tlačítka na ich úpravu. Ak je aplikácia v iných režimoch ako v režime editovania, tlačítka sa nezobrazujú vôbec. Pri názve triedy sa zobrazí tlačítko na zmenu názvu, pri každom atribúte a metóde sa zobrazia tlačítka na ich editovanie alebo vymazanie. Taktiež sa na diagrame zobrazia tlačítka na vymazanie triedy a vymazanie relácie. Po kliknutí na tieto tlačítka sa zobrazí prislúchajúce okno.

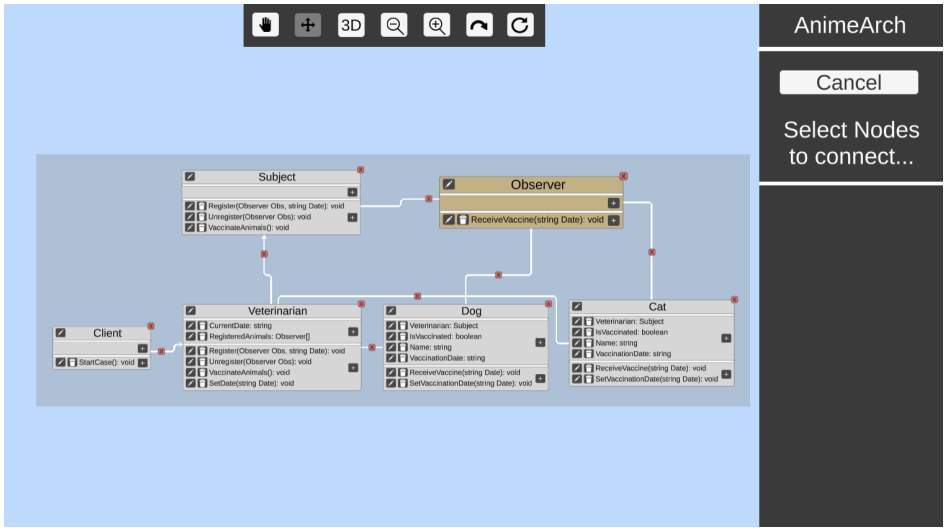


Obr. 2.2: Nová obrazovka editovania

Na bočnom paneli sú tlačítka na vrátenie sa na hlavnú obrazovku, tlačítko na uloženie súčasného diagramu aj s jeho rozložením a tlačítka na pridanie triedy a pridanie relácie. Po kliknutí na uloženie sa používateľovi zobrazí prehliadač súborov, kde si môže zvolit' kam a do akého formátu si praje uložit' diagram. Po kliknutí na tlačítko pridania triedy sa otvorí okno na správu novej triedy.

Po kliknutí na tlačítko pridania relácie sa najprv používateľovi zobrazí okno na voľbu typu relácie. Následne sa editor prepne do stavu výberu tried, ktoré má relácia spájat'. V prípade označenia dvoch tried sa medzi nimi vytvorí relácia a editor sa prepne naspät' do

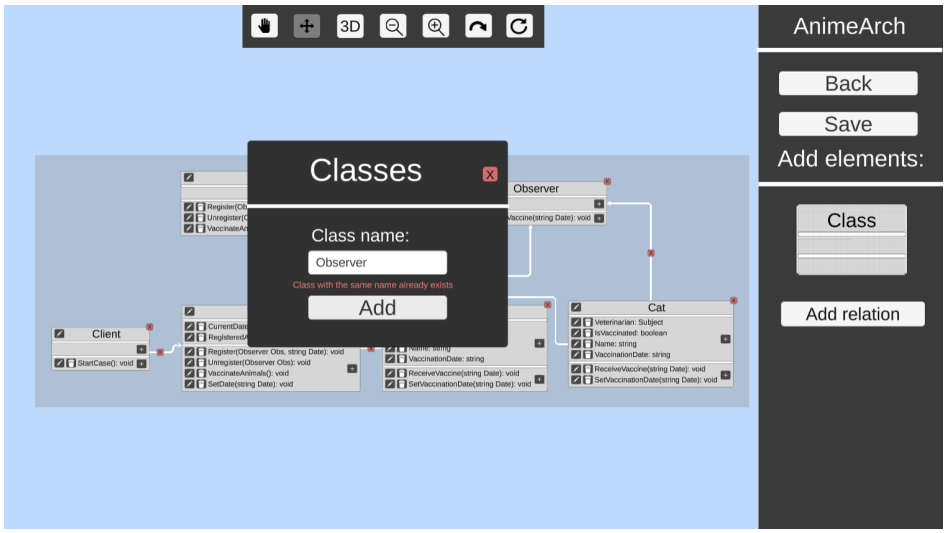
základného stavu. Ak však relácia daného typu už medzi zvolenými triedami existuje, editor na to používateľa upozorní formou okna so správou o chybe.



Obr. 2.3: Editor v stave pridania hrany medzi dvomi triedami

Okno na správu triedy

Po kliknutí na tlačítko pridania triedy sa zobrazí okno na správu triedy. Do textového poľa vie používateľ zadať meno novej triedy. Meno triedy musí začínať písmenom alebo podtržníkom a ďalej môže obsahovať aj čísla. Používateľovi nie je umožnené napísať nekorektný znak do textového poľa. V prípade, že používateľ nezadá žiadne meno, alebo zadá meno triedy, ktorá už existuje, zobrazí sa správa s konkrétnou chybou. Inak sa pridá nová trieda.



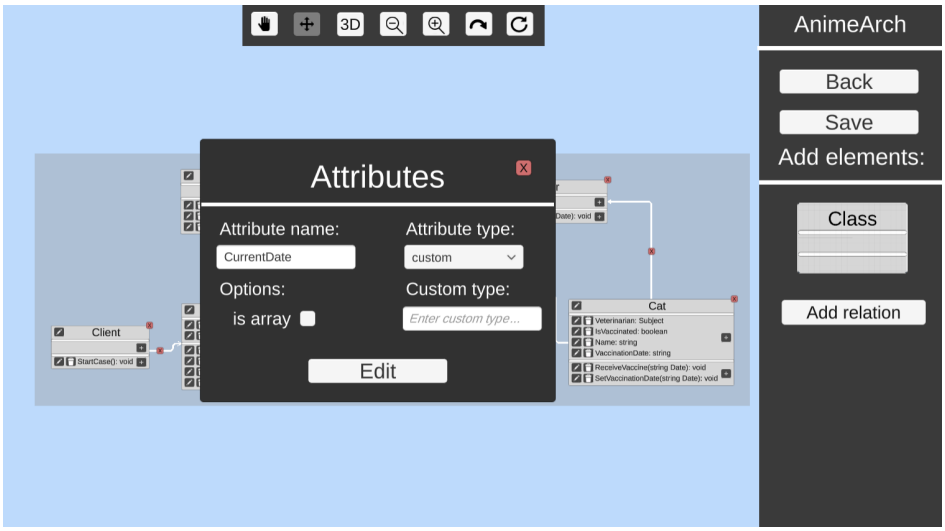
Obr. 2.4: Okno na správu triedy

Okno so správou triedy sa taktiež zobrazí po kliknutí na tlačítko s ceruzkou pri mene niektorej z existujúcich tried. V takom prípade bude textové pole predvyplnené pôvodným menom triedy. Rovnako ako pri vytváraní triedy musí používateľ korektne vyplniť textové

pole a následne sa upraví meno pôvodnej triedy.

Okno na správu atribútu

Po kliknutí na tlačítko pridania atribútu v hornej časti triedy sa otvorí okno na správu atribútu. V rámci okna používateľ musí zadať meno atribútu do textového poľa, ktoré podlieha rovnakým obmedzeniam ako meno triedy, popísané vyššie. Ďalej zvolí typ atribútu. Na výber má základné typy, typy tried, ktoré už existujú v diagrame, a tiež ma na výber vlastný typ. V prípade voľby vlastného typu sa zobrazí d'alšie textové pole na zadanie tohto typu. Taktiež môže používateľ zvoliť, či je daný atribút zoznamom. V prípade korektného zadania všetkých údajov sa pridá nový atribút do danej triedy, inak sa zobrazí správa s konkrétnou chybou.



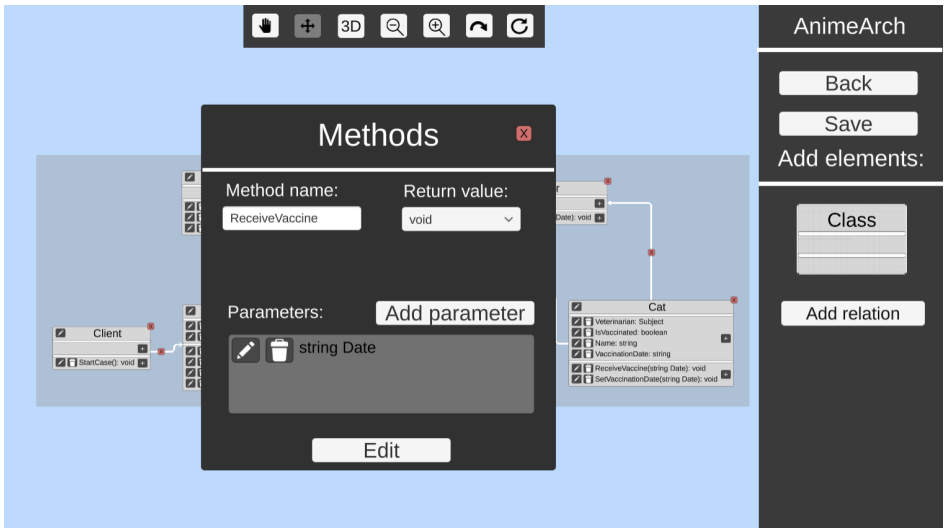
Obr. 2.5: Okno na správu atribútu

Okno na správu atribútu sa taktiež zobrazí po kliknutí na tlačítko s ceruzkou pri atribúte. V takom prípade sa všetky údaje predvyplnia na základe pôvodnej definície atribútu a po korektnom zadaní všetkých údajov sa upraví pôvodný atribút.

Okno na správu metódy

Po kliknutí na tlačítko pridania metódy v dolnej časti triedy sa otvorí okno na správu metódy. V rámci okna používateľ musí zadať meno metódy do textového poľa, ktoré podlieha rovnakým obmedzeniam ako meno triedy a atribútu, popísané vyššie. Ďalej zvolí výstupný typ metódy. Na výber má rovnaké typy ako pri výbere typu atribútu aj s vlastným typom. Rozdiel oproti oknu s atribútom je v paneli s parametrami metódy. Má na výber tlačítko pridania parametra a taktiež môže upraviť alebo vymazať niektorý z existujúcich parametrov. V prípade korektného zadania všetkých údajov, sa pridá nová metóda do danej triedy, inak sa zobrazí správa s konkrétnou chybou.

Rovnako ako pri atribútoch a menách tried, v prípade, že používateľ zvolí editovanie existujúcej metódy, okno sa zobrazí s predvyplnenými údajmi a po korektnom zadaní všetkých údajov, sa upraví pôvodná metóda.



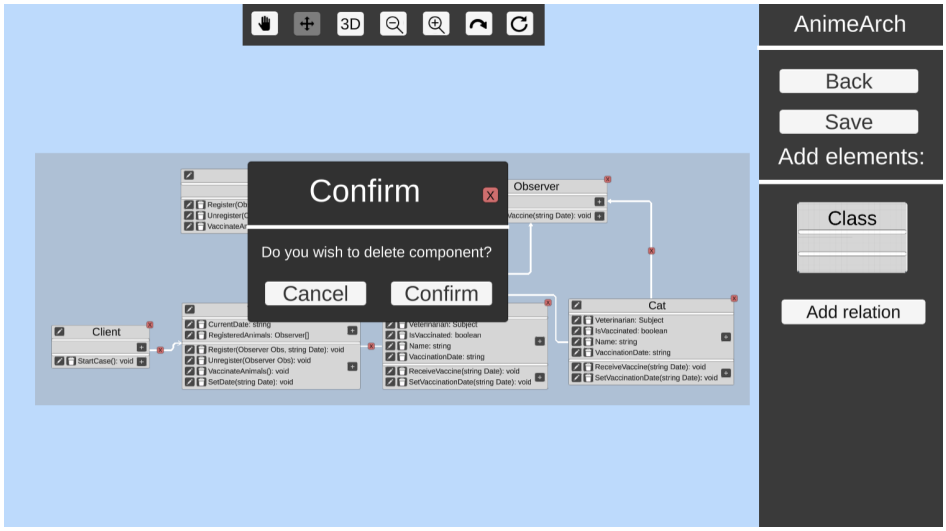
Obr. 2.6: Okno na správu metódy

Okno na správu parametra

Okno na správu parametra sa zobrazí po kliknutí na tlačítko pridania parametra alebo po kliknutí na tlačítko úpravy parametra v okne na správu metódy. Obsahom je totožné s oknom na správu atribútu a po správnom vyplnení všetkých údajov sa pridá parameter danej metóde a zobrazí sa opäť okno na správu metódy.

Okno na vymazanie komponentu diagramu

Okno na vymazanie komponentu diagramu sa pustí po kliknutí na ktorékoľvek z tlačítiek vymazania relácie, triedy, atribútu, metódy alebo parametra metódy. Ide o potvrdzovacie okno, či si používateľ skutočne praje odstrániť daný komponent a preto obsahuje tlačítko na potvrdenie tohto kroku a tlačítko na jeho zrušenie.



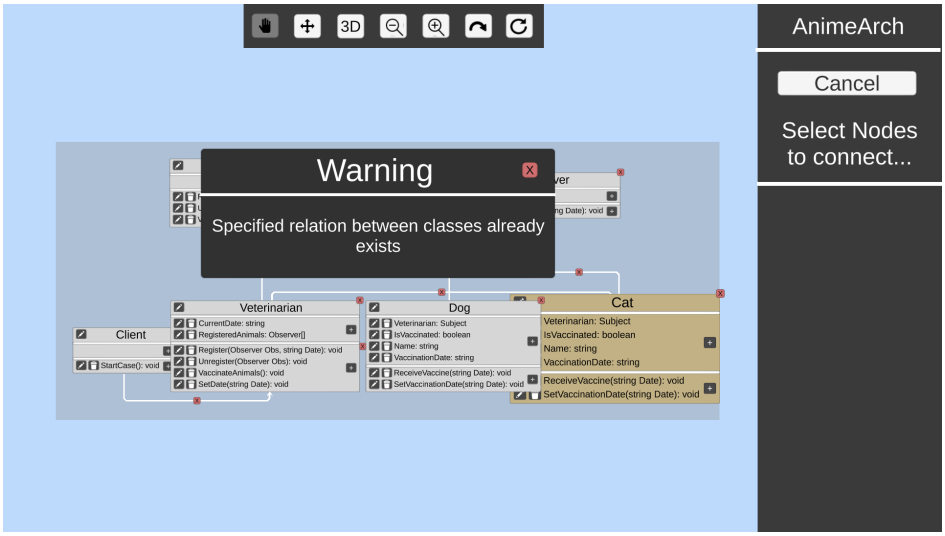
Obr. 2.7: Okno na potvrdenie vymazania komponentu

Okno na ukončenie AnimArchu

Okno na ukončenie AnimArchu je možné spustiť z akejkoľvek scény kliknutím na tlačítko *ESC* na klávesnici. Toto okno obsahuje rovnako ako okno na potvrdenie vymazania komponentu diagramu tlačítko na jeho zavretie, na jeho zrušenie alebo na potvrdenie, že si používateľ skutočne želá vypnúť AnimArch. V prípade, že sa používateľ rozhodne zavrieť toto okno, beh programu alebo spustenej animácie pokračuje ďalej od momentu, kedy bolo okno aktivované.

Okno so správou o chybe

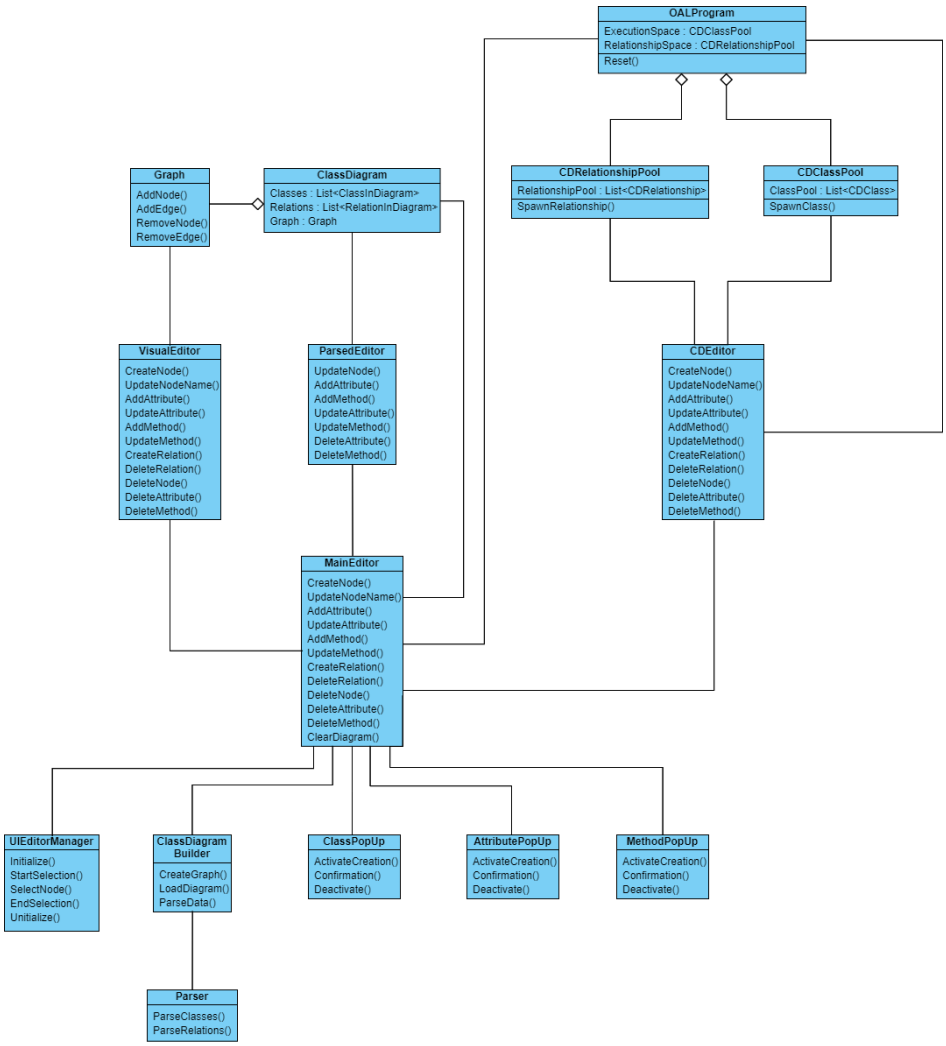
Okno so správou o chybe sa spúšťa, keď dôjde k chybe, napríklad ak používateľ chce pridať reláciu medzi dve triedy, ktoré túto reláciu už obsahujú, a túto chybu vypíše. Obsahuje iba tlačítko na zavretie okna.



Obr. 2.8: Okno so správou o chybe

2.2 Štruktúra tried na editovanie diagramu

V našom pôvodnom návrhu sme rozmýšľali o zachovaní metód na vytváranie a editovanie triedneho diagramu v jednej novovytvorenej triede. Tento prístup sa však rýchlo ukázal ako nevhodný vzhľadom na to, že každá úprava diagramu musí byť vykonaná na všetkých troch reprezentáciách diagramu - na vizuálnej reprezentácií, zjednodušenej reprezentácií diagramu načítanej zo súboru aj OAL reprezentácii diagramu. Vzhľadom na tento fakt sme sa nakoniec rozhodli rozdeliť metódy na úpravu každej vrstvy do samostatnej triedy.

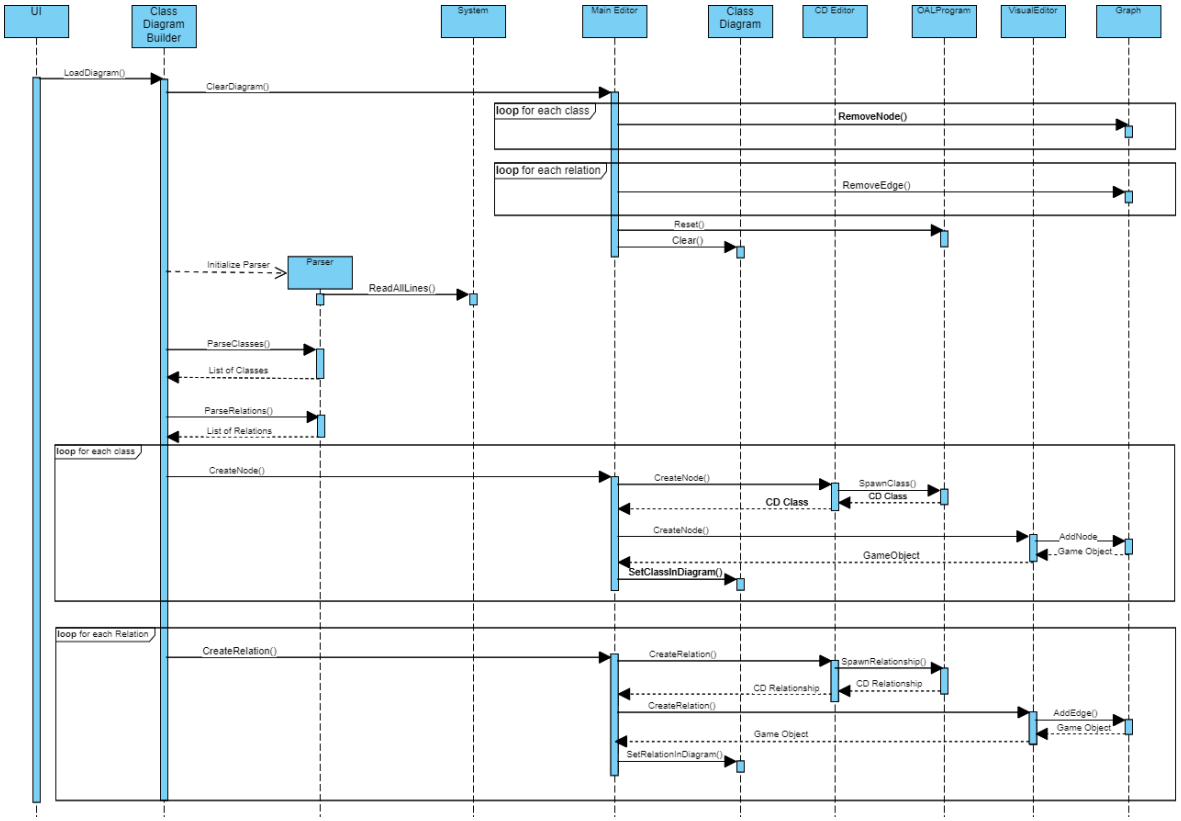


Obr. 2.9: Diagram tried slúžiacich na editovanie ClassDiagram

Ako je vidieť na diagrame obr. 2.9, každá reprezentácia má vlastnú triedu na jej úpravu. To predstavuje triedy ParsedEditor, ktorá komunikuje priamo s triedou ClassDiagram, VisualEditor, ktorá komunikuje priamo s triedou Graph a CDEditor, ktorá pracuje priamo s OAL reprezentáciou diagramu. Všetky obsahujú metódy na pridanie, odstránenie alebo úpravu komponentov diagramu.

Nad týmito editormi sedí MainEditor. Na základe návrhového vzoru Facade, predstavuje jednotné rozhranie, s ktorým bude komunikovať zvyšok aplikácie v prípade, že chce akokoľvek upraviť triedny diagram. MainEditor obsahuje v každej metóde postupné volanie všetkých

troch editorov a uloženie vykonanej zmeny do ClassDiagram. Tento prístup umožňuje podstatne jednoduchšiu čitateľnosť a taktiež kontrolu, že sa vykonajú všetky kroky nutné na korektnú úpravu diagramu.



Obr. 2.10: Nový sekvenčný diagram načítania diagramu zo súboru do AnimArchu

Poslednou navrhovanou zmenou v tejto časti je vyňatie metód na inicializáciu ClassDiagram zo súboru do samostatnej metódy ClassDiagramBuilder. Takým spôsobom bude ClassDiagram obsahovať metódy na získavanie údajov o diagrame, ale nebude obsahovať žiadne metódy na ich úpravu. ClassDiagramBuilder bude rovnako ako všetky ostatné triedy volať metódy MainEditor, čiže nebude priamo pristupovať ku ClassDiagram.

Na lepšie pochopenie prerozdelenia zodpovedností medzi novými triedami si opäť popíšeme scenár načítavania triedneho diagramu zo súboru, popísanom na sekvenčnom diagrame obr. 2.10. Potom ako používateľ zvolí možnosť načítania diagramu zo súboru sa zavolá metóda LoadDiagram triedy ClassDiagramBuilder. ClassDiagramBuilder najprv zavolá metódu ClearDiagram. MainEditor počas vykonávania ClearDiagram postupne premaže Graph, OAL reprezentáciu a na záver premaže zoznam ClassInDiagram nachádzajúci sa v triede ClassDiagram.

Po premazaní diagramu, ClassDiagramBuilder inicializuje Parser, ktorý si pri inicializácií prečíta súbor. Následne sa zavolajú metódy ParseClasses a ParseRelations, ktoré vrátia zoznam tried a zoznam vzťahov. Pre každú triedu získanú z parsera sa potom zavolá metóda CreateNode triedy MainEditor. Tá najprv zavolá CreateNode triedy CDEditor, čím vytvorí triedu v OAL reprezentácii diagramu a vráti takto získanú triedu. Následne sa zavolá Crea-

teNode triedy VisualEditor, ktorý vytvorí Unity GameObject v triede Graph a tento vráti. Na záver sa vytvorí ClassInDiagram, do ktorej sa uložia všetky tri reprezentácie triedy a tá sa pridá do triedy ClassDiagram. Rovnakým spôsobom sa pridá aj každý vzťah volaním MainEditor.

2.3 Plytké a hlboké editovanie

Pri práci na akomkoľvek editore je dôležité si položiť otázku, či chceme implementovať hlboké alebo plytké editovanie. Plytké editovanie znamená v princípe, že ak upravíme definíciu nejakého komponentu, neupraví sa na všetkých miestach, kde sa doteraz tento komponent používal. Upraví sa iba tam, kde sme ho zmenili. Hlboké editovanie znamená presný opak, definícia sa zmení na všetkých miestach.

V kontexte triednych diagramov si to vieme predstaviť napríklad pri zmene názvu triedy. V prípade, že zmeníme názov triedy a táto trieda je použitá ako typ atribútu nejakej inej triedy, tak ak sa zmení názov triedy a rovno sa zmení aj typ atribútov ostatných tried, ktoré túto triedu používali ako typ, ide o hlboké editovanie. V prípade, že sa zmení iba meno triedy a ostane staré meno pri typoch atribútov, ide o plytké editovanie.

V kontexte našej práce sme sa rozhodli využívať hlboké editovanie pri zmene názvov tried, a plytké editovanie v prípade, že sa trieda vymaže, keďže umožňujeme mať aj vlastné typy, ktoré nemusia byť reprezentované samostatnou triedou vo vytvorenom triednom diagrame.

2.4 Ukladanie diagramov do súborov

Súčasťou našej práce je pridať možnosť ukladania vytvoreného diagramu do súboru. Pôvodne sa na vytváranie diagramov využíval spomínaný program *Enterprise Architect*, z ktorého sa diagram exportoval vo formáte *xmi* a následne načítal do AnimArchu. Našim prvotným cieľom bolo ukladať súbory tiež v tomto formáte a týmto spôsobom zachovať spätnú kompatibilitu s *Enterprise Architect* a to takým spôsobom, že diagram vytvorený v AnimArchu bude možné otvoriť v tomto programe.

XMI je štandardný formát na báze XML slúžiaci na zdieľanie UML diagramov medzi rôznymi programami, v ktorých je možné s nimi pracovať. XMI súbor exportovaný z *Enterprise Architect* sa skladá z dvoch hlavných častí, xml elementov - model a extension. Model obsahuje package a ten ďalej obsahuje samotné komponenty diagramu - triedy a vzťahy medzi nimi. Časť extension obsahuje taktiež popis tried a vzťahov s referenciami na ich definície z časti model, avšak obsahuje dodatočné informácie napríklad o polohách a štýloch, ktoré sú viac špecifické programu *Enterprise Architect*. [2, 3] Viac o praktickej implementácií XMI pre potreby programov na prácu s UML je v [11].

Vzhľadom na komplexnosť programu *Enterprise Architect* je rozsiahlosť informácií v súboroch z neho exportovaných celkom veľká. AnimArch na inicializáciu diagramu zo súboru nevyužíva všetky tieto informácie, opiera sa najmä o údaje z časti extension. Na uloženie diagramu do tohto formátu, tak aby sme vedeli využiť existujúci parser z AnimArchu preto

nie je potrebné vyplnit' všetky informácie. Na zachovanie spätnej kompatibility s *Enterprise Architect* by bolo potrebné však všetky tieto údaje generovat', vd'aka čomu dostaneme súbor, ktorý nie je prehl'adný, a samotný Parser na generovanie všetkých týchto údajov by bol veľmi komplikovaný.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
    <xmi:Documentation/>
    <xmi:Model/>
    <xmi:Extension extender="AnimArch" extenderID="1.0">
        <elements>
            <element xmi:Type="uml:Package" name="name" scope="public"/>
            <element xmi:type="uml:Class" name="FirstClass"
            xmi:idref="1e83ebb6-cb6c-4eb1-84bb-4d75d75792fb"/>
            <element xmi:type="uml:Class" name="SecondClass"
            xmi:idref="5776f3ea-2d9e-4d88-a824-09649cb1892d"/>
        </elements>
        <connectors>
            <connector xmi:idref="FirstClassSecondClassAssociation">
                <source>
                    <model type="Class" name="FirstClass"/>
                </source>
                <target>
                    <model type="Class" name="SecondClass"/>
                </target>
                <properties ea_type="Association"
                direction="Source -&gt; Destination"/>
            </connector>
        </connectors>
        <primitivetypes/>
        <diagrams>
            <diagram>
                <model localID="1"/>
                <elements>
                    <element subject="1e83ebb6-cb6c-4eb1-84bb-4d75d75792fb"
                            geometry="Left=-103;Top=-319;Right=0;Bottom=0;"/>
                    <element subject="5776f3ea-2d9e-4d88-a824-09649cb1892d"
                            geometry="Left=159;Top=-320;Right=0;Bottom=0;"/>
                </elements>
            </diagram>
        </diagrams>
    </xmi:Extension>
</xmi:XMI>
```

Kód 2.1: Ukážka triedneho diagramu vo formáte XMI exportovaná z AnimArchu

Nakoniec sme preto ukladanie do súborov typu XMI zanechali vo verzii, kedy súbor vie prečítat' AnimArch, ale nie je možné ho importovat' do *Enterprise Architect*. Jednoduchý diagram obsahujúci dve prázdne triedy a jednu reláciu medzi nimi exportovaný z AnimArchu vo formáte XMI je vyobrazený aj kódom 2.1.

Na ukladanie animácií v AnimArchu sa využívajú súbory typu JSON. Preto sme sa rozhodli taktiež pridat' možnost' ukladania diagramu aj do tohto typu súboru. Vzhl'adom na to, že pri JSON nie sme viazaní na konkrétnu reprezentáciu, rozhodli sme sa pre priamu serializáciu tried Class, Attribute, Method a Relation existujúcich v AnimArchu do tohto súboru.

Vd'aka tomu bola implementácia parsera pre JSON veľmi priamočiara a samotné súbory sú podstatne viac prehľadnejšie a čitateľnejšie ako je vidieť aj kódom 2.2 vyobrazujúcom rovnaký diagram ako kód 2.1.


```
{"classes": [
    {
      "Name": "FirstClass",
      "Id": "1e83ebb6-cb6c-4eb1-84bb-4d75d75792fb",
      "Geometry": "Left=-103;Top=-319;Right=0;Bottom=0;",
      "Left": -103.0,
      "Right": 0.0,
      "Top": -319.0,
      "Bottom": 0.0,
      "Type": "uml:Class",
      "Attributes": [],
      "Methods": []
    },
    {
      "Name": "SecondClass",
      "Id": "5776f3ea-2d9e-4d88-a824-09649cb1892d",
      "Geometry": "Left=159;Top=-320;Right=0;Bottom=0;",
      "Left": 159.0,
      "Right": 0.0,
      "Top": -320.0,
      "Bottom": 0.0,
      "Type": "uml:Class",
      "Attributes": [],
      "Methods": []
    }
  ],
"relations": [
    {
      "ConnectorXmiId": "FirstClassSecondClassAssociation",
      "SourceModelType": "Class",
      "SourceModelName": "FirstClass",
      "SourceTypeAggregation": null,
      "TargetModelType": "Class",
      "TargetModelName": "SecondClass",
      "PropertiesEaType": "Association",
      "PropertiesDirection": "Source -> Destination",
      "FromClass": "FirstClass",
      "ToClass": "SecondClass",
      "OALName": "R1"
    }
  ]
}
```


Kód 2.2: Ukážka triedneho diagramu vo formáte JSON exportovaná z AnimArchu

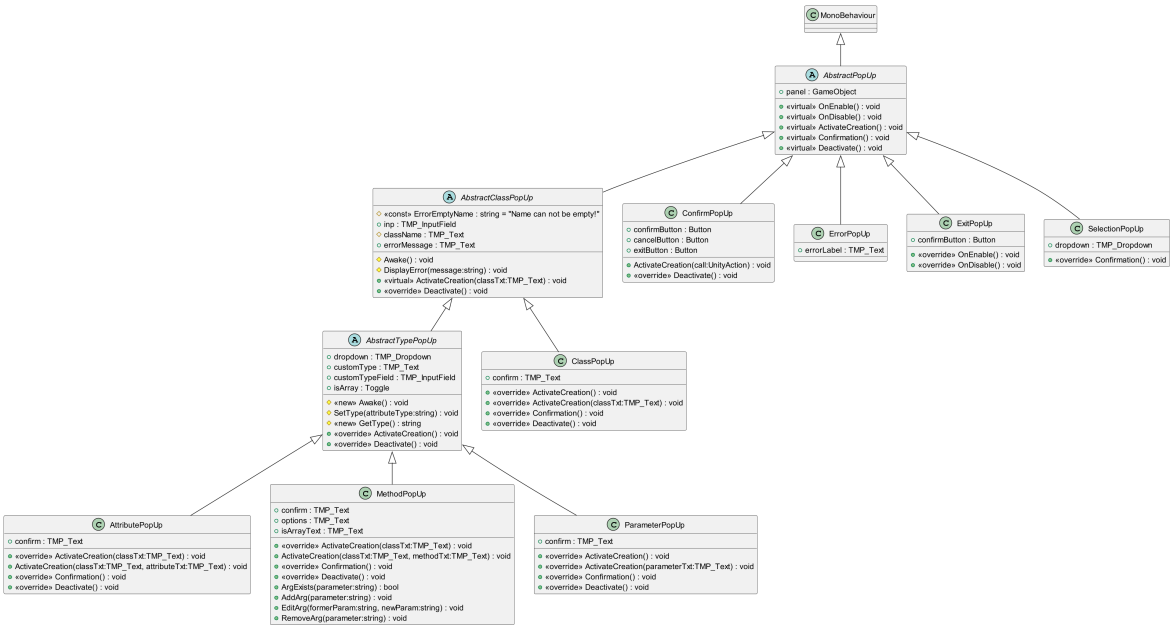
Kapitola 3

Popis realizácie vybraných súčastí editora

V tejto kapitole popíšeme štruktúru a konkrétnu implementáciu okien slúžiacich na editovanie diagramu, popíšeme postupy a algoritmy slúžiace na parsovanie diagramu do súboru. Ďalej popíšeme využitie návrhového vzoru State pri realizácii pridávania hrán do diagramu a na záver si popíšeme spôsob, akým prebieha hlboká editácia názvu tried.

3.1 Modálne okná

V našej práci sme sa rozhodli na editovanie rôznych častí diagramov využívať vlastné modálne okná. Vzhľadom na veľký počet typov okien, sme potrebovali vytvoriť jednotnú štruktúru na obsluhu týchto okien, aby sme sa vyhli duplicite, a aby budúce rozšírenie o akékoľvek ďalšie modálne okno, bolo jednoduché.



Obr. 3.1: Diagram tried obsluhujúcich modálne okná

Ako základ sme sa rozhodli vytvorit' triedu AbstractPopUp, od ktorej budú všetky ostatné okná dedit'. Každé modálne okno má minimálne tri funkcionality a to aktiváciu okna, potvrdenie vykonanej zmeny, napríklad pridanie triedy, a následnú deaktiváciu okna. Každú túto operáciu predstavuje jedna metóda v triede AbstractPopUp a tá obsahuje jej najvšeobecnejšie vykonanie.

Počas toho, ked' chceme mat' vyobrazené okno, chceme mat' všetky ostatné tlačítka alebo pohyblivé objekty deaktivované. Táto funkcionalita sa taktiež vykonáva v tejto triede a to v metódach onEnable a onDisable, čo sú štandardné metódy Unity MonoBehaviour objektov, ktoré sa spustia ked' je daný objekt zobrazený pre používateľa. V týchto metódach sa zavolá súkromná metóda SetButtonsActive zobrazená kódom 3.1. V tejto metóde sa deaktivujú alebo aktivujú na základe parametra active tlačítka bočného ako aj horného menu s nástrojmi a v prípade, že je diagram načítaný, aj všetky tlačítka slúžiace na úpravu diagramu.

```
private static void SetButtonsActive(bool active)
{
    if (DiagramPool.Instance.ClassDiagram.graph != null)
        DiagramPool.Instance.ClassDiagram.graph.GetComponentsInChildren<GraphicRaycaster>()
            .ForEach(x => x.enabled = active);

    var canvas = GameObject.Find("Canvas").transform;
    canvas.Find("RightMenu").GetComponentsInChildren<Button>()
        .Where(x => x.interactable)
        .ForEach(x => x.enabled = active);

    canvas.Find("TopMenu").GetComponentsInChildren<Button>()
        .Where(x => x.interactable)
        .ForEach(x => x.enabled = active);

    canvas.Find("TopMenu").GetComponentsInChildren<EventTrigger>()
        .ForEach(x => x.enabled = active);

    ToolManager.Instance.SetActive(active);
}
```

Kód 3.1: Metóda na aktiváciu a deaktiváciu všetkých tlačítiek

Priamo od tejto triedy dedia modálne okná na potvrdenie zmazania, ukončenie aplikácie, zvolenie pridávanej relácie, okno s chybovou hláškou a AbstractClassPopUp.

Trieda AbstractClassPopUp zabezpečuje základnú kostru pre modálne okná, ktoré obsahujú vstupné pole, napríklad pre meno triedy, ako aj atribút na uloženie názvu triedy, ktorú aktuálne upravujeme. Okrem toho táto trieda obsahuje metódu na zobrazenie chybovej hlášky. Ked'že zaobaľuje táto trieda aj univerzálne vstupné pole, obsahuje aj validáciu, že vstup od používateľa je korektný.

Vzhľadom na to, že z diagramov vytvorených v prostredí AnimArch sa dá priamo generovat' zdrojový kód, je potrebné nové názvy pre diagram validovat' vzhľadom na štandardnú konvenciu pre názvy v programovacích jazykoch. Validácia prebieha tak, ako je zobrazené kódom 3.2 a teda v prípade, že ide o prvý znak nového názvu pre komponent diagramu, je

```
inp.onValueChanged.AddListener(delegate(string arg)
{
    if (string.IsNullOrEmpty(arg))
        return;
    if (arg.Length == 1 && (char.IsLetter(arg[0]) || arg[0] == '_'))
        inp.text = arg;
    else if (arg.Length > 1 && char.IsLetterOrDigit(arg[^1]) || arg[^1] == '_')
        inp.text = arg;
    else
        inp.text = arg[..^1];
});
```

Kód 3.2: Validácia vstupného poľa pre názvy komponentov diagramu

povolené dať akékoľvek písmeno alebo znak "_", ďalej je povolené písať navyše aj cifry. Iné znaky povolené nie sú a používateľovi nie je dovolené nimi písať - automaticky sa po napísaní odstránia.

Od tejto triedy priamo dedí trieda na obsluhu modálneho okna na vytvorenie, resp. úpravu tried, a posledná abstraktná trieda v kontexte modálnych okien a to AbstractTypePopUp. Ide o triedu, ktorá obsahuje obsluhu textových polí a rolovacieho zoznamu slúžiacich na zvolenie typu napríklad atribútu.

Na zvolenie typu používame kombináciu rolovacieho zoznamu, prepínača, či chceme, aby typ bol zoznamom, a textové pole v prípade, že chceme definovať vlastný typ. V rolovacom zozname máme na výber z rôznych štandardných typov, vlastného typu ako aj typov tried, ktoré už máme definované v našom diagrame. Tieto typy sa získavajú jednoducho priamo z triedy ClassDiagram a pri každej aktivácií takéhoto okna sa zoznam typov aktualizuje. Textové pole pre vlastný typ podlieha rovnakej validácii ako ostatné textové polia. Na zistenie, aký typ si používateľ zvolil, slúži metóda GetType.

Od AbstractTypePopUp dedia triedy na obsluhu modálnych okien pre atribúty, metódy a parametre metód. Všetky modálne okná, ktoré slúžia na úpravu komponentov diagramu vedia fungovať v dvoch režimoch - režim pridania komponentu a režim úpravy komponentu. Rozdiel medzi týmito dvoma režimami prebieha v tom, s akými argumentami zavoláme funkciu pre aktiváciu daného modálneho okna.

Na kóde 3.3 môžeme vidieť definície funkcií pre aktiváciu modálneho okna na úpravu atribútov triedy. V prípade, že modálne okno aktivujeme len s názvom triedy, ktorá bude mať daný atribút, spustí sa všeobecná aktivačná metóda definovaná v rodičovskej triede s tým, že sa uloží názov triedy a modálne okno sa aktivuje v režime pridania nového atribútu.

V prípade, že sa modálne okno aktivuje aj s textom atribútu, na základe tohto textu sa vyplnia všetky polia a taktiež zavolá metóda SetType triedy AbstractTypePopUp, aby sa korektne nastavil aj typ atribútu. Ako posledné sa vytiahne z ClassDiagram konkrétny atribút, ktorý upravujeme, aby sme vedeli korektne zobrazovať chybové hlášky, ako aj zavolať metódu na úpravu atribútu miesto metódy na jej pridanie.

```
public override void ActivateCreation(TMP_Text classTxt) {
    base.ActivateCreation(classTxt);
    confirm.text = "Add";
}

public void ActivateCreation(TMP_Text classTxt, TMP_Text attributeTxt) {
    ActivateCreation(classTxt);
    var text = attributeTxt.text.Split(": ");
    var formerName = text[0];

    var attributeType = text[1];
    SetType(attributeType);

    inp.text = formerName;
    _formerAttribute = DiagramPool.Instance.ClassDiagram.FindAttributeByName(className.text,
        formerName);

    confirm.text = "Edit";
}
```

Kód 3.3: Metódy na aktiváciu modálneho okna na úpravu atribútov

Vďaka tomuto prístupu k modálnym oknám vieme mať spoločné funkcionality ako validácia alebo ošetrenie modálnosti okien na jednom mieste. Všetky tieto triedy obsahujú iba metódy na obsluhu UI, neobsahujú metódy na úpravu diagramu. Na reálne zakomponovanie zmien sa využíva MainEditor, ako bolo popísané v kapitole 2.2.

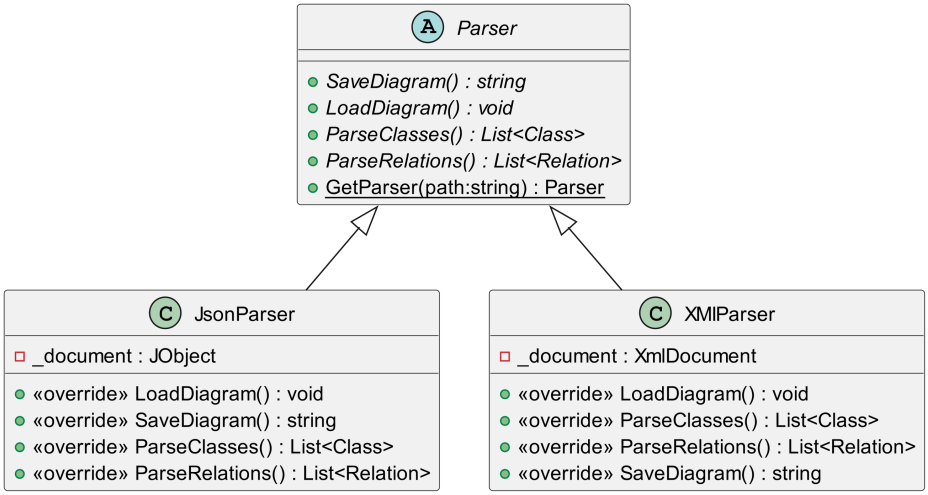
3.2 Parsovanie diagramov

Keďže sme v našej práci vyskúšali ukladať a načítavať diagramy z rôznych typov súborov, vznikla potreba vytvoriť jednotné rozhranie pre všetky takéto parseri. Naša finálna štruktúra je znázornená na obr. 3.2 a obsahuje abstraktnú triedu s metódami na uloženie diagramu do súboru, na otvorenie súboru s diagramom a na samotné prečítanie tried a vzťahov.

Poslednou metódou, ktorú obsahuje parser, je statická metóda GetParser. Táto metóda na základe cesty k súboru zistí typ súboru, ktorý chceme spracovať a podľa toho vráti príslušný parser. Využitím tejto štruktúry vieme na mieste, kde potrebujeme získať parser, zavolať túto metódu a ďalej vieme volať metódy parsera bez ohľadu na to, ktorý konkrétny parser sa momentálne používa.

3.2.1 XMI Parser

Pôvodný XMI parser obsahoval len metódu na načítanie diagramu zo súboru. Jeho rozšírenie sme spravili takým spôsobom, aby bol zásah do tejto existujúcej metódy minimálny. Ako bolo spomenuté v kapitole 2.4 formát súboru XMI je komplexný, a preto aj ukladanie do neho nie je priamočiare, a jeho priebeh si teraz popíšeme.



Obr. 3.2: Diagram tried slúžiacich na parsovanie

Prvým krokom nášho algoritmu je inicializovanie potrebnej štruktúry XMI súboru a to vytvorenie hlavičky obsahujúcej linky na UML a XMI schémy, ktoré používame, a d'alej vytvorenie elementov Documentation, Model a Extension. Existencia elementu Documentation je potrebná na správne načítanie dokumentu zo súboru, avšak jeho obsah nie je pre naše potreby dôležitý, preto ho zanecháme prázdny. Do elementu Model je potrebné vyplniť iba samotné parametre metód, ktoré máme v triedach.

Všetky ostatné informácie o diagrame čerpá naša metóda na načítanie diagramu z elementu Extension. Tento element obsahuje elementy Connectors, PrimitiveTypes, Diagrams a Elements. PrimitiveTypes je podobne ako Documentation potrebný, ale jeho obsah je pre naše účely prázdny.

Pri parsovaní sa najprv vytvorí element Elements, do ktorého sa najprv pridá element Package s názvom diagramu. Ďalej sa postupne pridajú všetky triedy, kde každá trieda predstavuje jeden Element. Atribúty triedy sa ukladajú do elementu Attributes, kde sa do hlavičky attribute uloží jeho identifikačné číslo, názov atribútu a do elementu properties, ktorý je dieťaťom attribute, sa uloží jeho typ.

Metódy triedy sa ukladajú do elementu operations, kde každá metóda je vlastný element Operation. Ten má v hlavičke identifikačné číslo a názov metódy a jeho potomkovia sú type, čo je návratový typ metódy a parameters, čo sú argumenty metódy a ich štruktúra je identická so štruktúrou, ktorá popisuje atribúty triedy.

V elemente Connectors sú popísané všetky relácie medzi našimi triedami. Každá relácia je jeden element typu Connector a skladá sa z elementov source, ktorý obsahuje názov zdrojovej triedy, target, ktorý obsahuje názov cieľovej triedy a taktiež properties, ktoré obsahujú typ relácie a jej smer.

Element diagrams popisuje dodatočné informácie o rozložení diagramu. Na základe identifikačného čísla tried popisuje, kde konkrétne v priestore sa nachádzajú vzhľadom na polohu od ľavého a od horného okraja diagramu. Na záver metóda vráti XML string, ktorý sa d'alej uloží do súboru, ktorý je špecifikovaný prehliadačom súborov.

3.2.2 JSON Parser

Pri parsovaní do súboru JSON sme sa rozhodli využit' alternatívny prístup k ukladaniu a načítavaniu do a zo súborov, a to priamu serializáciu a deserializáciu tried Class, Method, Attribute a Relation reprezentujúce komponenty diagramu, ktoré sa aj pri XMI Parseri priamo načítajú zo súboru.

Na to, aby sme vedeli využit' serializáciu a deserializáciu, nám v jazyku C# stačilo rozšírit' definíciu týchto tried o označenie "[Serializable]". Využitím existujúcich knižníc jazyka C# vie byt' ukladanie do súboru realizované pár riadkami zobrazenými kódom 3.4.

```
public override string SaveDiagram() {
    ParsedEditor.ReverseNodesGeometry();
    var classes = DiagramPool.Instance.ClassDiagram.GetClassList();
    var relations = DiagramPool.Instance.ClassDiagram.GetRelationList();
    var serializedDiagram = JsonConvert.SerializeObject(new { classes, relations });
    ParsedEditor.ReverseNodesGeometry();
    return serializedDiagram;
}
```

Kód 3.4: Serializovanie diagramu do formátu JSON

Kódovanie polohy diagramu v rámci AnimArchu sa mierne líši od polohy zakódovanej v prostredí EnterpriseArchitect. Pri inicializácii diagramu v triede ClassDiagramBuilder sa táto poloha prepočítava. Aby sme nepotrebovali rozlišovat' odkiaľ pochádza diagram a kedy polohy prepočítat' a kedy nie, prepočítame polohy tried priamo pri parsovaní do JSON volaním metódy ReverseNodesGeometry().

Deserializácia týchto tried z objektov je rovnako priamočiara ako ich serializácia. Na rozdiel od niekoľko sto riadkových metód existujúcich pri načítaní diagramu z formátu XMI nám stačia dve krátke metódy zobrazené kódom 3.5.

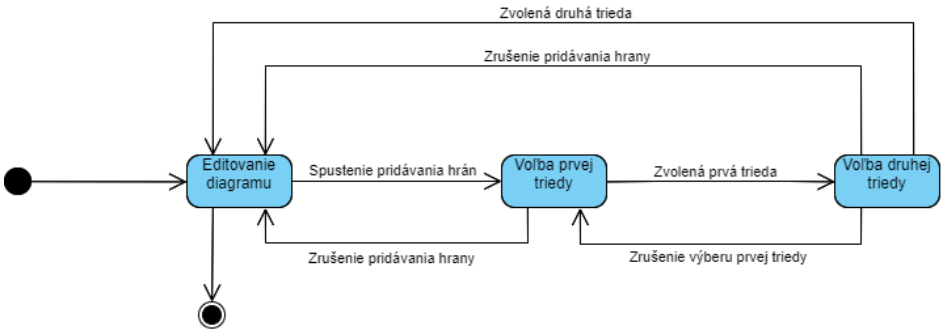
```
public override List<Class> ParseClasses()
{
    var classes = _document["classes"];
    return classes?.ToObject<List<Class>>();
}

public override List<Relation> ParseRelations()
{
    var relations = _document["relations"];
    return relations?.ToObject<List<Relation>>();
}
```

Kód 3.5: Deserializovanie diagramu z formátu JSON

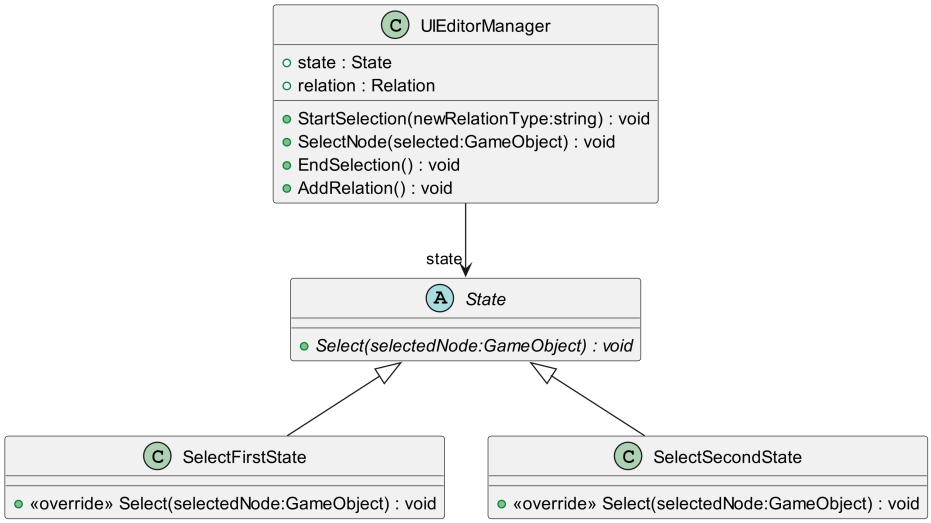
3.3 Pridávanie relácie

Priebeh pridávania relácie, načrtnutý v kapitole 2.1, vieme popísat' stavovým diagramom vyobrazeným na obr. 3.3.



Obr. 3.3: Stavový diagram pridávania relácie

Po prepnutí do režimu editovania diagramu sa nachádzame v stave, keď nepridávame žiadnu hranu. Po spustení pridávania hrany kliknutím na tlačítko *Add relation* sa dostaneme do stavu voľby prvej triedy, ktorú bude nová relácia spájať. Po zvolení prvej triedy kliknutím na ňu sa dostaneme do stavu voľby druhej triedy. Z tohto stavu si vieme opätovným kliknutím na prvú triedu zrušiť výber prvej triedy a tak sa dostať naspäť do stavu výberu prvej triedy. Zvolením druhej triedy sa pridá nová hrana a dostaneme sa naspäť do stavu, keď nepridávame žiadnu hranu. Zo stavu voľba prvej triedy a voľba druhej triedy sa vieme dostať do tohto pôvodného stavu kliknutím na tlačítko *Cancel*, čím ukončíme pridávanie hrany.



Obr. 3.4: Diagram tried obsluhujúcich pridávanie hrany

Na implemetáciu tohto priebehu výberu hrán vieme využiť návrhový vzor State, ktorý takisto, ako je z jeho názvu zrejmé, pracuje so stavmi. Využitím spomínaného návrhového vzoru vieme rozdeliť potencionálne komplikovanú metódu na výber hrán, v ktorej by sme si museli určovať, či sa jedná o výber prvej alebo výber druhej triedy do dvoch samostatných metód, čo takisto zabezpečí ich lepšiu čitateľnosť.

Náš finálny diagram tried, ktoré majú na starosti pridávanie hrany je znázornený na obr. 3.4. Hlavná trieda, ktorá obsluhuje aj iné časti editovania diagramov, sa nazýva UIEditorManager. Trieda State, od ktorej dedia stavy, obsahuje jednu metódu a to Select, ktorá zaobstará spracovanie danej zvolenej triedy. Počas pridávania hrany sú tlačítka na editovanie diagramu

zobrazené, ale nie je možné na ne kliknúť.

Spustenie režimu výberu hrán sa vykoná zavolaním metódy StartSelection triedy UIEditorManager a jej argumentom je typ hrany, ktorú pridávame. V rámci tejto metódy sa inicializuje atribút relation v triede UIEditorManager s daným typom hrany a taktiež sa ako stav určí trieda SelectFirstState. Ďalej je voľba tried sprostredkovávaná metódou SelectNode triedy UIEditorManager. V rámci tejto metódy sa v prípade, že stav nie je null, zavolá metóda Select súčasneho stavu, ako je vidieť aj na kóde 3.6 a ďalšia logika je implementovaná už priamo v prislúchajúcich triedach.

```
public void SelectNode(GameObject selected)
{
    if (!active || state == null)
        return;
    state.Select(selected);
}
```

Kód 3.6: Metóda SelectNode triedy UIEditorManager

Trieda SelectFirstState reprezentuje stav výberu prvej hrany. V rámci metódy Select sa zvýrazní zvolená trieda, uloží jej názov do atribútu relation triedy UIEditorManager a zmení stav na SelectSecondState.

Trieda SelectFirstState reprezentuje stav výberu druhej hrany. V prípade, že sa opätovne zvolí prvá trieda, tak sa zruší jej zvýraznenie, jej názov sa odstráni z atribútu relation a opäť sa zmení stav na SelectFirstState. Ak bola zvolená iná trieda, tiež sa uloží do atribútu relation a zavolá sa metóda AddRelation triedy UIEditorManager. Tá zavolá metódu AddRelation triedy MainEditor v prípade, že pridávaná relácia ešte neexistuje medzi spájanými triedami.

Na záver pridávania relácie sa zavolá metóda EndSelection triedy UIEditorManager, ukázaná na kóde 3.7. Táto metóda sa tiež zavolá v prípade, že používateľ zruší pridávanie hrany. Počas jej priebehu sa zruší zvýraznenie prvej zvolenej triedy v prípade, že bola označená, a taktiež sa nastavia relation a state na null, čím sa efektívne dostaneme naspäť do prvého stavu, kedy nepridávame hranu.

```
public void EndSelection()
{
    SetDiagramButtonsActive(true);
    if (relation.SourceModelName != null)
        Animation.Animation.Instance.HighlightClass(relation.SourceModelName, false);
    relation = null;
    state = null;
}
```

Kód 3.7: Metóda EndSelection triedy UIEditorManager

3.4 Hlboké editovanie názvov tried

Na realizáciu hlbokého editovania názvov tried, potrebujeme zmenit' toto pomenovanie na všetkých miestach, kde sa používa. Priebeh algoritmu preto vieme rozdelit' na tri základné kroky. Prvým krokom je úprava názvu samotnej triedy. Druhým krokom je úprava označenia triedy v relevantných reláciách. Tretím krokom je úprava typov v atribútoch a metódach všetkých tried, kde sa pôvodný názov používa ako typ.

Prvý krok algoritmu je najjednoduchší na realizáciu. V prvom rade sa nájde upravovaná trieda v ClassDiagram a následne sa zmení názov triedy vo všetkých troch reprezentáciách. Vd'aka využitiu referencií v jazyku C# nie je potrebné upravovat' OALProgram, pretože využíva tie isté objekty ako ClassDiagram. Po upravení názvov je potrebné ešte zavolat' metódu UpdateNode triedy VisualEditor, aby sa upravil názov triedy aj v diagrame zobrazenom používateľovi.

Následne sa vykoná druhý krok algoritmu. Na jeho realizáciu potrebujeme získat' všetky relácie z ClassDiagram. Tieto relácie postupne prejdeme a v prípade, že sa pôvodné pomenovanie triedy vyskytovalo ako atribút FromClass alebo ToClass upravíme adekvátne názvy všetkých reprezentácií danej relácie. Vzhl'adom na to, že názov spájaných tried nemá vplyv na ich vizuálnu reprezentáciu, nie je potrebné volat' VisualEditor, aby vykonal prekreslenie relácií.

V tret'om kroku postupne prechádzame všetky triedy. Pre každú triedu si získame relevantné atribúty pomocou funkcie where jazyka C# ako je vidiet' na kóde 3.8.

```
var attributesWithOldNameAsType = new List<Attribute>(otherClassInDiagram.ParsedClass.Attributes
    .Where(x => x.Type.Contains(oldName)));
```

Kód 3.8: Filtrovanie relevantných atribútov

Kontrolu vykonávame takýmto spôsobom, aby sme vedeli získat' aj atribúty, ktorých typom je zoznam pôvodných tried, avšak vieme tak získat' aj falošné zhody. Preto musíme odstránit' kľúčove znaky označujúce, že sa jedná o zoznam, a ak aj v takom prípade nájdeme zhodu, zavoláme metódu UpdateAttribute triedy MainEditor. Tá následne zavolá VisualEditor, ParsedEditor aj CDEditor, aby adekvátnym spôsobom upravila atribút na všetkých miestach v programe.

Obdobným spôsobom sa vykonáva aj úprava atribútov. Tiež si pre každú triedu získame relevantné metódy pomocou funkcie where jazyka C# ako je vidiet' na kóde 3.9. Na rozdiel od hl'adania atribútov potrebujeme taktiež skontrolovat' argumenty daných metód.

```
var methodsWithOldNameAsType = new List<Method>(otherClassInDiagram.ParsedClass.Methods
    .Where(x => x.ReturnValue == oldName ||
        x.arguments.Any(arg => arg.Split(" ")[0].Contains(oldName))));
```

Kód 3.9: Filtrovanie relevantných metód

Pre každú takto získanú metódu potrebujeme rovnako ako pri atribútoch overit', či nešlo o falošnú zhodu pre návratovú hodnotu. Získavanie zhody pre argumenty metód je komplexnej-

```
var attributeType = newMethod.arguments[i].Split(" ")[0];
var attributeName = newMethod.arguments[i].Split(" ")[1];
var formerArray = attributeType.Contains("[]");
attributeType = Regex.Replace(attributeType, "[\\[\\]\\n]", "");
if (attributeType == oldName)
    newMethod.arguments[i] = newName + (formerArray ? "[]" : "") + " " + attributeName;
```

Kód 3.10: Úprava typu argumentu metód

šie, pretože pri Parsed reprezentácií si ich pamätáme ako stringy. Preto je potrebné argument rozdelit' na dve časti, odstránit' kľúčové znaky označujúce zoznam hodnôt, a ak aj v takom prípade je zhoda, nahradíme tento argument ako je ukázané kódom 3.10.

V prípade, že bol nahradený návratový typ metódy alebo niektorý z argumentov metódy, je na záver zavolaná metóda UpdateMethod triedy MainEditor. Tá rovnako ako UpdateAttribute postupne zavolá ParsedEditor, CDEditor a VisualEditor, aby bola metóda upravená na všetkých relevantných miestach.

Záver

Cieľom našej práce bolo obohatiť existujúci prototyp AnimArch o funkcionalitu editovania diagramov v triednej a objektovej vrstve. Bolo potrebné navrhnúť spôsob pridávania tried, atribútov, metód a argumentov funkcií ako aj relácií. Ďalej bolo požadované pridať spôsob ukladania diagramov do súborov. Pridanie novej funkcionality malo byť integrované s ostatnými časťami tohto prototypu spôsobom, ktorý nepoškodí existujúce funkcionality a samotný návrh mal byť dostatočne všeobecný na to, aby prípadné rozšírenie jeho funkcionalít v budúcnosti bolo priamočiare.

Počas implementácie sme si uvedomili, že pridávaná funkcionalita sa bude najmä týkať triednej vrstvy, keďže objektová vrstva je generovaná automaticky na základe OAL skriptu počas priebehu animácie. Naše zmeny do tejto oblasti boli preto minimálne.

Z ostatných požiadaviek sa nám podarilo splniť všetky. Vytvorili sme spôsob pridávania a úpravy diagramu pomocou modálnych okien, zabezpečili ukladanie diagramu do dvoch rôznych formátov a navyše sme obohatili a upravili niektoré existujúce funkcionality, aby bola ich implementácia prehľadnejšia, univerzálnejšia a samotné používanie prototypu hladšie.

Výsledky našej práce už boli zakomponované aj do sieťového prepojenia AnimArchu. Tým sa overilo, že naša implementácia bola robustná, keďže na túto realizáciu bolo potrebné iba minimálne zasahovať do nami napísanej logiky obsluhujúcej úpravu diagramov.

Celkovo je však veľa oblastí, na ktorých sa dá ešte v rámci prototypu pracovať. Jednou zo základných funkcionalít, v ktorej vidíme zlepšenie je rozšírenie AnimArchu o relácie agregácie a kompozície. Rovnako vidíme možnosť rozšírenia triednych diagramov o reprezentáciu rozhraní (interface) ako aj zabezpečenie dedenia metód v prípade, že je medzi dvoma triedami relácia generalizácie. Ďalej je potrebné rozdelenie bočného menu na viacero úrovní, aby bolo prehľadnejšie, úplnejšie a celkovo obsahovalo menej komponentov naraz.

Z dlhodobejšieho hľadiska je v pláne rozšíriť prototyp o sekvenčné diagramy, pridať do sieťovej integrácie animácie a dotiahnuť exportovanie diagramov do XMI formátu tak, aby bolo takto vytvorené súbory možné používať aj v ostatných programoch, ktoré ich podporujú. Z praktického hľadiska vzniká potreba vytvoriť automatizované testy pre existujúce súčasti prototypu, aby sme zaistili jednoduchšiu integráciu nových súčastí v budúcnosti a lepšiu ochranu voči poškodeniu existujúcich modulov, ako aj lepšie riešenie pre priebežné spájanie paralelne vyvíjaných funkcionalít.

Literatúra

[1] Executable UML. [Citované 2023-04-02] Dostupné na https://abstractsolutions.co.uk/our-services/executable-uml/.

[2] XMI Import and Export. [Citované 2023-04-16] Dostupné na https://sparxsystems.com/enterprise_architect_user_guide/14.0/model_publishing/importexport.html.

[3] *XML Metadata Interchange (XMI) Specification, Version 2.5.1.* Object Management Group, Inc, 2015.

[4] *Unified Modeling Language, Version 2.5.1.* Object Management Group, Inc, 2017.

[5] Scott W. Ambler. Approaches to Agile Model Driven Development. [Citované 2023-04-02] Dostupné z http://agilemodeling.com/essays/amddApproaches.htm.

[6] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from uml to alloy. *Software & Systems Modeling*, 9:69–86, 2010.

[7] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of uml models: a systematic review of research and practice. *Software & Systems Modeling*, 18:2313–2360, 2019.

[8] Kirill Fakhroutdinov. UML, Meta Meta Models and Profiles. *[Citované 2023-04-02] Dostupné na* https://www.uml-diagrams.org/uml-meta-models.html, 2017.

[9] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.

[10] Andreas Kästner. Metamodeling with Metamodels Using UML/MOF including OCL. [Citované 2023-04-02] Dostupné na https://www.db.informatik.uni-bremen.de/teaching/courses/ss2020_eis/week10/H-mm.pdf.

[11] Björn Lundell, Brian Lings, Anna Persson, and Anders Mattsson. Uml model interchange in heterogeneous tool environments: An analysis of adoptions of xmi 2. In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*, pages 619–630. Springer, 2006.

[12] Stephen J Mellor and Marc J Balcer. *Executable UML: a foundation for model-driven architecture.* Addison-Wesley Professional, 2002.

[13] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.