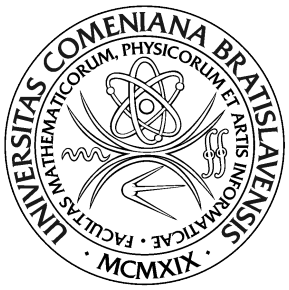
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



VIZUALIZÁCIA SOFTVÉROVÝCH ARCHITEKTÚR A GENEROVANIE ZDROJOVÉHO KÓDU

Diplomová práca

2022 Bc. Filip Novák

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

VIZUALIZÁCIA SOFTVÉROVÝCH ARCHITEKTÚR A GENEROVANIE ZDROJOVÉHO KÓDU

Diplomová práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	doc. Ing. Ivan Polášek, PhD.

Bratislava, 2022

Bc. Filip Novák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

54559536

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Vizualizácia softvérových architektúr a generovanie zdrojového kódu
Visualization of software architectures and source code generation

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 OAL), interaktívnej grafiky v Unity a prototypu animácie UML modelu z roku 2019/2020. Modely je možné vytvárať v CASE nástroji Enterprise Architect a v samotnom prototype, naprogramovanom v C# v spolupráci s rámcom Unity. Platforma Unity umožňuje prácu v 2D ale aj 3D priestore a migrovať aj do virtuálnej (VR) alebo rozšírenej reality (AR). Práca bude súčasťou rozbiehaného výskumu podpory kolaboratívneho modelovania a vizualizácie vo VR/AR priestore.

Navrhnite možnosť vytvorenia úložiska scenárov, architektonických štýlov a vzorov. Doplňte vznikajúci prototyp aj o možnosť generovania zdrojového kódu.

Výstupom DP bude nástroj, obohatený o bázu konkrétnych scenárov, štýlov a vzorov, ktorý by sa dal používať pri modelovaní a vizualizácii softvérovej štruktúry a funkcionality alebo na testovanie funkčnosti a animácie architektonických štýlov a vzorov. Pomohol by aj pri výučbe softvérového inžinierstva na vysvetľovanie modelov, štýlov a vzorov a na podporu experimentovania.

Cieľ:
- Metóda fúzie dynamického a štrukturálneho modelu na vizualizáciu scenáru prípadov použitia a funkcionality architektúry
- Generovanie zdrojového kódu
- Vytvorenie katalógu štýlov a vzorov pre skúmanú metódu
- Vykonanie experimentov s predpripraveným katalógom vzorov na verifikáciu vhodnosti navrhovanej metódy

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.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Buschmann F. et al.: Pattern-oriented software architecture: a pattern language for distributed computing, Vol. 4. New York : John Wiley & Sons, 2007.

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 10.12.2020

Dátum schválenia: 10.12.2020 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a za pomoci konzultácií u môjho školiteľa.

. . . . . .

Bratislava, 2022 Bc. Filip Novák

Poďakovanie

Abstrakt

Táto práca sa venuje vylepšeniu a pridaniu funkcionality do softvéru AnimArch, ktorý slúži na vytváranie diagramov tried a otestovanie funkčnosti navrhovaných softvérov pomocou animácie. Súčasťou tejto práce je prehľad základných termínov súvisiacich súvisiacich s AnimArchom a oboznámenie sa so softvérom samotným. Ďalej tu popisujeme návrh a implementáciu našich cieľov zlepšenie vytvárania animácie pomocou rozdelenia uceleného animačného skriptu do jednotlivých metód pre lepšiu štruktúru a manipulovanie s animáciou. pridanie funkcionality, pomocou ktorej dokážeme generovať Python zo zdrojového kódu animácie. Využili me pritom parser generátor ANTLR. Následne vytvorenie katalógu softvérových štýlov a návrhových vzorov. Hlavným prínosom našej práce je obohatiť spomínaný nástroj AnimArch.

Kľúčové slová: AnimArch, MDD, OAL, parsovanie, ANTLR, xUML, Python

Abstract

Keywords: AnimArch, MDD, OAL, parsing, ANTLR, xUML, Python

Obsah

Úvod **1**

Motivácia **2**

1 Analýza a možnosti vývoja riadeného modelom (MDD) **3**

1.1 Vývoj riadený modelom (MDD) 3

1.2 xUML 4

1.3 OAL 7

1.4 Prototyp AnimArch 11

1.5 Metódy parsovania 17

2 Vylepšenie tvorby animačného skriptu v nástroji AnimArch **19**

3 Generovanie zdrojového kódu aplikácie v jazyku Python **20**

4 Tvorba katalógu softvérových štýlov a návrhových vzorov **21**

5 Výskum a evaluácia **22**

5.1 Výskum využitia jazyka OAL pri generovaní do zdrojového kódu jazyka Python 22

6 Záver **25**

Úvod

Motivácia

Na tejto diplomovej práci ma najviac motivuje to, že jej cieľom je pomôcť ľudom tým, že im jednoducho vysvetlíme zdanlivo možno pre nich ťažkú tému na pochopenie pomocou nášho nástroja AnimArch. Na zložitých diagramoch tried dokážeme ukázať pomocou animácie beh a volanie jednotlivých metód a ich vykonávanie v telách týchto metód, aby ľudia pochopili ako funguje program, ktorý je reprezentovaný diagramom tried alebo aj inými diagramami.

Kapitola 1

Analýza a možnosti vývoja riadeného modelom (MDD)

V tejto kapitole si vysvetlíme a oboznámime sa so základnými pojmami, ktoré sú pre našu prácu dôležité.

1.1 Vývoj riadený modelom (MDD)

Vývoj softvéru je vo všeobecnosti zložitý proces, na ktorý už v dnešnej dobe existuje veľa rôznych metodík vývoja softvéru, pomocou ktorých si ten vývoj dokážeme uľahčiť. Jedným z nich je **Vývoj riadený modelom** (ďalej len MDD z anglického slovného spojenia "Model Driven Development").

MDD je metodika používaná pri vývoji softvéru, ktorá sa zameriava na vytváranie a využívanie doménových modelov. Tieto modely sú potom rôznymi spôsobmi transformované do podoby výsledného softvéru. Hlavnou výhodou tohoto je, že modely sa snažíme vyjadrovať pomocou konceptov, ktoré sú oveľa menej viazané na základnú implementačnú technológiu a zároveň sú oveľa bližšie k problémovej doméne v porovnaní s najpopulárnejšími prog-

ramovacími jazykmi [Sel03]. Pomocou **MDD** zvyšujeme úroveň abstrakcie na ktorej potom vývojári vytvárajú a vyvíjajú softvér, s cieľom zjednodušiť a formalizovať rôzne činnosti a úlohy, ktoré tvoria životný cyklus softvéru [HT06]. Zavádzame pomocou modelov štruktúru.

Základným grafickým jazykom, ktorý je spätý s pojmom **MDD** je **UML** (Unified Modeling Language). Umožňuje nám modelovať rôzne úrovne softvérových abstrakcií. Budeme sa mu venovať v nasledujúcej časti.

Keďže jedným zo zmyslov softvéru **AnimArch** je aby pomáhal pri začiatkoch vývoja akéhokoľvek softvéru tým, že by si v ňom používatelia dokázali vytvoriť modely pomocou diagramov, tak pojem **MDD** je pre nás veľmi významný.

1.2 xUML

Spustiteľný UML (ďalej len xUML z anglického slovného spojenia "Executable Unified Modeling Language") je možné chápať ako vysoko abstraktný jazyk, ale aj ako metodiku vývoja softvéru. Jazyk **xUML** je podmnožina jazyka **UML** a jeho účelom je zadefinovať sémantiku subjektov. Špecifikuje nám súbor pravidiel, pomocou ktorých dávame jednotlivým objektom správanie [MMB02].

Dôležitou časťou **xUML** je *Jazyk akcií* (Action language), ktorému v softvéri **AnimAch** zodpovedá jazyk *OAL*. Mení doteraz statický pohľad modelov tried v **UML** na spustiteľné, vykonateľné modely, ktoré je možné testovať a ladiť. Tieto modely reprezentujú softvérové systémy, ktoré sú nezávislé od konkrétnej technologickej platformy použitej na ich implementáciu. O tomto *jazyku akcie* sa viac dozvieme v samostatnej sekcii.

Aby sme tomu všetkému lepšie pochopili musíme sa najskôr zoznámiť s **UML**.

UML

Jednotný modelovací jazyk (ďalej len UML z anglického slovného spojenia "Unified Modeling Language") je grafický jazyk na vizualizáciu, špecifikáciu, konštrukciu a dokumentáciu softvérových systémov. Môže popisovať aj systémové interakcie [EPLF98]. **UML** využíva na zobrazovanie svojich modelov rôzne typy diagramov.

Diagramy obsahujú grafické prvky usporiadané tak, aby ilustrovali konkrétnu časť systému. Model systému má väčšinou niekoľko diagramov rôznych typov, ktoré závisia od jeho cieľu [EPLF98]. **UML** diagramy v súčasnej dobe rozdeľujeme na dve základné skupiny:

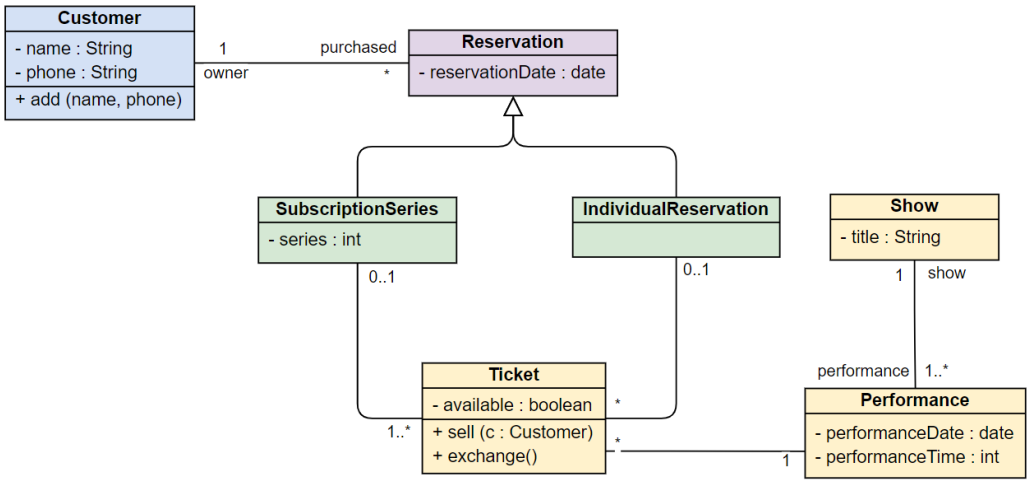
1. ***Štruktúrne diagramy***
    - Diagram tried
    - Diagram komponentov
    - Diagram zloženej štruktúry
    - Diagram nasadenia
    - Diagram balíčkov
    - Diagram objektov
2. ***Diagramy Správania***
    - Diagram aktivít
    - Diagram prípadov použitia
    - Stavový diagram
    - ***Diagramy interakcie***
        - Sekvenčný diagram

- Diagram komunikácie
- Diagram prehľadu interakcií
- Diagram časovania

Diagramy majú veľmi dôležitú vlastnosť a tou je abstrakcia. Každý diagram znázorňuje určitý pohľad na systém, pohľad z určitého uhlu.

Vrámci **AnimArchu** sa najviac zaoberáme konkrétnym typom diagramu a to diagramu tried. Diagram tried popisuje statickú štruktúru tried navrhovaného systému. Triedy sú predstavované ako „veci“, s ktorými sa v systéme narába. Triedy môžu byť navzájom prepojené rôznymi spôsobmi, môžu byť asociované, závislé na iných triedach, špecializované, alebo zabalené (zoskupené ako jednotka). Všetky tieto vzťahy sú znázornené v diagrame triedy spolu s ich vnútornou štruktúrou tried z hľadiska atribútov a operácií (metód) [EPLF98]. Veľké systémy mávajú niekoľko diagramov tried popisujúcich ich štruktúru.

Príklad jednoduchého diagramu tried môžeme vidieť na obrázku 1.1.



Obr. 1.1: Príklad diagramu tried

Znázornené triedy sa skladajú najviac z troch častí: názov, atribúty a operácie (metódy).

Využitie **xUML** metodiky a jej súčastí v **AnimArchu** je významné z hľadiska tvorby diagramov tried a simulovania funkčnosti navrhnutého systému. Konkrétne s pomocou **UML** dokážeme vytvoriť diagramy tried a pomocou jazyka akcie v **xUML** si dokážeme otestovať vytvorený model systému.

1.3 OAL

Jazyk akcií objektov (ďalej len OAL z anglického slovného spojenia "Object Action Language") je platformovo nezávislý vysokoúrovňový programovací jazyk, ktorý využívame ako jazyk akcií v komponente **xUML**. **OAL** sa v mnohých smeroch podobá niektorým programovacím jazykom ako napríklad Java, C++ alebo Python, ale zároveň je jednoduchší ako väčšina programovacích jazykov. Snaží sa byť jednoduchý, preložiteľný, abstraktný a vedomý si modelu ktorého je súčasťou. Jeho zápis je o niečo minimálny, ale je dostatočne bohatý na modelovanie všetkého potrebného [Fac20].

Využitie paltformovo nezávislého jazyka nám poskytuje dobrý základ pre generovanie zdrojové kódu, napríklad do jazyka Python (budeme sa tomu venovať) zo zápisu animácie. Naša verzia jazyka ktorú využívame, pozostáva z podmnožiny pôvodného jazyka **OAL**. Podmnožina bola ešte modifikovaná pre potreby zápisu animácií. Pomocou tohto jazyka, hlavne jeho podmnožiny, budeme vytvárať a zapisovať animácie.

V podmnožine sa nachádzajú:

- **Priradenia** - je možné vytvárať lokálne premenné. Premenné netreba deklarovať. Nie sú explicitne typované, avšak prvým priradením do

premennej je určený aj jej typ. Keď sa pokúsime priradiť hodnoty iného typu do existujúcej premennej, má to za následok chybu. Existujúce typy sú integer (celé číslo), real (reálne číslo), boolean (logická premenná True/False) a string (reťazec). Možné je aj priradenie existujúcej inštancie triedy do špeciálnej premennej (instance handles).

- **Výrazy** - v jazyku je možné zapisovať výrazy, ktorých vyhodnotením vzniká výsledná hodnota. Je možné ich použiť pri priraďovaní alebo riadiacich konštrukciách (*if-elif-else*, *for each-in*, *while*).

- **Dopyty do inštančného priestoru** - určujú manipuláciu s inštanciami tried. V rámci inštančného priestoru je možné vytvárať a odstraňovať inštancie tried, vytvárať a odstraňovať vzťahy medzi nimi, a získavať inštancie tried, ktoré spĺňajú dané podmienky. Získavaním inštancií je možné ich priradiť do už spomínaných špeciálnych premenných, ktoré obsahujú jednu inštanciu triedy alebo kolekciu inštancií triedy.

- **Riadiace konštrukcie** - pomocou nich možno ovplyvňovať tok vykonávania animácie. Medzi riadiace konštrukcie patria *if-elif-else* (podmienka), *for each-in* a *while* (cykly), *break* a *continue* (riadenie cyklov). Riadiace konštrukcie je možné ľubovoľne vnárať medzi sebou, príkazy*break* a *continue* (riadenie cyklov) spôsobujú chybu, ak sa nachádzajú mimo cyklov.

Pre vytvárané animácie pôvodné konštrukcie neboli postačujúce, a tak bol **OAL** v **AnimArchu** modifikovaný tým, že sa rozšíril o nové konštrukcie, ktoré pomohli pri tvorbe animácií.

Nové konštrukcie:

- **Animačný krok** - predstavuje volanie medzi metódami tried a jeho dôsledkom je vizuálny efekt animácie (zvýrazňovanie a vysvecovanie). Neovplyvňuje tok vykonávania.

- **Paralelný blok a vlákno** - pre pôsobivejšiu celkovú animáciu, **AnimArch** umožňuje paralelné toky vykonávania. V paralelnom bloku je možné vytvoriť vlákna (thread), ktoré sú pseudoparalelne vykonávané. Kvôli vizuálnemu efektu sú vlákna v rámci jedného paralelného bloku synchronizované - vlákna sa vykonávajú nezávisle, pri narazení na animačný krok na seba vlákna "počkajú".

Medzi ďalšie nové konštrukcie patria aj nami pridané príkazy za účelom zlepšenia a obohatenie podmnožiny **OAL**. Tieto konštrukcie, ktoré si ukážeme sme pridali vrámci našej diplomovej práce, kvôli z interaktívneniu procesu počas vykonávania animácie a aj kvôli generovaniu Python kódu.

Nové nami pridané konštrukcie:

- **Vytvorenie zoznamu a pridanie prvku** - pre jednoduchšie uchovanie inštancií konkrétnej triedy do špeciálnej premennej. Pri vytváraní zoznamu určujeme jeho typ a pri pridávaní prvku, musí byť ten prvok rovnakého typu ako zoznam. Nie je možné si vytvoriť zoznam typu integer (celé číslo), real (reálne číslo), boolean (logická pre-menná True/False) alebo string (reťazec).

- **Vstup a výstup** - pomocou príkazu pre vstup dokážeme urobiť vykonávanie animácie trochu interaktívne a vyžiadať si od používateľa dáta počas animácie. Príkaz pre výstup môže dať používateľovi informácie o správnosti systému tým, že počas alebo na konci animácie sa vypíšu výsledky.

Príkazy **OAL** sa využívajú v **AnimArchu** pri tvorbe animácií. Príklady základných príkazov si môžeme pozrieť v tabuľke 1.1.

Tabuľka 1.1: Konštrukcie a príklady príkazov v jazyku OAL

Konštrukcie	***Zápis príkazov v OAL***
Priradenie a Výrazy	$z = 1 + 4/(4 + 52)$; $y = x * 25 > 5$; text = "Hello"; Bolt = dogInstance;
Dopyty do inštančného priestoru	create object instance dog1 of Dog; relate dog1 to owner across R5; select many dogs from instances of Dog; unrelate dog1 from owner across R5; delete object instance dog1;
Riadiace konštrukcie	if x > 99 x = 0; else x = x + 1; end if; – for each dog in all_dogs dog.isHungry = FALSE; end for; – while (x < 100); z = z + x; end while;
Animačný krok	call from Dog::Init() to Owner::Register() across R2;
Vytvorenie zoznamu a pridanie prvku	create list dogList of Dog; add dog1 to dogList;

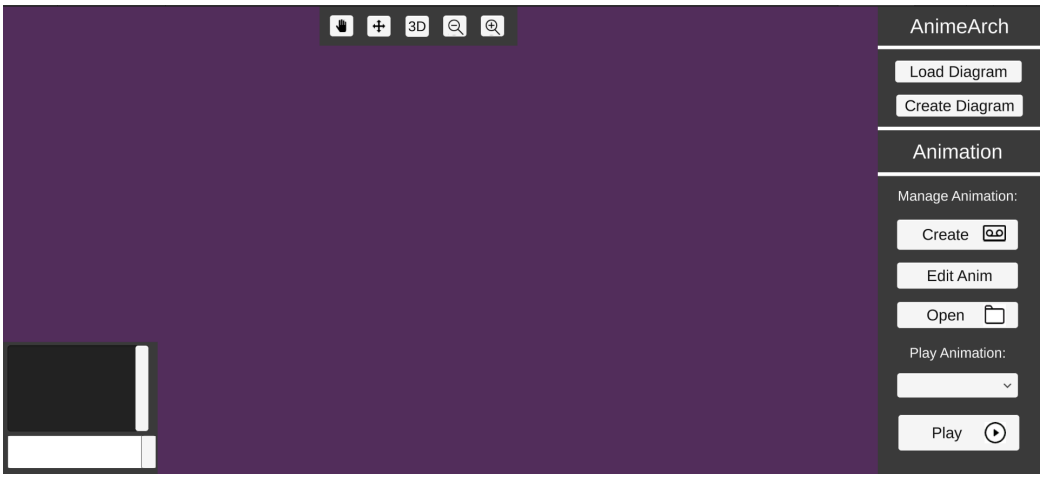
Vstup a výstup	num = int(read(“Write number:”)); write(“Your number is: ”, num);

1.4 Prototyp AnimArch

Cieľom našej diplomovej práce je obohatiť vytváraný nástroj **AnimArch** o nové funkcie. Aby sme sa k nim mohli dostať, najskôr sa s ním musíme oboznámiť a vysvetliť si jeho základné funkcie. V tejto sekcii si aj ukážeme ako vyzerá jeho používateľské prostredie pre lepšiu predstavu.

Nástroj **AnimArch** sa hlavne zameriava na zlepšenie vysvetlenia vybraných architektonických vzorov pomocou animácie ich správania v diagramoch tried. Tento softvér je Unity aplikáciou napísanou v jazyku C#, ktorá sa dá používať aj ako plugin modul softvéru *Enterprise Architect*. *Enterprise Architect* slúži ako nástroj pre kreslenie diagramov tried. Používateľ si môže diagramy vytvárať aj v softvéry **AnimArch** a zobraziť si ich v 3D priestore pomocou spomínaného enginu Unity. Jedna z ďalších funkcionalít je, že používateľ si môže vytvárať scenáre vykonania, ktoré mu pomôžu rýchlo overiť jeho architektúru, bez toho, aby musel vytvárať ďalšie komplikované modely správania. Vytvorenie scenára si vyžaduje iba klikanie na triedy v diagrame tried. **AnimArch** pri tomto klikaní generuje scenár opísaný kódom, ktorý je napísaný v odvodenej verzii jazyka **OAL** (Object Action Language) a navyše je možné ho aj vylepšiť pridaním svojho vlastného kódu.

Na obrázku 1.2 môžeme vidieť jednotlivé funkcie, ktoré tento nástroj ponúka. Detailnejšie si ich vysvetlíme, aby sme lepšie pochopili celý nástroj **AnimArch** a zároveň si ujasníme, ktoré z týchto funkcii budú pre nás dôležité v zmysle našej diplomovej práce.



Obr. 1.2: Používateľské rozhranie nástroja AnimArch

Zoznam základných funkcionalít:

- *Načítanie Diagramu* (Load Diagram)
- *Vytvorenie Diagramu* (Create Diagram)
- *Vytvorenie, Upravovanie a Otvorenie Animácie* (Create, Edit, Open Anim)
- *Prehrať Animáciu* (Play)

Načítanie Diagramu

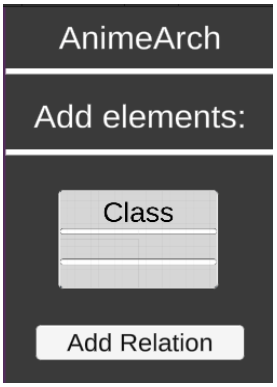
Toto tlačidlo súvisí z vyššie spomínaným softvérom *Enterprise Architect*. Keď chceme toto tlačidlo, použiť musíme mať na našom zariadení uložené diagramy tried, ktoré sú vytvorené softvérom *Enterprise Architect*. Zjednodušene povedané, pomocou tohto softvéru si vytvoríme diagram tried, ktorý uložíme vo formáte XMI. Stlačením tlačidla **Načítanie Diagramu** si vyberieme XMI súbor reprezentovaný diagramom, ktorý chceme načítať, potom

sa tento súbor sparsuje a následne sa vytvorí vizuálna reprezentácia diagramu v **AnimArchu**.

Vytvorenie Diagramu

Diagram tried je statický diagram popisujúci štruktúru navrhovaného systému zobrazením jeho tried a vzťahov medzi nimi.

Pomocou tohto tlačidla si samostatne dokážeme vytvoriť diagram tried v nástroji **AnimArch**. Po rozkliknutí dostaneme na pravej strane menu, ktoré je zobrazené na obrázku 1.3. Používateľ má na výber z dvoch možností:



Obr. 1.3: Menu pri vytváraní diagramu

- *Trieda* (Class)
- *Pridať Vzťah* (Add Relation)

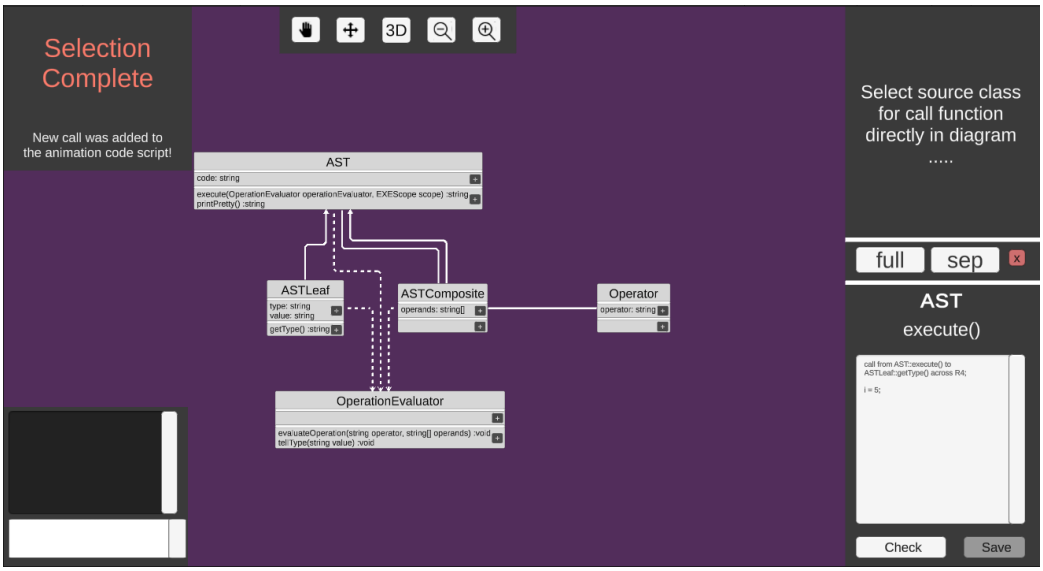
Pomocou obdĺžnika *Trieda* si dokážeme vytvoriť reprezentujúcu triedu navrhovaného systému, ktorá môže obsahovať svoje atribúty a metódy.

Pomocou *Pridať Vzťah* sa umožní používateľovi pridať zobrazeným triedam konkrétny vzťah, ktorý nám ukazuje ako triedy navzájom súvisia v čase vykonávania systému.

Vytvorenie, Editovanie a Otvorenie Animácie

Ak máme vytvorený alebo načítaný diagram a klikneme na tlačidlo vytvorenia animácie, dostaneme sa do stavu znázorneného na obrázku 1.4.

V strede vidíme zobrazený diagram tried, na ktorom chceme vytvoriť animáciu, konkrétne animačný skript. Tvorba tohto skriptu je založená na



Obr. 1.4: Rozhranie pre vytvárania animácie

priamom klikaní jednotlivých tried diagramu. Kliknutím na konkrétnu triedu a následne na nejakú jej metódu si určíme začiatok animácie a ďalším kliknutím na nejakú inú triedu a jej metódu si určíme kam sa chceme dostať. Po týchto úspešne zvládnutých krokoch sa nám automaticky do textového poľa na pravej strane zapíše zdrojový kód reprezentujúci volanie vybratej nami začiatočnej metódy do nami zvolenej konečnej metódy. Tento zdrojový kód je zapísaný v modifikovanej verzii jazyka **OAL**. Na obrázku 1.4 si môžeme všimnúť ze tento vygenerovaný kód reprezentujúci animáciu sa nachádza v tele metódy, ktorej meno je napísané nad textovým poľom spolu s aj menom triedy do ktorej patrí. Následne ak chceme tak môžeme vytvárať ďalšie volania pomocou klikov alebo si môžeme doplniť do textového poľa, ktoré znázorňuje konkrétne telo metódy svoj vlastnoručne napísaný kód v jazyku **OAL**, pomocou ktorého si odkážeme odsimulovať beh nášho systému. Takto vytvorenú animáciu reprezentovaná zdrojovým kódom v animačnom skripte si môžeme uložiť vo formáte JSON.

Konkrétne tvorba animácie je pre našu diplomovú prácu dôležitá pretože sa jej aj budeme trochu venovať.

Editovanie animácie znamená, že môžeme upraviť alebo zmeniť zdrojový kód animačného skriptu. Po kliknutí na toto tlačidlo sa nám zobrazí rozhranie na obrázku 1.4, lenže textové polia niektorých metód budú vyplnené zdrojovým kódom na základe načítanej animácie. Aby sme mohli túto funkciu použiť potrebujeme mať aktuálne vytvorenú alebo načítanú animáciu zo súboru, ktorá je uložená v našom systéme.

Otvorenie animácie znamená, že si môžeme načítať do **AnimArchu** uloženú animáciu vo formáte JSON. Na obrázku 1.2 sa nám potom meno tejto animácie zobrazí v časti "Play Animation" v bielom poli hneď po týmto názvom.

Prehrať Animáciu

Ak máme vytvorený alebo načítaný diagram tried a k nemu vytvorenú alebo načítanú animáciu, po spustení tlačidla "Play" (*Prehrať Animáciu*) sa nám zobrazí používateľské rozhranie s menu na ľavo vyobrazené na obrázku 1.5, aj s konkrétnym diagramom tried.

Pod pojmom animácie si v našom nástroji predstavujeme zvýrazňovanie textu a vysvecovanie tried, metód spolu s ich hranami na základe volaní metód napísaných v jazyku **OAL** v zdrojovom kóde. Príklad takejto animácie je možné pozorovať na obrázku 1.5, kde vidíme že z triedy *AST*, jej metódy *execute()* voláme metódu *getType()* nachádzajúcu sa v triede *ASTLeaf*.

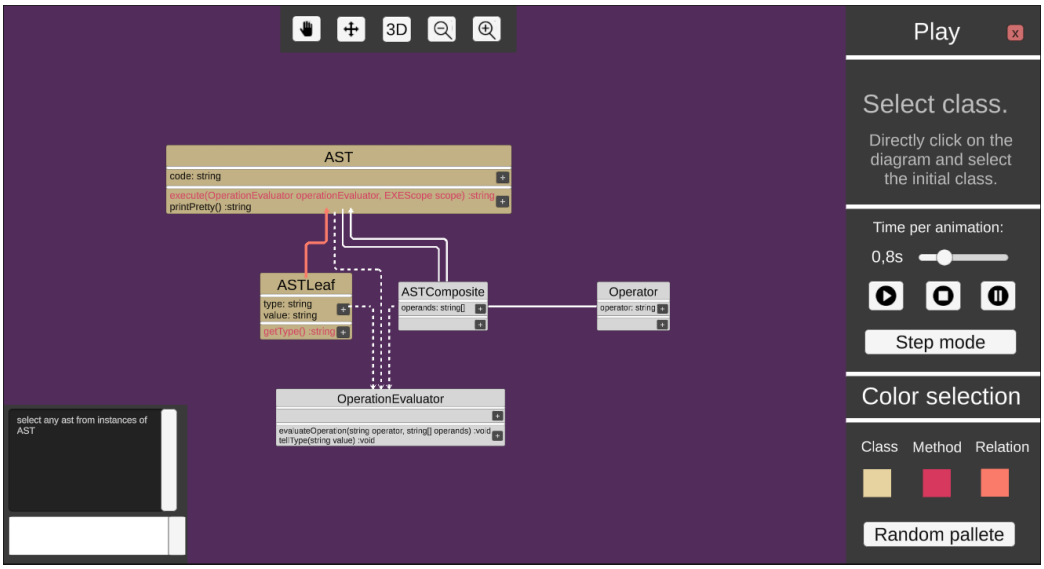
Spôsob akým spustíme našu animáciu na diagrame tried je nasledovný. Klikneme na konkrétnu triedu a na pravej strane pod názvom "Play" sa zobrazia mená metód z ktorých môže animácia začínať. Musíme si ale uvedomiť,

že sa tam zobrazia len tie názvy metód vybratej triedy, ktoré majú v sebe zdrojový kód, ktorý sme im dali pri vytváraní animácie. Používateľ si jednu vyberie kliknutím a následne po stlačení tlačidla *Play* (zobrazeným pomocou šipky), ktoré sa nachádza v strednej časti pravého menu sa animácia spustí. Ak sme použili pri tvorbe animácie v zdrojovom kóde príkazy na zadanie vstupu alebo na zapisovanie výstupu, tak je pre nás významné aj okienko nachádzajúce sa v ľavom dolnom rohu na obrázku 1.5. Pre napísanie vstupných údajov môže používateľ využiť spodnú bielu konzolu a ak sa má niečo počas animácie simulujúcej beh programu vypísať, tak sa to zobrazí v čiernej konzole, ktorá slúži len na čítanie.

Takýmto jednoduchým spôsobom pomocou animácie si môžeme odsimulovať navrhovaný softvér bez toho aby sme ho museli celý naprogramovať od základov.

V menu máme viacero spôsobov ovládanie animácie, ktorá je spustené. Môžeme zmeniť jej rýchlosť, ďalej ju môžeme zastaviť, pozastaviť alebo sa presunúť do krokovacieho režimu (Step mode). V tomto režime je možné si animáciu odkrokovať pomocou dvoch tlačidiel znázorňujúcich šipky doľava a doprava. Jeden takýto animačný krok znamená vysvietenie objektu alebo zvýraznenie textu. Poslednou možnosťou ktorú mažeme použiť v tomto menu je náhodné zmenenie palety farieb, ktorá nám určuje jednotlivé farby triedy, metódy a ich vzťahu (hrany) v animácii.

Jedným z cieľov našej diplomovej práce je pridanie novej funkcionality (nové tlačidlo v menu, ktoré sa bude nachádzať na obrázku 1.2) a tou je generovanie Python kódu zo zdrojového kódu jazyka **OAL**. Bližšie si o tejto funkcii povieme v samostatnej kapitole.



Obr. 1.5: Rozhranie pre prehratie animácie

1.5 Metódy parsovania

Táto časť je hlavne zameraná na analýzu základných metód parsovania. Parser alebo syntaktický analyzátor číta postupnosť tokenov generovaných lexikálnym analyzátorom a konštruuje syntaktický strom. Tokeny generované lexikálnym analyzátorom sú v programovacích jazykoch väčšinou identifikátory, čísla, reťazce, operátory a tak podobne. Úlohou parsera je overovať, či vstupný reťazec môže byť generovaný zadanou bezkontextovou gramatikou pre vstupný jazyk. Upozorňuje na syntaktické chyby v danom kóde. Parsovacie algoritmy sú dôležitým nástrojom pri analýze počítačových jazykov.

Rôzne parsovacie metódy majú často spoločné vlastnosti. Jednou z týchto charakteristík je prístup metódy pri konštrukcii syntaktického stromu. Parsovacie metódy delíme na dva typy [Thu07]:

- **Prístup zhora dole**
- **Prístup zdola hore**

Metóda zhora nadol sa pokúša porovnať vstupný reťazec s gramatikou tým, že zvažuje odvodenia gramatického neterminálu. Kladie si otázku, ktoré pravé strany (z ktorých môžeme niečo vyprodukovať) možno odvodiť od tejto ľavej strany. Začína cieľovým symbolom a prechádza nadol k vrcholu listu. Metóda zdola nahor začína vrcholmi listov a zvažuje redukcie na neterminály. Pýta sa, na aké neterminály sa tieto prvky na pravej strane dajú zredukovať. Postupne sa pohybuje od vrcholov listov smerom nahor k cieľovému symbolu [Thu07].

Kapitola 2

Vylepšenie tvorby animačného skriptu v nástroji AnimArch

V tejto kapitole si objasníme ako sme vylepšili tvorbu animačného skriptu tým, že sme hlavný skript rozdelili do menších animačných skriptov na základe jednotlivých metód.

Kapitola 3

Generovanie zdrojového kódu aplikácie v jazyku Python

V tejto časti sa zameriame hlavne na implementáciu parsera, pomocou ktorého dokážeme z jazyka OAL vygenerovať výsledný Python kód.

Kapitola 4

Tvorba katalógu softvérových štýlov a návrhových vzorov

Táto kapitola sa zameria na tvorbu a prehľad katalógu rôznych softvérových štýlov a návrhových vzorov, ktoré budú použité v nástroji AnimArch ako šablóny.

Kapitola 5

Výskum a evaluácia

Táto kapitola sa zameria na náš výskum a zhodnotenie dosiahnutých výsledkov.

Snažíme sa skúmať možnosti využitia jazyka **xUML**, konkrétne v našom prípade komponent jazyk akcií **OAL** vrámci nášho prototypu **AnimArch**. Kladieme si otázku, či je jazyk **OAL** schopný prepojiť štrukturálny (statický) a dynamický model. Zameriavame sa aj na využite jazyka **OAL** pri generovaní do zdrojového kódu jazyka Python.

5.1 Výskum využitia jazyka OAL pri generovaní do zdrojového kódu jazyka Python

Pre lepšie pochopenie si predstavíme jednoduchý príklad generovania zdrojového kódu do Pythonu. Na obrázku 5.1 majme zobrazený kód v ktorom sa nachádza stručný prehľad zápisu tried spolu s ich metódami vo forme pseudokódu. Konkrétne kód nachádzajúci sa v telách metód je zapísaný v jazyku **OAL**.

```
class Veterinarian
        create list registered_dogs of Dog;

        method registerDog(dog)
                add dog to registered_dogs;
        end method;

        method chipDogs()
                for  each dog in registered_dogs
                        dog.update();
                end for;
         end method;
end class;

class Dog
        chipped = FALSE;

        method update()
                chipped = TRUE;
        end method;
end class;

class Owner
        PetDog = UNDEFINED;

        method buyPetDog()
                create object instance PetDog of Dog;
        end method;

        method registerPetDog(vet)
                vet.registerDog(PetDog);
        end method;
end class;


method main
        create object instance o of Owner;
        create object instance vet of Veterinarian;
        o.buyPetDog();
        o.registerPetDog(vet);
        vet.chipDogs();
end method;
```

Obr. 5.1: Príklad čipovania psíkov v jazyku **OAL**

Zo zobrazeného **OAL** kódu chceme vygenerovať zdrojový kód v jazyku Python, ktorý môžeme vidieť na obrázku 5.2.

```
class Veterinarian:
        def __init__(self):
                self.registered_dogs = None

        def registerDog(self, dog):
                self.registered_dogs.append(dog)

        def chipDogs(self):
                for dog in self.registered_dogs:
                        dog.update()

class Dog:
        def __init__(self):
                self.chipped = False

        def update(self):
                self.chipped = True

class Owner:
        def __init__(self):
                self.PetDog = None

        def buyPetDog(self):
                self.PetDog = Dog()

        def registerPetDog(self, vet):
                vet.registerDog(self.PetDog)


o = Owner()
vet = Veterinarian()
o.buyPetDog()
o.registerPetDog(vet)
vet.chipDogs()
```

Obr. 5.2: Príklad čipovania psíkov v jazyku Python

Kapitola 6

Záver

Literatúra

[EPLF98] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML toolkit*, volume 1. Wiley New York, 1998.

[Fac20] One Fact. Action language (oal) tutorial. `https://xtuml.org/learn/action-language-tutorial/`, Aug 2020. [Online, accessed 6-December-2021].

[FPV17] Matej Ferenc, Ivan Polasek, and Juraj Vincúr. Collaborative modeling and visualization of software systems using multidimensional uml. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 99–103. IEEE, 2017.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1. edition, 1994.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*, pages 1–39. World Scientific, 1993.

[HT06] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.

[MMB02] Stephen J Mellor, Stephen Mellor, and Marc J Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley Professional, 2002.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[TGF+05] Trung Dinh Trong, Sudipto Ghosh, Robert B. France, Michael Hamilton, and Brent Wilkins. Umlant: An eclipse plugin for animating and testing uml designs. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '05, pages 120–124. Association for Computing Machinery, 2005.

[Thu07] Adrian Thurston. Generalized parsing techniques for computer languages. Technical report, School of Computing, Queen's University, 2007.

Zoznam obrázkov

1.1 Príklad diagramu tried 6
1.2 Používateľské rozhranie nástroja AnimArch 12
1.3 Menu pri vytváraní diagramu 13
1.4 Rozhranie pre vytvárania animácie 14
1.5 Rozhranie pre prehratie animácie 17

5.1 Príklad čipovania psíkov v jazyku **OAL** 23
5.2 Príklad čipovania psíkov v jazyku Python 24

Zoznam tabuliek

1.1 Konštrukcie a príklady príkazov v jazyku OAL 10