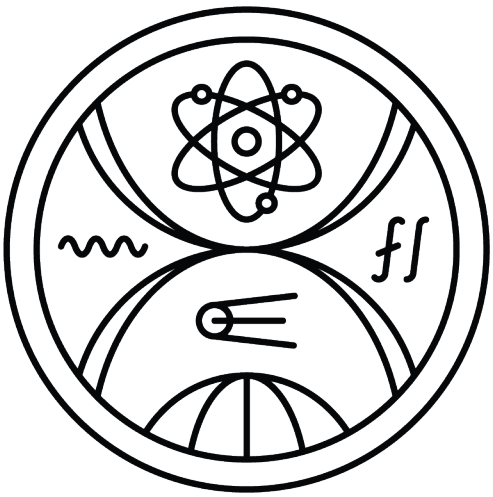
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



NÁVRH SOFTVÉROVÝCH RÁMCOV POMOCOU MDD

DIPLOMOVÁ PRÁCA

2023
BC. ONDREJ RICHNÁK

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

NÁVRH SOFTVÉROVÝCH RÁMCOV POMOCOU MDD

DIPLOMOVÁ PRÁCA

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

Bratislava, 2023
Bc. Ondrej Richnák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Ondrej Richnák
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: Návrh softvérových rámcov pomocou MDD
Design of software frameworks using MDD

Anotácia: Zložitosť vývoja rozsiahlych softvérových systémov nás núti k výskumu a pokusom zaviesť do oblasti softvérového inžinierstva nové metódy modelovania a implementácie, ktoré by podporili ľahšie porozumenie, rozširovanie a znovupoužitie funkcionality a softvérových znalostí v zdrojovom kóde.
Jednou z možností je vytvoriť všeobecné softvérové rámce, ktoré by podporili rýchlejší vývoj pomocou Model Driven Development (MDD).

Cieľ: Cieľom práce je návrh a implementácia dvoch softvérových rámcov v diagrame tried xUML a v jazyku OAL a test jeho funkčnosti vo vygenerovanom zdrojovom kóde.
Výstup DP obohatí katalóg softvérových štýlov a vzorov, ktoré by bolo možné použiť pri modelovaní softvérovej štruktúry na overenie a testovanie funkčnosti vyvíjaného systému.
Pomohlo by tiež pri výučbe softvérového inžinierstva vysvetliť modely, štýly a vzory a podporiť experimentovanie.
Ako prípadovú štúdiu vytvorte softvérový rámec pomocou architektonického štýlu Blackboard alebo Pipes and Filters.
Overte na jednoduchom expertnom systéme jeho použiteľnosť.

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.

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

Yigitbas, E., Gorissen, S., Weidmann, N. et al. Design and evaluation of a collaborative UML modeling environment in virtual reality. Journal on Software and Systems Modeling, Springer 2022

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

Poďakovanie: Rád by som poďakoval môjmu školiteľovi doc. Ing. Ivan Polášek, PhD., za vedenie práce, cenné rady, konzultácie a množstvo trpezlivosti a motivácie pri tvorbe diplomovej práce. Taktiež by som chcel srdečne poďakovať Ing. Lukášovi Radovskému a mojej priateľke. Bez ich podpory a povzbudzovania by táto diplomová práca nebola možná.

Abstrakt

V diplomovej práci sme sa venovali novým metódam modelovania a implementácie architektonických štýlov. Pričom hlavnou náplňou práce bolo vytvorenie spôsobu na reprezentáciu dvoch zvolených štýlov, a to „Pipes and Filters“ a „BlackBoard“. Pri tvorbe týchto architektonických štýlov sme sa zamerali na praktickú reprezentáciu a jej jednoduchosť s cieľom zabezpečiť pochopenie zo strany širokej verejnosti, nie len zúženému okruhu ľudí s pokročilým vzdelaním v oblasti architektonických štýlov.

V štýle „Pipes and Filters“ sme sa pozreli na jednoduchší príklad „Fibonacciho postupnosť“, v ktorom sme sa snažili zachytiť všetky podstatné časti štýlu, ako sú filtre, potrubia, ktoré slúžia na posielanie dát medzi filtrami. V prípade filtrov sme sa zamerali na lineárne spracovanie. Taktiež sme chceli ukázať všetky možné využitia filtrov ako možnosti transformácie, zlučovania a delenia.

Následne sme spracovali štýl „Blackboard“, v ktorom sme sa pokúsili implementovať o niečo zložitejší príklad „Expertného paralelného medicínskeho systému s pravidlami na detekciu chorôb“.

V práci sa dozvieme bližšie informácie o vývojovom procese v softvéri AnimArch, ako aj o úskaliach spojených s vývojom samotným, problémoch a riešeniach, na ktoré sme prišli a ktorými sme vylepšili softvér AnimArch.

Kľúčové slová: AnimArch, Pipes and Filters, Blackboard, MDD, Python

Abstract

In our thesis, we focused on new methods of modeling and implementing architectural styles. The main objective of the work was to create a way to represent two selected styles, namely „Pipes and Filters“ and „BlackBoard“. In developing these architectural styles, we focused on practical representation and simplicity to ensure understanding by a wide audience, not just a narrow circle of people with advanced education in architectural styles.

In the „Pipes and Filters“ style, we examined a simpler example, the „Fibonacci sequence“, in which we attempted to capture all essential parts of the style, such as filters, pipes used for data transmission between filters. Regarding filters, we focused on linear processing. We also aimed to demonstrate all possible uses of filters, such as transformation, merging, and splitting.

Subsequently, we elaborated on the „Blackboard“ style, where we attempted to implement a somewhat more complex example of an „Expert parallel medical system with rules for disease detection“.

The thesis provides detailed information on the development process in the AnimArch software, as well as the challenges associated with development itself, the problems encountered, and the solutions we devised to enhance the AnimArch software.

Keywords: AnimArch, Pipes and Filters, Blackboard, MDD, Python

Obsah

Zoznam skratiek

skratky	anglicky	slovensky
BB	BlackBoard	Tabuľa
P&F	Pipes and Filter	Dátovody a Filtre
MD	Model Driven	Modelom riadený
MDD	Model Driven Development	Vývoj riadený modelom
MDE	Model Driven Engineering	Modelovo riadené inžinierstvo
MDT	Model Driven Testing	Modelovo riadené testovanie
MDA	Model Driven Architecture	Modelovo riadená architektúra
UML	Unified Modeling Language	Unifikovaný modelovací jazyk
xUML	Executable Unified Modeling Language	Vykonávateľný unifikovaný modelovací Jazyk
OAL	Object Action Language	Jazyk akcií objektov
EA	Enterprise Architect	Enterprise Architect
UI	User Interface	Používateľské rozhranie
KS	Knowledge Source	Zdroj vedomostí

Zoznam obrázkov

Úvod

V modernom svete softvérového inžinierstva je modelovanie architektúr a vizualizovanie softvérových systémov kritickým krokom v ich vývoji. Jedným z kľúčových nástrojov na dosiahnutie tejto vizualizácie je rozšírený Unified Modeling Language (xUML), ktorý poskytuje štandardizovaný jazyk (OAL) na popis štruktúry a správania.

V rámci tejto práce budeme detailne popisovať iteratívny postup pri implementácii jednotlivých rámcov, analyzovať identifikované problémy a zároveň overíme funkčnosť generácie kódu v jazyku Python. Týmto spôsobom sa zameriame na praktickú stránku našej práce, kde sa budeme venovať konkrétnym krokom a rozhodnutiam pri implementácii rámca **Pipes and Filters** a **Black board** v kontexte modelovania architektúr s využitím xUML.

Cieľom tejto diplomovej práce je prehĺbiť naše pochopenie modelovania architektúr pomocou xUML a skúmať možnosti implementácie rámcov **Pipes and Filters** a **Black board** v kontexte softvérového návrhu. Tým sa snažíme dosiahnuť nielen lepšiu štruktúru systémov, ale aj ich spôsob implementácie, a hlavne celkové porozumenie. Prácu zakončíme otestovaním jednotlivých rámcov, podla vhodne zvolenej techniky. Pre oba štýly **Pipes and Filters** a **Black board** to bude kvalitativne ohodnotenie relevancie pomocou standardu pre vyvoj (RJT).

1 Analýza metód softvérového inžinierstva a použitých nástrojov

1.1 Základná terminológia

V tejto časti kapitole si podrobnejšie vysvetlíme a oboznámime sa so základnými pojmy, ktoré sú kľúčové pre našu prácu.

1.1.1 Modelovo orientované prístupy/ Prístupy Modelovej Orientácie

V tejto časti sa zameriame na celkovú abstrakciu rôznych rámcov MD (Model-Driven - modelom riadený) prístupov vývoja.

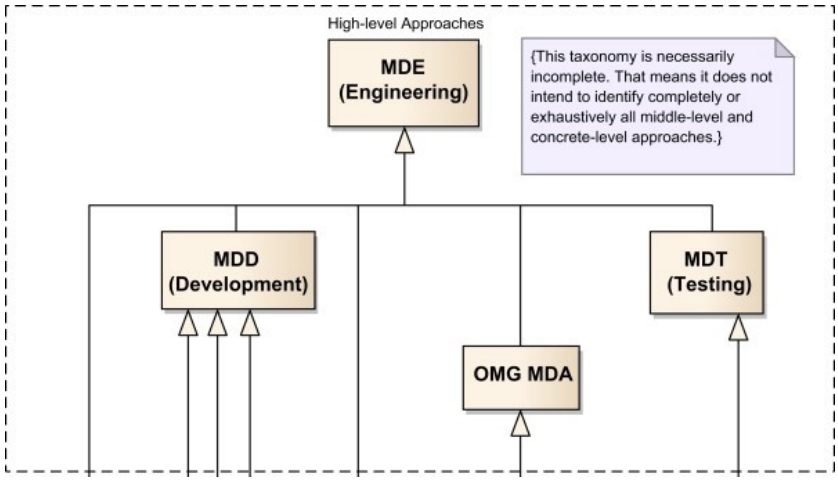
MDE - Model Driven Engineering / Modelovo riadené inžinierstvo

Kľúčové aspekty **MDE** zahŕňajú vytváranie a správu modelov ako centrálneho artefaktu v procese vývoja softvéru. To zahŕňa vývoj a integráciu rôznych nástrojov, klasifikovaných ako integrované vývojové prostredia (IDE), nástroje na počítačom podporované inžinierstvo softvéru (CASE) a MetaCASE nástroje. Tieto nástroje si kladú za cieľ podporovať efektívnu implementáciu **MDE** uľahčením vytvárania modelovacích jazykov, poskytovaním prostredia pre návrh a správu modelov, umožňujúca overovania a analýzu modelov, podporovaním transformácií model-model

a model-text a v niektorých prípadoch umožnenie interpretácie a vykonávania modelov.

V priebehu posledného desaťročia bolo vyvinutých niekoľko nástrojov, ktoré inkorporujú princípy **MDE**. Medzi akademickými nástrojmi patria ProjectIT, MetaSketch, AtomPM, ... zatiaľ čo komerčné nástroje zahŕňajú Microsoft Visual Studio Visualization and Modeling SDK, Enterprise Architect a mnohé ďalšie.

Preukázaný prieskum o **MDE**, ktorý je nájdeme v texte[24], je výsledkom výskumu a poskytuje jednotný konceptuálny model pre celkové porozumenie **MDE** a jeho kľúčových konceptov. Tento koncep definuje základné pojmy ako sú: systém, model, meta-model a modelovací jazyk, zahŕňajúc abstraktnú a konkrétnu syntax, sémantiku, transformácie modelov a mnohé daľšie. Najvišiu vrstvu abstrakcie je možné vidieť na obr. 1.1.



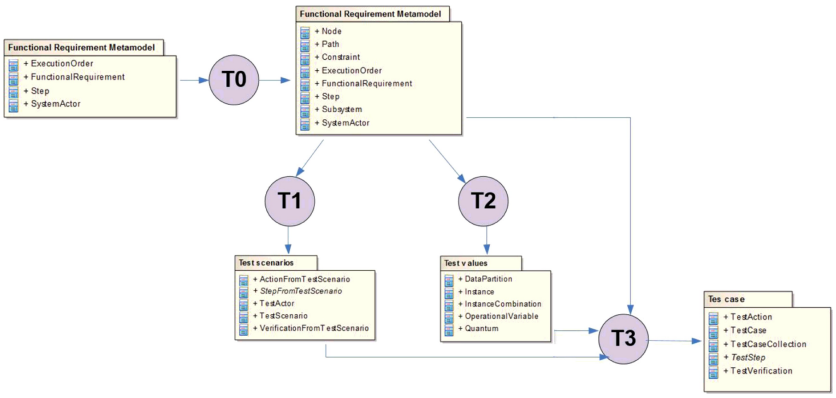
Obr. 1.1: najvyššia abstrakcia MD [24]

MDT - Model Driven Testing / Modelovo riadené testovanie

Model-Driven Testing (MDT) alebo modelovo riadené testovanie zahŕňa vytváranie modelov, ktoré zameriavajú na správanie softvérového systému. Tieto modely slúžia sú komplexov reprezentáciou požiadaviek, špecifikácií a dizajnových prv-

kov. Ponúkajú nám jasný a štruktúrovaný pohľad na plánovanú funkcionalitu systému. Na rozdiel od tradičných testovacích metodológií, ktoré pozostávajú hlavne z manuálneho vytvárania testovacích prípadov a scenárov. **MDT** nám zavádza automatizáciu do samotného srdca systému.

Hlavnou výhodou **MDT** je jeho schopnosti automaticky generovať testovacie prípady z modelov. Dôsledkom tejto automatizovanej generácia testovacích prípadov je zabezpečenie systematickejší a konzistentnejší testov. S využitím iných MD prístupov, **MDT** nielenže urýchľuje testovací proces, ale minimalizuje aj riziko ľudských chýb [11] príklad implementácie meta modelu 1.2.



Obr. 1.2: meta model pre MDT[11]

MDD - Model Driven Development / Vývoj riadený modelom

Modelom riadený vývoj (MDD) je metodika pre vývoj softvéru, ktorá umožňuje používateľom vytvárať sofistikované aplikácie prostredníctvom zjednodušených abstrakcií s využitim vopred definovaných komponentov reprezentujúce obchodné potreby a riešenia technických problémov,ktoré sú zakotvené v modeloch pred generovaním kódu[12].

Predovšetkým v programoch umožňujúcich generovanie kódu z triednych diagramov, je analýza modelovo orientovaného vývoja nevyhnutná. Proces začína

návrhom v rámci UML. Následne nám pokytuje kostru pre generovanie kódu, čím priamo podporuje vývojový.

Vďaka abstrakcii modelov,dokážeme porozumieť komplexným problémom. Vzhľadom na obrovskú zložitosť softvérových systémov sú modely vhodné pre tento druh vývoja. MDD má vo svojom jadre potenciál v zjednodušovania práce a automatizovaní istých úloh v softvérovom vývoji.

Jedna z hlavných podstát MDD je v tom, že vývoj softvéru primárne zahŕňa vyjadrovanie nápadov a konceptov namiesto samotných programových iplemnetaciji. Čím umožňuje údržbu nezávislých modelov bez ohľadu na platformu[9].

Úplný potenciál MDD spočíva v závere, teda automatickom generovaní kompletného a overiteľného kódu z modelu, pričom štandardy UML určujú najlepšie postupy, pre kompatibilitu a znovu použiteľnosť častí systémov.

MDA - Model Driven Architecture / Modelovo riadená architektúra

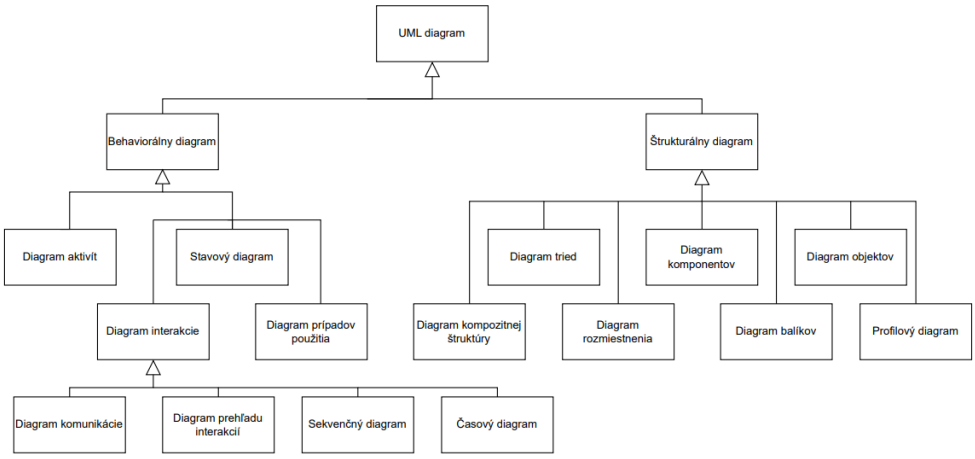
Model Driven Architecture (MDA) je rámec pre vývoj softvéru, definovaný skupinou Object Management Group (OMG). Kľúčovým aspektom MDA je dôležitosť modelov v procese vývoja softvéru. V rámci MDA je vývoj softvéru riadený činnosťou modelovania softvérového systému.

Cyklus tradičného vývoja v MDA sa líši hlavne povahov artefaktov vytváraných počas procesu vývoja. Týmito artefaktami sú formálne modely, teda modely, ktoré môže pochopiť počítač. Jedným z týchto modelov je Platform Independent Model (PIM).

PIM je model s vysokou úrovňou abstrakcie, nezávislý na akejkoľvek implementačnej technológii. Tento model popisuje softvérový systém bez ohľadu na konkrétnu technológiu implementácie. V PIM je systém modelovaný z hľadiska obchodných procesov. Napríklad, či bude systém implementovaný na mainframu s relačnou databázou alebo na aplikačnom serveri, nie je dôležité v PIM. Hlavným cieľom PIM je poskytnúť abstraktný pohľad na softvérový systém[15].

1.1.2 UML - Unified Modeling Language / Unifikovaný modelovací jazyk

Unified Modeling Language (skrátene „UML“) predstavuje univerzálny jazyk vhodný na vizuálne modelovanie. Slúži na špecifikáciu, vizualizáciu, konštrukciu a dokumentáciu artefaktov softvérového systému, pričom je využívaný na pochopenie, dizajn, prezeranie, konfiguráciu, udržiavanie a kontrolu informácií o danom systéme[22].UML umožňuje vytvárať rôzne typy diagramov, ktoré sú rozdelené do dvoch hlavných kategórií: štrukturálne a behaviorálne[11], pre úplné pochopenie členenia sa môžme pozrieť na obr.1.3.



Obr. 1.3: členenie diagramov podľa druhu správania [5]

Behaviorálne / Diagramy znázorňujúce chovanie

Zobrazujú dynamické správanie objektov systému v čase.

Diagramy znázorňujúce štruktúru

reprezentujú statickú štruktúru systému a jeho častí. Zachytávajú interakcie na rôznych úrovniach abstrakcie čim umožňujú detailnú reprezentáciu štruktúr navrhnutého systému. Táto reprezentácia pozostáva s tried, rozhraní a vzájomných vzťahov, ktoré sú asociácia, generalizácia a závislosť.

1.1.3 xUML- Executable Unified Modeling Language / Vykonávateľný unifikovaný modelovací jazyk

Spustiteľný UML (známy aj ako xUML) môžeme chápať ako vysoko abstraktný jazyk a metodiku vývoja softvéru. Predstavuje pracovanie na ďalšej, vyššej úrovni abstrakcie, ktorá abstrahuje konkrétne programovacie jazyky a rozhodnutia o organizácii softvéru. V praxi to znamená, že špecifikácia vytvorená v xUML môže byť využívaná v rôznych softvérových prostrediach bez potreby úprav. To umožňuje využívať xUML pri tvorbe uz spominaneho modelu PIM (Platform Independent Model). xUML tiež definuje súbor pravidiel, ktoré určujú správanie jednotlivých objektov[18].

xUML podporuje modelovanie údajov, ich spracovanie a riadenie prostredníctvom grafických diagramov, ako sú diagramy komponentov, tried a stavové diagramy. Modely v xUML sú spustiteľné, umožňujú testovanie, ladenie a meranie výkonu. Spustiteľný UML podporuje modelom riadenú architektúru (MDA) prostredníctvom špecifikácie modelov nezávislých od platformy[16].

Na koniec treba dodať, že xtUML (Executable and Translatable UML / Vykonávateľný a prekladateľný unifikovaný modelovací jazyk) predstavuje metodológiu a súbor nástrojov používaných pri vývoji riadenom modelom. Táto metodika vychádza z princípov xUML, čo umožňuje vytvárať modely softvérových systémov, ktoré možno priamo spustiť alebo preložiť do vykonateľného kódu. xtUML je zatiaľ poslednou významnou evolúciou UML, ktorá tiež umožňuje súčasne používanie ako xUML, ale zároveň poskytuje možnosť kompilácie do konkrétneho programovacieho jazyka[26][4].

1.1.4 OAL - Object Action Language / Jazyk akcií objektov

OAL (Object Action Language), známy aj ako Jazyk akcií objektov, predstavuje platformou nezávislý vysoko-úrovňový programovací jazyk používaný ako jazyk akcií v rámci metodológie xUML. OAL sa v mnohých aspektoch podobá niektorým programovacím jazykom, ako napríklad Java, C++, C# pričom je zároveň jednoduchší ako väčšina bežných programovacích jazykov. Snaží sa byť jednoduchý, prekladateľný, abstraktný. Jeho zápis je minimálny, avšak dostatočný na modelovanie všetkého potrebného[4].

Verzia jazyka OAL používaná v prostredý **AnimArch** tvorí podmnožinu pôvodného OAL. Táto podmnožina bola ďalej modifikovaná a rozširovaná s cieľom uspokojiť požiadavky na záznam našich animácií v tejto práci. S využitím tohto jazyka sme sa snažili vytvoriť príslušné animácie.

1.2 Použité nástroje

V priebehu realizácie nasej prace sme využívali viacero nástrojov zameraných na vývoj prostredníctvom MDD. Tieto nástroje nám umožnili vytvárať diagramy rámcov, následne zobrazovať tieto diagramy spolu s tzv. animáciou vytvorenou prostredníctvom jazyka OAL priamo na príslušnom diagrame.

1.2.1 Enterprise Architect

Enterprise Architect je vizuálny nástroj pre modelovanie a dizajn, postavený na štandardoch OMG UML. Podporuje návrh a konštrukciu softvérových systémov, podnikových procesov a odvetvových domén. Taktiež umožňuje modelovanie organizačnej architektúry a riadenie fáz vývoja aplikácií, vrátane sledovateľnosti, riadenia projektov a zmien v kóde. Používa UML ako hlavný modelovací jazyk a integruje sa s ďalšími špecifikáciami OMG UML a odvetvovými rámci.

Pre prácu bol tento software použití na generovanie prvotných XML súborov ktoré sme následne importovali do AnimArch.

1.2.2 AnimArch

V našej práci využívame niektoré funkcionality poskytované nástrojom AnimArch a zároveň rozširujeme jeho databázu dostupných príkladov návrhových vzorov a stylov. AnimArch je aplikácia postavená na Unity a napísaná v jazyku C#. Jeho hlavným zámerom je poskytnúť používateľovi pomoc pri porozumení štruktúry vlastného kódu. Funkčnosť AnimArch spočíva v zobrazovaní diagramov tried a animácií, ktoré vizualizujú interakcie medzi metódami daných tried.

Tento nástroj umožňuje importovať diagramy tried z nástroja Ënterprise Architectälebo vytvárať jednoduché diagramy priamo v prostredí AnimArch. Tieto diagramy obsahujú triedy a vzťahy ako asociácie, agregácie a generalizácie, pričom umožňujú pridávať atribúty a funkcie s parametrami triedam.

Animácie v AnimArch zobrazujú spustený kód v jazyku OAL a umožňujú vytvárať jednoduché animácie prostredníctvom grafického prostredia. Tieto animácie sa ukladajú vo formáte JSON, čo umožňuje kompatibilitu, jednoduché spúšťanie a pochopenie kódu. Okrem toho je možné vytvárať, ukladať a načítavať diagramy a animácie zo súborov.

V AnimArch je tiež možné konvertovať kód napísaný v jazyku OAL do kódu v jazyku Python s dodržaním konzistencie a funkčnosti softvéru. Táto funkcia podporuje vývoj v súlade so zásadami metódy MDD. V našej práci sa venujeme najmä rozšíreniu databázy diagramov a animácií nástroja AnimArch, čo umožní používateľom lepšie porozumieť interakciám v rámci implementovaných návrhových vzorov.

1.3 Návrhové štýly a vzory

Cieľom našej práce je navrhnúť rámece pre dva štýly, čo predstavuje výraznu odchýlku od implementácie vzorov. V tejto kapitole sa preto zameriame na základné rozdiely, spôsoby na to ako dosiahneme kód z animácie, a zároveň na to, ako je animácia reprezentovaná v našom pracovnom prostredí. V tejto časti si priblížime zásadné odlišnosti medzi štýlmi a vzormi.

1.3.1 Návrhové štýly

Architektonický štýl, upresňuje súbor princípov a usmernení, ktoré určujú organizáciu komponentov systému a vzťahy medzi nimi. Poskytuje abstrakciu na vysokej úrovni. Štýly pomáhajú formovať celkovú štruktúru softvérovej aplikácie, ovplyvňujú rozhodnutia o tom, ako rôzne komponenty vzájomne komunikujú [25]. Štýly sú často vyberané na základe špecifických požiadaviek a obmedzení systému. Rôzne štýly zdôrazňujú rôzne aspekty. Pre príklad sme si vybrali zopár najznámejších architektonických štýlov:

Klient-Server Tento štýl(z ang. Client-Server) rozdeľuje aplikáciu na klienta a server, pričom klient je zodpovedný za užívateľské rozhranie a server spravuje úložisko údajov a požiadavky na spracovanie.

Mikroservisný V tomto štýle (z ang. Microservices) je systém rozdelený na malé, nezávislé služby, ktoré komunikujú prostredníctvom dobre definovaných API(rozhraní). Každá služba je zodpovedná za konkrétnu obchodnú logiku ktorú vykonáva.

Udalosťou-riadený Pri tomto štýle(z ang. Event-Driven) sa zameriava na produkciu, detekciu, spotrebu a reakciu na udalosti. Komponenty komunikujú prostredníctvom udalostí, týmto umožňuje voľne prepojené a škálovateľné systémy.

Vrstevný Tento štýl(z ang. Layered) organizuje komponenty do horizontálnych vrstiev, ako sú prezentačná pod ktorou si vieme predstaviť to čo užívateľ vidí (UI), obchodná logika a perzistentná vrstva zvaná aj ako vrstva prístupu k údajom. Každá vrstva má špecifickú zodpovednosť za ktorú zodpovedá.

Na zaver treba dodať ze výber správneho architektonického štýlu je kľúčový pre úspech vytváraného softvéru, pretože ovplyvňuje kvality systému, ich flexibilita, udržateľnosť a škálovateľnosť.

1.3.2 Návrhové vzory

Návrhový vzor je vlastne spôsob ktorí nám pomáha rieši často vyskytujúce sa problémy v softvérovom návrhu. Je to šablóna ktorú je možné prispôsobiť na riešenie konkrétneho problému flexibilným a efektívnym spôsobom. Návrhové vzory nie sú úplnými maketami alebo striktne definovaním kódom. Namiesto toho poukazujú na spôsob komunikácie medzi definovanými objektami[23].

Návrhové vzory zahrňujú najlepšie postupy pre softvérový vývoj. S ich súdržnosťou v systéme vieme vytvárať kód, ktorý je modulárny, flexibilný a škálovateľný, čo sú klúčové vlastnosti ktoré by mať každý softvéry návrh. Niektoré bežné návrhové vzory zahŕňajú:

Singelton Najčastejšie používaný, zabezpečuje, že trieda má len jednu inštanciu a poskytuje globálny prístup k nej.

Factory Method Tento vzor zvaní aj vzor továrne definuje rozhranie pre vytváranie objektu, ale necháva výber jeho typu na podtriedach, vytvára inštanciu vhodnej podtriedy. Patri medzi vzory ktoré už boli implementované v AnimArch

Observer Definuje závislosť medzi objektmi tak, že keď sa jeden objekt mení, všetci jeho závislí sú automaticky upozornení a aktualizovaní. Taktiež už bol implantovaní v AnimArch.

Prostredníctvom využívania návrhových vzory prispievame k vytváraniu udržateľného, škálovateľného a efektívneho softvéru prostredníctvom podpory opätovného použitia overených riešení pre časté problémy v softvérových návrhoch. Zároveň ich využitím nám slúžia ako sprievodca vo projektoch.

1.3.3 Hlavné rozdiely medzi štýlmi a vzormi

Záverom je, že rozdiel medzi architektonickými štýlmi a architektonickými vzormi spočíva v ich dôraze a rozsahu. Architektonické vzory sa sústreďujú na riešenie konkrétnych problémov v danom kontexte a poskytujú komplexné riešenie. Naopak, architektonické štýly sa zameriavajú na širší architektonický prístup a poskytujú ľahšie usmernenie o tom, kedy môže byť určitý štýl vhodný alebo nevhodný. Hoci sa na prvý pohľad môže zdať, že rozlišovanie medzi architektonickými štýlmi a vzormi je pedantické, dúfam, že uvedený prístup pomôže zjednodušiť s pomocou využitia tohto zdroju[21]:
vzor: problém, kontext → architektonický prístup
štýl: architektonický prístup

1.4 Generovanie Python kódu

Túto časť budeme venovať možnosti generovania kód, pre našu prácu bude dôležitý python kód ktorý bude za pomoci využitia Antleru generovaný, zároveň sa dozvieme ako tento proces prebieha. Keďže pomocou softvéru AnimArchu v ktorom robíme našu diplomovú prácu vieme generovať aj kód je pre nás dôležite priblížiť tieto technológie. V prací sa zameriavame hlavne na python kód v ktorím overíme funkčnosť finálnych implementácii na ktorých bude následne prebiehať náš výskum.

1.4.1 ANTLR

Nástroj ANTLR (ANother Tool for Language Recognition), ktorý je výkonný generátor syntaktického analyzátoru (parsera). Dôvodom priblíženia tohto prostriedku je fakt, že existujúci parser v AnimArchu bol vytvorený pomocou tohto nástroja.Nástroj ANTLR ako taký sme nepoužili ale využili bola jeho modifikovaná verzia pre AnimArch. Tento nástroj slúži na preklad štruktúrovaného text na základe formálneho popísania jazyka nazývaného gramatika[20].

ANTLR je široko používaný na tvorbu parserov pre rôzne jazyky, nástroje a frameworky. Jeho vstupom je bezkontextová gramatika v rozšírenej Backus-Naurovej forme (EBNF), čo je notácia popisujúca formálnu gramatiku jazyka, konkrétne **OAL** v našom prípade spomenutí v kapitole 1.1.4.

Tento parser pracuje na princípe rekurzívneho zostupu a vytvára parsovací strom, ktorý sa postupne spracováva od koreňa k listom. Tento spôsob parsovania je známy ako parsovanie zhora nadol. ANTLR 4 používa technológiu LL(*) alebo ALL(*), ktorá predpovedá prepisovanie neterminálov pomocou funkcie nazvanej adaptivePredict. Na rozdiel od statickej analýzy gramatiky sa ALL(*) prispôsobuje vstupným vetám, ktoré sú mu prezentované počas parsovania[20].

1.4.2 Generovanie kódu

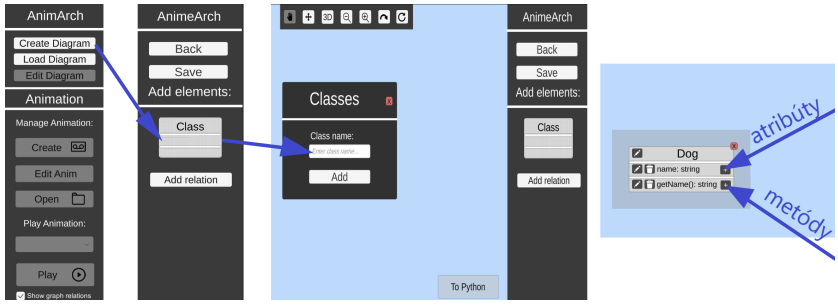
V súčasnej verzii AnimArch je možnosť generovať kód, ktorý z OAL kódu vytvorí ekvivalentný kód v jazyku Python. Toto tlačidlo je dostupné priamo po spustení prostredia, ako môžete vidieť v softvéri po ikonou pythonu. Pred kliknutím na tlačidlo je nevyhnutné načítať diagram a príslušnú animáciu (OAL kód). Po kliknutí má užívateľ možnosť vybrať miesto pre uloženie súboru. Následne, s pomocou týchto vygenerovaných súborov, môžeme overiť funkcionalitu návrhov, ktoré sme vytvorili. Táto časť bude súčasťou našich výskumných aktivít, ktorými sa budeme zaoberať v časti **Evaluácia a výsledky**.

1.5 Vizualizácia softvérových architektúr AnimArch

Vizualizácia je kľúčovou zložkou našej práce, a dôležitou častou hodnotiacej časti a preto je potrebné pochopiť všetky možnosti ktoré máme prístupné v softvéri AnimArch.

1.5.1 Vizualizácia v 2D AnimArch

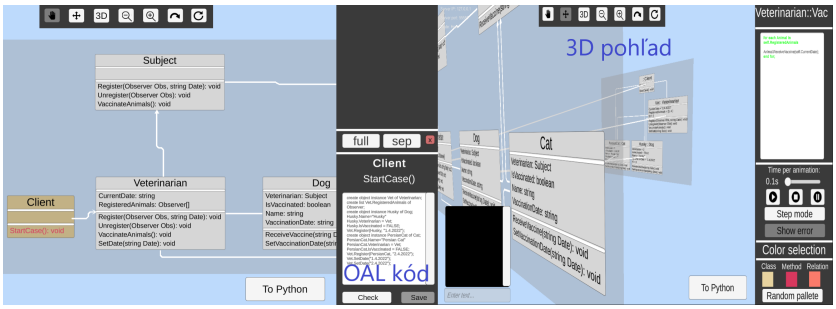
Animarch vieme používať aj ako softvére pre kreslenie diagramov, bez využitia vlastnosti ze môžme písať OAL kód. To znamená ze software sa dá používať aj ako spomínaní software Enterpise Architekt ale aj ako mnohé iné draw.io (online) alebo PlantUML (online). Každý z týchto softvéreou má svoju silu ale aj svoje nedostatky. AnimArch ma vo svojom základe možnosť vytvárať len triedne diagramy. Pre toto stačí kliknúť na tlačidlo „Create Diagram“ a následne na kliknutím a objekt „Class“ po ktorom dostaneme výzvu na pridanie mena pre vytvorenú triedu. Teraz môžme pridávať atribúty a metódy jednotlivým triedam ktoré sme vytvorili, celý postup je vidieť na obrázku 1.4. Na záver môžme uložiť tento diagram pod tlačidlom „Save“. AnimArch ponuka aj možnosť načítania už vytvorených diagramov a pokračovania v ich editovaný pod možnosťou tlačidla „Load Diagram“ a následne „Edit Diagram“, po skončený editovania je potrebne znovu uložiť pomocou „Save“ tlačidla. Animach nám ponúka možnosť presúvania a pochybovania s triedami, priblíženie, oddialenie a presunutie sa do bodu centra v prípade ak sa presunieme tam kde nebolo žiadané, toto platý aj pri využitý prostredia v 3D priestore.



Obr. 1.4: práca v 2D priestore aplikácie AnimArch

1.5.2 Vizualizácia v 3D AnimArch

Práca v 3D je prístupná priamo v základe tejto aplikácie ale pre využitie hlavného potenciálu 3D priestoru je potrebne ovládať OAL gramatiku pre písanie animácii. Pre použitie je nutné ovládať základné relačné vzťahy medzi triedami inak vzniká možnosť zlej animácie a užívateľ nemusí vedieť kde vznika chyba. Ak ideme vytvoriť animáciu diagramu je potrebné kliknúť v menu na tlačidlo „Create“ a následne vyselektovať konkrétu triedu a jej metódu. Pre každú animáciu je vhodné mať jednu triedu navyše ako s metódou „StartCase“, toto nieje striktne nutná požiadavka. Táto trieda načíta potrebne objekty a naštartuje celý chod animácie. Napísaním OAL kódu ktorím rozpohibujeme animáciu aj v tretej dimenzii teraz už je dobré aby sme ju aj vyžili. Pod delidlom hore je aj možnosť rotovania kamery 3D priestore. Týmto procesom môžme sledovať ako jednotlive triedy a im patriace objekty komunikujúce medzi sebou na objektovej vrstve. Tuto vlastnosť budeme využívať pri testovaní našich návrhov. Na obrázku 1.5 ukazujem ako vyzerá AnimArch v 3D. AnimArch podporuje aj VR a vytváranie diagramov pomocou VR-hadsetu. Tento prístup bol vytvorení a preskúmaný inými študentami.



Obr. 1.5: Pohľad na AnimArch v 3D

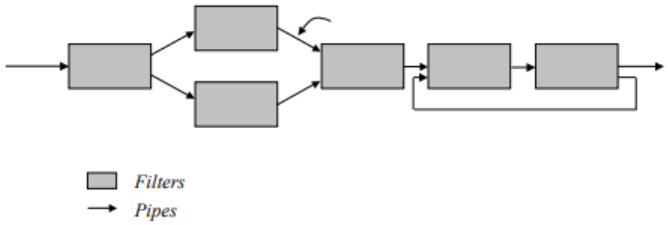
2 Návrh softvérových rámcov a ich implementácia

V nasledujúcej kapitole zosumarizujeme naše zistenia o architektonických štýlov Pipes and Filters a Blackboard, ktoré sa pokúšame implementovať v prostredí AnimArch. Budeme sa venovať jednotlivým iteráciám, ktoré sme vykonávali počas implementácií jednotlivých štýlov. Taktiež opíšeme problémy, ktoré vznikli v priebehu vývoja.

2.1 Charakteristika architektonického štýlu Pipes and Filters

Architektonický štýl dátovodov a filtrov (z ang. Pipes and Filters), označovaný v tejto sekcii skratkou **P&F** predstavuje štruktúru, ktorá využíva systém sérií filtrov, ktoré sú nezávislými komponentami, pričom komunikácia medzi nimi je realizovaná prostredníctvom dátovodov. [10].

P&F silne zdôrazňuje modularitu a funkčnosť jednotlivých filtrov. Tieto vlastnosti efektívne zvyšujú zrozumiteľnosť, udržateľnosť a rozširovateľnosť systému. Na obrázku 2.1, reprezentujúceho generickú implementáciu tejto topológie, môžeme vidieť jednotlivé komponenty (filtre), ktoré môžu byť nahradené v prípade chyby, alebo zmeny požiadaviek, bez nutnosti zmeny celkového procesu spracovania dát[10].



Obr. 2.1: generická topológia P&F [10]

Filters / Filtre Sú nezávisle samostatné komponenty, ktoré sú zodpovedné za vykonávanie prenosu, alebo zmeny dát. Práve z tohto dôvodu ich správanie delíme na 4 kategórie [7]:

1. dáta iba posúva ďalej, filter je pasívny
2. dáta iba prijíma, filter je pasívny
3. dáta prijíma a posúva, filter je pasívny
4. dáta prijíma a posúva a **spracuje**, filter je **aktívny**

Pipes / Dátovody Nám slúžia ako komunikačné kanály, ktoré spájajú filtre a uľahčujú dátový tok z jedného filtra do ďalšieho.

Source Medzi ďalšie základne pojmy v tejto architektúre patrí aj pojem „Sorce“. Tento pojem nie je v literatúre jednoznačne zadefinovaný. Podľa citácie [19] „Sorce“ predstavuje komponent zodpovedný za inicializáciu dát pre prvý filter v rade, teda začiatočný bod smerovania filtrov.

Sink V tejto architektúre je nevyhnutné zadefinovať aj koncový bod smerovania filtrov a tým je komponent „Sink“. Tento komponent má za úlohu finalizáciu od-

povede série filtrov v rade. „Sink“ je považovaný za filter, avšak naša predstava a pochopenie problematiky vylučuje túto klasifikáciu. Napriek snahe sa nám nepodarilo nájsť lepšiu alternatívu k jeho implementácii než v[19].

Vlastnosti architektonického štýlu P&F Ako už bolo spomenuté vyššie, architektonický štýl **P&F** má veľa možností pre praktické využitie:

Rozpájateľnosť / Decopuling Filtre v tomto architektonickom štýle operujú s voľnými spätnými väzbami, to znamená, že sú vzájomne nezávislé a nevedia o sebe navzájom. Táto vlastnosť podporuje flexibilitu a modifikovateľnosť, pretože akékoľvek zmeny v jednom filtre nemajú vplyv na ostatné filtre.

Opakované použitie / Reusability Každý filter je navrhnutý tak, aby mohol byť opätovne použiteľný, pretože majú špecifickú funkcionalitu. To podporuje ich znovu použiteľnosť a zjednodušuje vývoj nových systémov s využitím už existujúcich filtrov.

2.1.1 Problematika spojená so štýlom Pipes and Filters

V tejto časti uvádzame problémy, ktoré môžu nastať pri implementácii architektonického štýlu **P&F**. Toto sú tie najčastejšie:

Nadbytok režijného výdaja / Overhead Použitím dátovodov na prenos údajov medzi filtrami môžu nastať isté problémy serializácie, prípadne deserializácie.

Komplexnosť / Complexity Tento architektonický štýl prináša výhody pre väčšie systémy, ale častokrát prináša aj zbytočnú komplexnosť vo vzájomne prepojených systémoch.

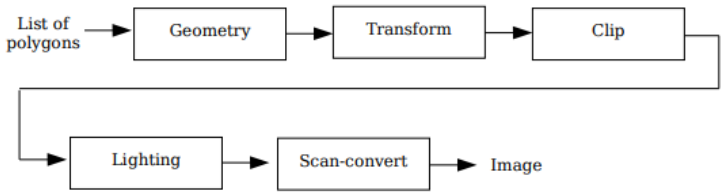
Problémy spojené s architektonickým štýlom Pipes and Filters Ďalej bližšie opíšeme problémy, s ktorými sme sa potykali pri implementácií architektonického štýlu Pipes and Filters. Snažili sme sa navrhnúť ľahko škálovateľný a zároveň udržateľný rámec, ktorý by reprezentoval tento architektonický štýl. Problémy v implementácii vznikli až vtedy, keď sme analyzovali praktickú realizáciu tohto štýlu v konkrétnych programovacích jazykoch, a to v C#[3],Java[1],Python[2].

Výraznou odchýlkou od našich predchádzajúcich skúseností je použitie dátovodov v každej z týchto implementácií. Tento objekt bol iba imaginárny a nikdy nepredstavoval žiadnu konkrétnu dátovú štruktúru, čo viedlo k mnohým zlým návrhom v úvodných fázach našej práce

2.1.2 Predchádzajúce implementácie štýlu Pipes and Filters

Systémy Unix Pri využívaní shell comandov v prostredí unix (operačný systém) sa často stáva, že príkazy pretekajú cez ďalšie príkazy. Na tento účel sa využíva znak | ako dátovodov a expressions sú zreťazené do podoby „exp | exp | exp“, čím sa vytvára celkový dátový tok.

Obrazové spracovanie Pre toto spracovanie sa nám podarilo nájsť pekný článok zameraný presne na tento problém [19]. Autori v ňom priamo opisovali ako je možné využiť túto architektúru na spracovanie obrazu. Na obrázku 2.2 od autorov môžeme vidieť ich prístup na sériové spracovanie obrazu.



Obr. 2.2: exemplár spracovania obrazu s využitím topológie P&F [19]

Na obrázku2.2 môžeme vidieť použitie rôznych obrazových transformácií, pričom každá z transformácií predstavovala iný filter.

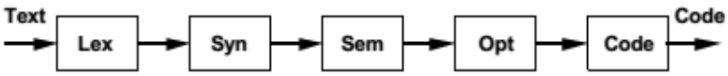
Dávkové spracovávanie dát Pri takejto implementácii je kladený dôraz na to, aby dáta boli spracovávané postupne a nie naraz. Využitie filtrov predstavuje dátové spracovania podľa požiadaviek, ktoré potrebujeme. Pri použití architektúry pipes and filters vieme jednoducho nastoliť dávkovanie a detekovanie chýb, pokiaľ niektorý z použitých filtrov nesplňuje požadované kritériá.

2.1.3 Návrh štýlu Pipes and Filters

Pri výbere problému sme sa zameriavali hlavne na jednoduché problémy ako je dávkové spracovanie, alebo triviálne rekurzívne prípady, kde by sme vedeli poukázať na rozdielnosť a spôsoby využitia rôznych filtrov. V tejto časti preukážeme 2 návrhy, ktorými sme sa zaoberali.

Referencie na objekty: V tomto probléme išlo o to, že sa nám jednotlivé objekty uložené v zoznamoch reprezentovali iba ich *ID*. Tento problém bol vyriešený zmenou reprezentácie na meno objektu pri vytváraní. To znamená, že ak vytvárame objekt takto: *create object instance COVID19 of Diagnosis;*, tak meno bude **COVID19**.

Použitie dávkového spracovávania na vstupy používateľa Spôsob ako dávkové spracovanie funguje sme už opísali v sekcii 2.1.2 Toto spracovanie využíva lineárne aplikovanie filtrov jedného za druhým obrázok 2.3. Tento spôsob aplikácie je krásnou ukážkou sily implementácie tohto štýlu. Z hľadiska spracovania a implementácie je lineárne spracovanie dát jednoduchou záležitosťou, ktorá nie je vôbec náročná. Preto sme tento spôsob ihneď vylúčili ako nevhodný príklad pre implementáciu celkovej architektúry.



Obr. 2.3: exemplár dávkového spracovania [7]

Ukážka Pipes and Filters na Fibonacciho postuponosti Predpokladáme, že každý z nás sa už stretol s implementáciou fibonacciho postupnosti. Preto si myslíme, že tento príklad by mohol byť správnym príkladom, na ktorom sa dá zrozumiteľne vysvetliť využitie štýlu pipes and filters. Existuje viacero vhodných implementácií: **Rekurzívna** kedy riešime volania funkcie samého do seba s *n-1* a *n-2*. Na obrázku 2.4 môžeme vidieť implementáciu v kóde python. Teraz túto ukážku môžeme prerobiť do jednotlivých filtrov, kde *n¡0*, predstavuje jeden nezávislý celok. Potom *n == 1* predstavuje ďalší a *n == 1* alebo *n == 2* posledný a následne rekurzívne volanie s návratom.

```
def Fibonacci(n):
   if n < 0:
        print("Incorrect input")
    elif n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
print(Fibonacci(9))
```

Obr. 2.4: kód rekurzívna implementácia fibonacciho postupnosti

Lineárna V tomto príklade máme jasne definovaný postup. Filtre zostávajú rovnaké ako v rekurzívnej implementácii, ale na rozdiel od rekurzie, si vieme udržiavať dáta v premenných počas behu.

Tento spôsob implementácie prináša veľa výhod, pretože návratová hodnota zodpovedá premennej *b*, ktorá bola definovaná na začiatku behu programu ako je ilustrované na obrázku 2.5. Týmto spôsobom nám vie vzniknúť filter výstup s jasne definovanou podmienkou.

```
def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return 0
    elif n == 1:
        return b
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b
print(fibonacci(9))
```

Obr. 2.5: kód lineárnej implementácia fibonacciho postupnosti

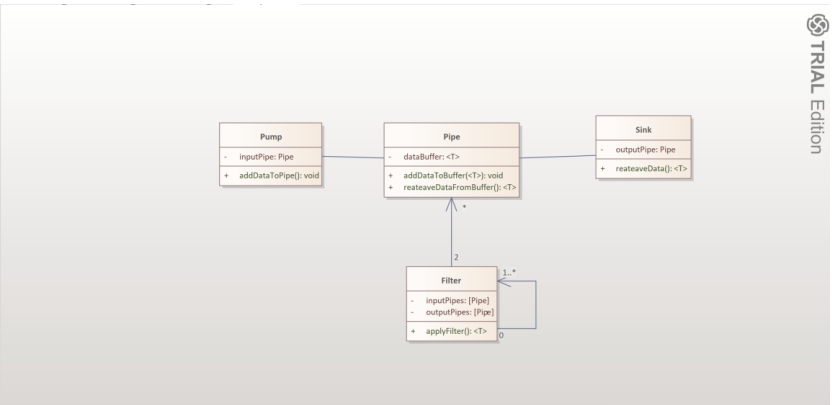
2.2 Implementácia rámcu Pipes and Filters

V nasledujúcich podkapitolách sa budeme venovať jednotlivým iteráciám, ktorými sme prešli počas vývoja.

2.2.1 Iterácia 1 - Prvotná expozícia P&F

V prvej iterácií sme začali skúmať možnosti tvorenia diagramov v aplikácii Enterprise Architect (ďalej už len EA) a AnimArchu. Počas tejto iterácie sme otestovali všeobecnú implementáciu štýlu a pokusnú implementáciu fibonacciho postupnosti. V softvéri EA sme vytvorili class komponenty štýlu. Tento štýl pozostáva zo 4 komponentov vid. obr. 2.6: Pump, Sink, Pipe(dátovod) a Filter. Class Pipe prevezuje class Pump a Sink. Filter dedí dva alebo viac objektov dátovod a podmienkou je, aby každý filter mal minimálne 2 dátovody. Class filter môže dediť ďalšie filtre *1..**. V EA sme si vygenerovali .xml súbor tohto diagramu. V

tejto implementácii sme využívali generické typy atribútov, ktoré by boli implementované jednotlivou dátovou štruktúrou, napr. list<string>, ... Class pump ma atribút input Pipe typu Pipe(dátovod) a umožňuje metódy add data to pipe, teda pridávať vstupné dáta do systému.

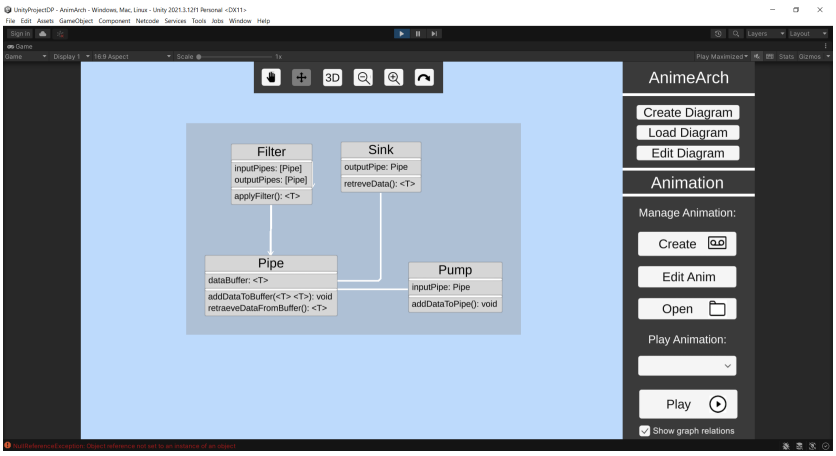


Obr. 2.6: všeobecná implementácia štýlu P&F v Enterprise Architect

Class sink má atribút outPipe typu Pipe a pomocou metódy retrieve umožňuje získať spracované dáta po aplikovaní všetkých filtrov. Class pipe je trieda, ktorá uchováva dáta v atribúte databuffer. Pomocou metódy addDataToBuffer vieme pridávať dáta do dátovodu a pomocou metódy retrieveDataFromBuffer vieme vyberať. Class filter pozostáva z dvoch atribútov, vstupným a výstupným dátovodom.

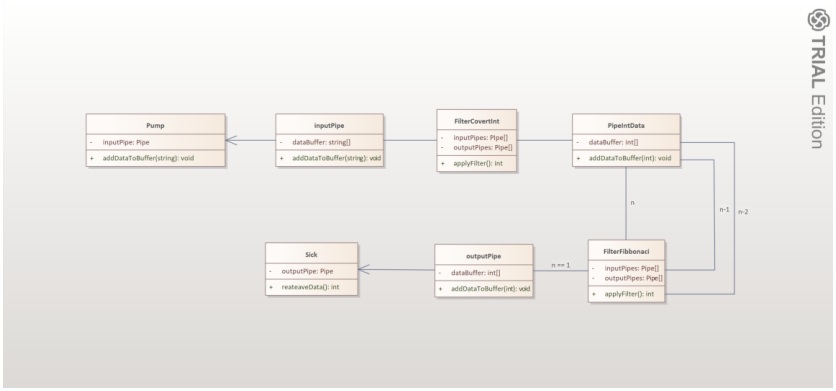
Na obrázku 2.7 môžeme spustenie vygenerovaného xml súbor z EA v prostredí AnimArchu. Pri bližšom pohľade na obrázok môžeme vidieť chybu v dedičnosti filtrov. Problémom bolo, že vzťah *1..** sa neprejavil v prevedení AnimArchu. Pre riešenie tohto problému je nutné navrhnúť všeobecnejšie riešenie dedičnosti. Ďalším problémom boli generické typy, ktoré AnimArch nepodporuje. Riešenie tohto problému je zložité, a bližšie sa k nemu vyjadríme v ďalších implementáciách.

Rovnako ako v prvej časti sme sa pokúsili pracovať so softvérom EA a snažili sme sa implementovať rekurzívnu fibonacciho postupnosť, kde filterfibonacci vracal n-1 a n-2 ako vstupné dáta do dátovodu. V tomto príklade môžeme vidieť z obr.2.8 čítanie vstupu ako string, následnú konverziu do integeru pomocou filtra



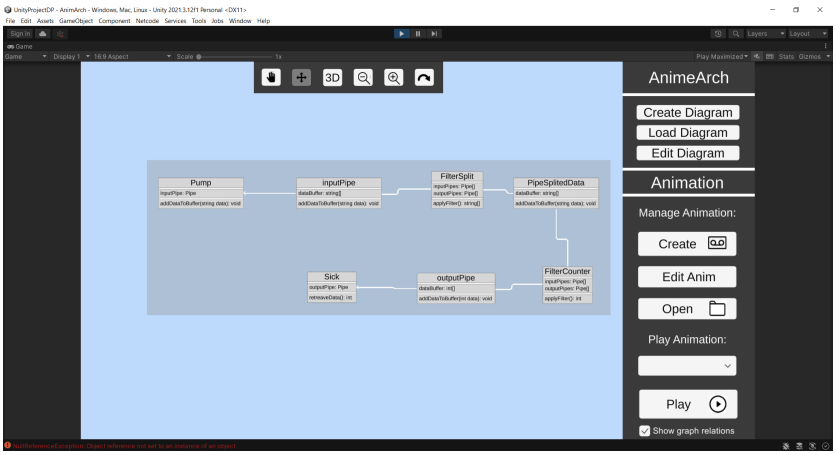
Obr. 2.7: konverzia všeobecnej implementácie 2.6 do AnimArch

filterConvertInt a výstupné dáta sa dostanú do konečného dátovodu outputPipe a následne sa zobrazia v Sink.



Obr. 2.8: implementácia fibonacciho v Enterprise Architekt

Tak ako v predošlom prípade sme aj tento model uložili do xml súboru a následne nahrali do aplikácie AnimArch obr.2.9. Pri tejto implementácii sme evidovali stratu vzťahov medzi triedami *filterCounter, pipeIntData*.



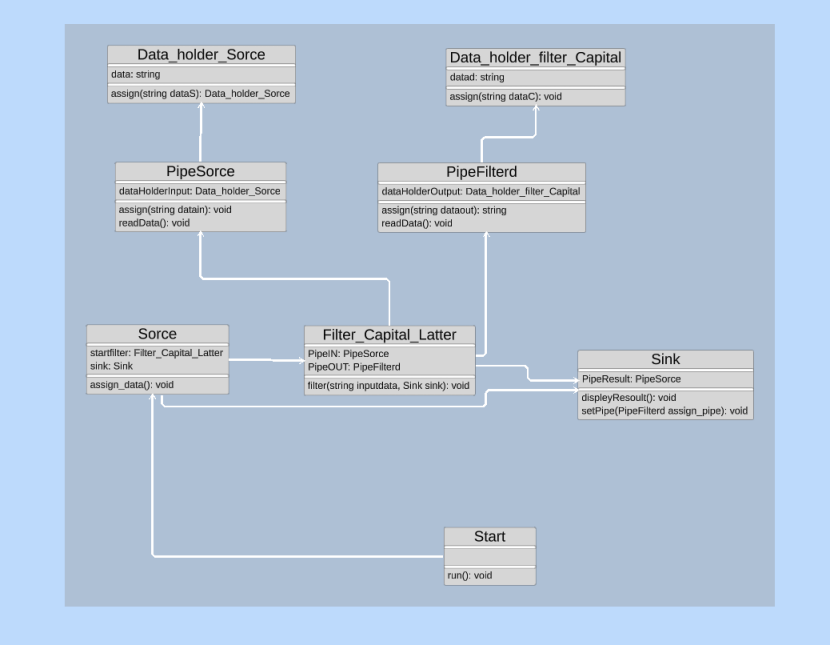
Obr. 2.9: konverzia implementácie 2.8 do AnimArch

2.2.2 Iterácia 2 - dávkové spracovanie P&F

V priebehu tejto iterácie sme sa odchýlili od prvej fázy, teda od výberu implementácie fibonacciho postupnosti na batch processing. Namiesto toho sme sa snažili osvojiť si jazyk OAL a získať skúsenosti s prácou v AnimArch.
Na obrázku2.10 môžeme vidieť implementáciu dátového spracovania textu, pričom sme sa pokúsili filtrovať veľké písmená. Na tomto diagrame môžeme vidieť filter *Filter_Capital_Letter*, ktorého metóda filter() prijíma vstupné dáta typu string a pozná výstupné miesto, kam majú byť dáta posielané. Samotná metóda prejde každé písmeno zo stringu a zapíše ho do výstupného dátovodu *PipeFiltered*. Tento diagram bol vytvorený v prostredí AnimArch.

Na obrázku 2.11 môžeme vidieť implementáciu OAL kódu pre metódu filter z class *filter_capital_letter* pomocou *create object instance* vytvoríme objekty *PipeSource* a *PipeFiltered* a ďalšie. Pomocou volania metódy *assign* na objekte *PipeIN* pridáme vstupné dáta. Metóda *readData* zabezpečuje prečítanie vstupných dáta z dátovodu a následne vykonaním for cyklu vyfiltrujeme veľké písmená.

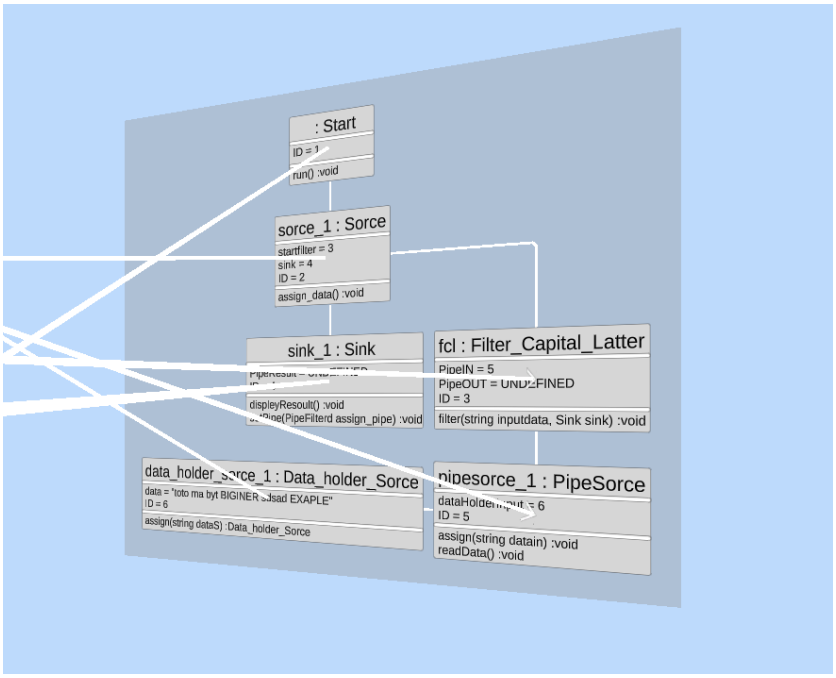
AnimArch nám umožňuje aj sledovanie animácie na objektovom svete. Pre príklad veľkých písmen to vyzerá takto obr. 2.12 Z obrázka môžeme vidieť jasné nedostatky v OAL kóde, keďže niektoré objekty nemajú priradené iné objekty.



Obr. 2.10: triedny diagram pre filtrovanie veľkých písmen

```
create object instance pipesorce_1 of PipeSorce;
self.PipeIN = pipesorce_1;
create object instance pipefilterd_1 of PipeFilterd;
self.PipeOUT = pipefilterd_1;

self.PipeIN.assign(inputdata);
pipesorce_1.readData();
outputData = "";
for each latter in inputdata:
if (latter == Capital)
outputData += latter;
end if;
end for;
self.PipeOUT.assign(outputData);
```

Obr. 2.11: kód OAL k filtrovanie veľkých písmen



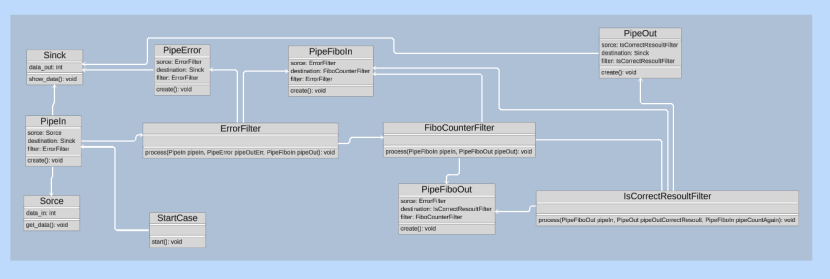
Obr. 2.12: objektový diagram pre filtrovanie veľkých písmen

2.2.3 Iterácia 3 - Ucelenie modelu P&F

V tejto iterácií sme sa vrátili späť k fibonacciho problému, ktorý sme sa pokúsili implementovať v iterácií 1. Rozšírili sme ho o zdroje [19] ako môžme vidieť na obr.2.13. To znamená, že sme pridali objekty pre vstup a výstup dát, teda Source a Sink. Následnú rekurziu sme sa pokúsili vyriešiť pridaním ďalšieho filtra, cez ktorý dáta prechádzali (*IsCorrectResultFilter*). Tento filter by vyhodnotil, či je fibonacciho iterácia už v konečnom stave, alebo ešte pokračuje vo výpočte. V prípade ak je iterácia konečná, dáta sa posunú do triedy *Pipeout* a následne do sinku. V opačnom prípade, teda ak iterácia pokračuje, tak sa dáta presúvajú do *PipeFiboin* , ktorá posúva dáta ďalšiemu filtru *FiboCounterFilter*, ktorý následne vykonáva ďalší výpočet fibonacciho iterácie.

Celková implementácie nepracuje na rekurzívnej báze, ale na lineárnej báze fibonacciho počítania. Jediná rekurzívna class, ktorú využívame je pipe fiboin, do

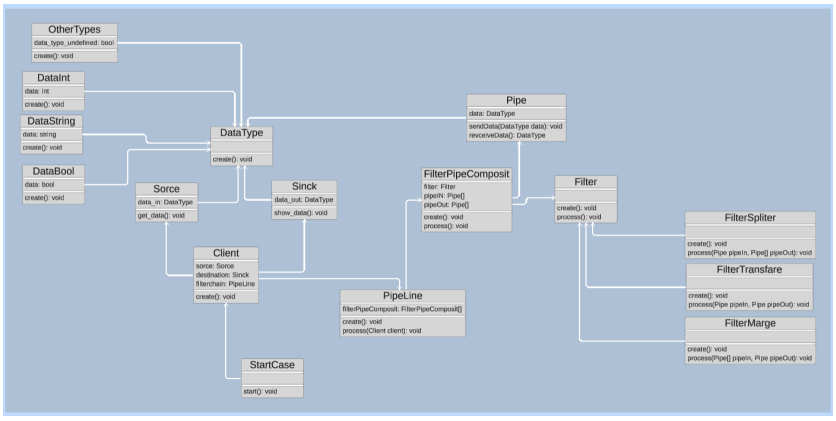
ktorej posúvame dáta.



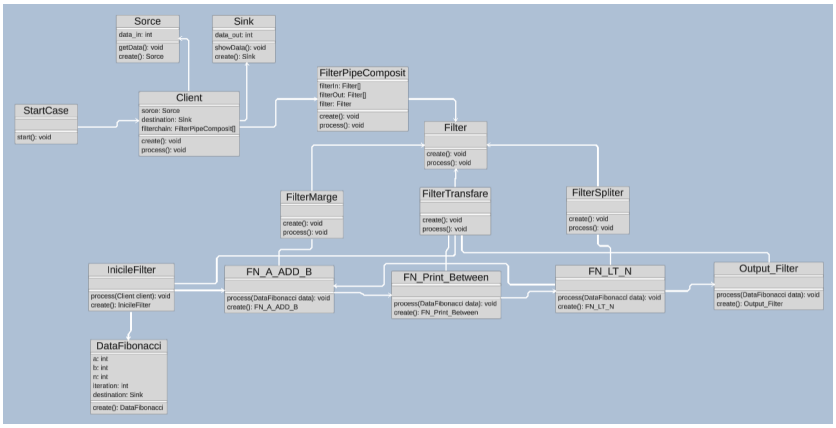
Obr. 2.13: diagram tried pre fibonacciho postupnosť

Na obrázku2.14 sme skombinovali doteraz zistené poznatky a zároveň sme odstránili generické typy využité v 1. iterácií. Tie sme nahradili triedou *DateType*. Zároveň sme sa rozhodli používať ustálené názvy pre objekty *Source* a *Sink* . Pred touto implementáciou sme mali vždy problémy so spúšťaním nasledovných filtrov, preto sme sa rozhodli implementovať triedu *Pipeline*, ktorá obstaráva celý systém filtrov a dátovodov. *FilterPipeComposite* nám slúži ako kompozit objektu filtra a vstupných a výstupných dátovodov, ktoré prislúchajú k danému filtru. Trieda *Pipe* slúži len ako dátový prenášač. Jednotlivé filtre pri vykonávaní svojej práce dostávajú na vstup dva parametre, ktorými sú vstupné a výstupné dátovody. Následne z nich vyberajú a do nich posielajú dáta.

Počas tejto iterácie sme sa zamysleli aj nad **ne**využitím triedy Pipe, keďže cez túto triedu dáta len pretekajú a nemajú obzvlášť inú funkcionalitu. Ako môžeme vidieť v implementácií na obrázku2.15 celkový triedny diagram sme zúžili na nutné minimum, aby sme odstránili zbytočnú komplikovanosť celkového systému. Zároveň na obrázku je možno vidieť, že naša implementácia fibonacciho, začína iniciálnym filtrom, ktorého cieľom je nastavenie hodnôt, následne sčítacím filtrom, ďalej filtrom vypisujúcim medzi výpočty fibonacciho postupnosti počas behu programu a nakoniec porovnávacím filtrom, ktorý buď znova opakuje výpočet, alebo posúva dáta do výstupného filtra.



Obr. 2.14: všeobecná implementácia štýlu P&F

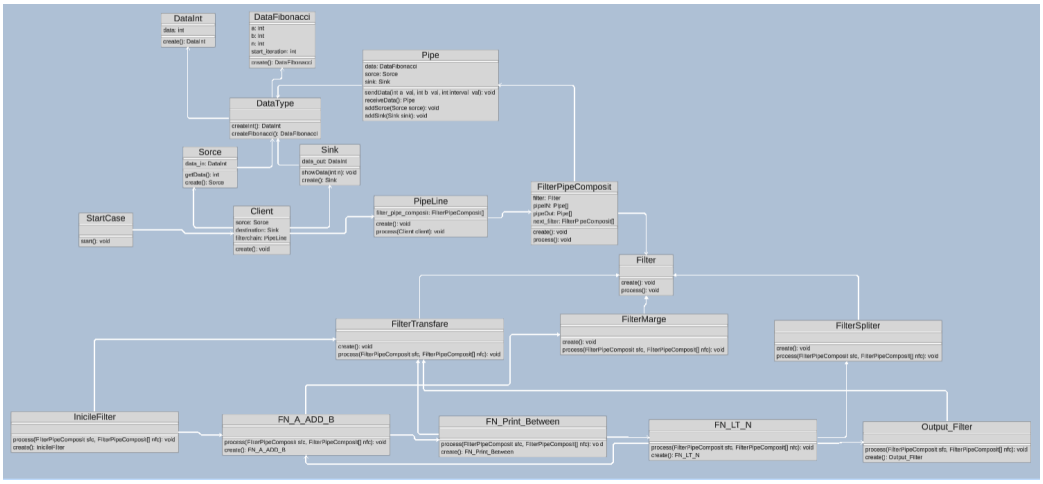


Obr. 2.15: implementácia štýlu P&F bez dátovodov

2.2.4 Iterácia 4 - Finálna implementácia P&F

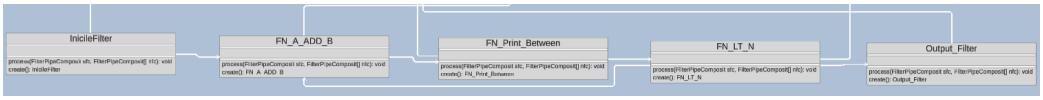
Táto iterácia je poslednou, ktorú tu opíšeme v časti určenej pre implementáciu rámca P&F. Na obrázku 2.16 prezentujem class diagram celého rámca, ktorý zahŕňa aj implementáciu fibonacciho postupnosti pomocou využitia komponentov štýlu.

Pri pohľade na obrázok 2.17 je možné pozorovať prepojenie inštancií objektu filtra. Zároveň môžeme všimnúť aj kľúčovú vlastnosť - znovupoužitia už exis-



Obr. 2.16: úplné zachytenie rámca P&F

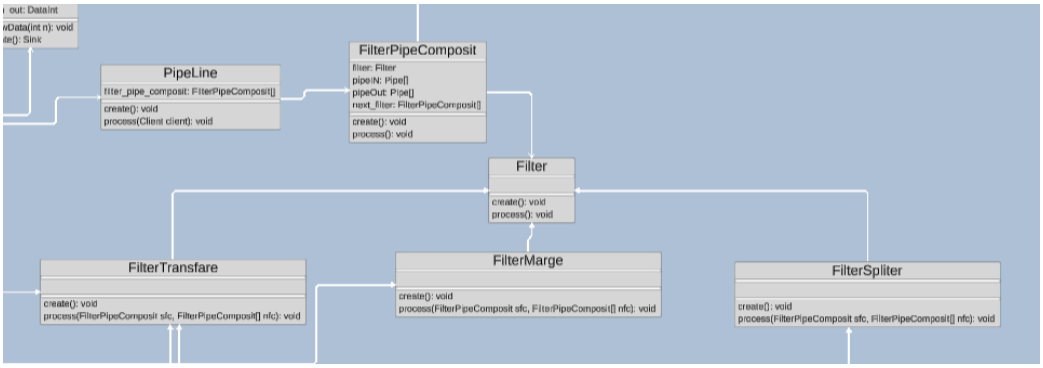
tujúceho filtra pre ďalšie iterovanie výpočtov. Táto vlastnosť je definovaná pomocou abstraktných tried filtrov.



Obr. 2.17: fibonacciho postupnosť priblížený pohlaď na triedny diagram

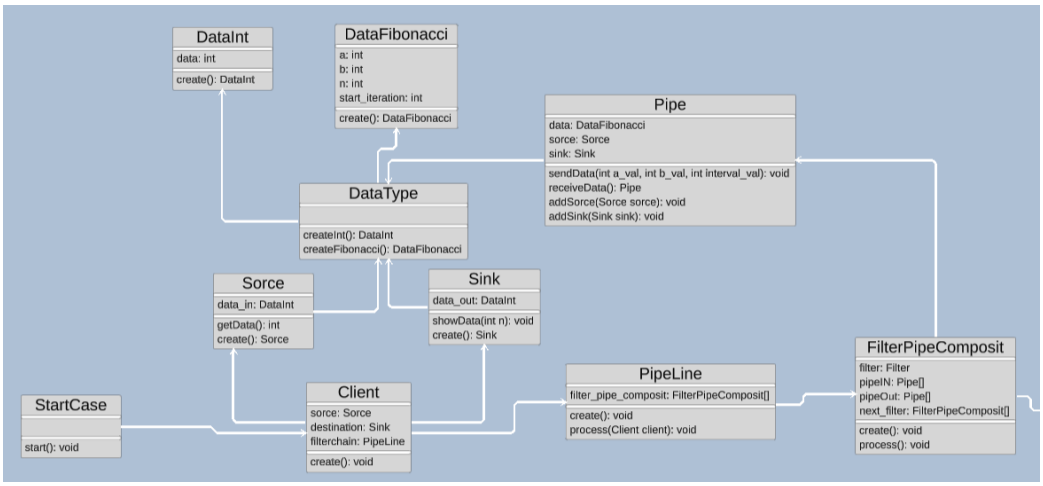
Pre našu implementáciu sú to triedy *FilterTransfare*,*FilterMarge* a *FilterSpliter*. Každá z týchto tried implementuje určitú vlastnosť filtrov, ktorú sme opísali v časti 2.1. Vlastnost filtrov uvedených na obr.2.18, okrem spracovania dát, ktorú môže robiť každá z nich, sú nasledovné: *FilterTransfare* má jeden vstupný a jeden výstupný dátovod, *FilterMarge* vie spracovať vstupy z viacerých dátovodov, a data posiela do jedného výstupného dátovodu. *FilterSpliter* má jeden vstupný a viacero výstupných dátovodov.

V tejto podcasti opíšeme obrázok 2.19, v ktorom môžeme vidieť implementáciu štýlu P&F. Trieda *DataType* vytvára objekty príslušných dátových typov pre problém, ktorý riešime. *Client* nám predstavuje rozhranie pre spúšťanie a vytváranie potrebných tried pre náš problém. V triede *PipeLine* môžeme vidieť atribút *filter_pipe_-*



Obr. 2.18: priblížený pohlaď na triedy spojené s componenotm filter

composit ktorý nám reprezentuje postupnosť aplikovania filtrov v poradí. Na záver trieda *FilterPipeComposit* nám predstavuje objekt, ktorý si pamätá akú filtráčnu vlastnosť má, aké dátovody používa a zároveň pozná aj svojich nasledovníkov v rade.



Obr. 2.19: priblížený pohlaď na implementáciu štýlu P&F

OAL kód V tejto časti sa budeme zaoberať implementáciou v OAL kóde a ako sme dospeli k tejto implementácii pomocou opisu filtrovacích komponentov.

Na úvod začneme s triedou *InicialFilter*, ktorej OAL kód je na obrázku 2.20.

Prechádzame všetky vstupné dáta pomocou for cyklu. Vyberieme dátovod, ktorý používa táto trieda, a nastavíme hodnoty na úvodné hodnoty pre výpočet fibonacciho postupnosti, teda *a = 0, b = 1, n = vstup užívateľa a iterácia = 0*. V prípade, že užívateľ zadal zlý vstup (n ¡= 0), zavoláme metódu na dedenom atribúte *sink.showData()* ktorá vypíše chybu, alebo spustíme proces na ďalšom filtri. Vzhľadom na všeobecnú implementáciu triedy *FilterPipeComposit* znovu používame for cyklus na toto volanie, aj keď je nasledovník len jeden filter.

```
for each IN in sfc.pipeIN
data_pipe = IN.receiveData();
data_pipe.data.n = data_pipe.sorce.data_in.data;
data_pipe.sendData(0,1,0);
if (data_pipe.sorce.data_in.data <= 0)
data_pipe.sink.data_out.data = 0;
data_pipe.sink.showData(data_pipe.data.n);
else
for each next_f_init in nfc
next_f_init.filter.process(next_f_init,next_f_init.next_filter);
end for;
end if;
end for;
```

Obr. 2.20: kód OAL triedy - InicileFilter

V obrázku 2.21 v triede *FN_A_ADD_B* môžno vidieť klasickú implementáciu fibonacciho počítania, kde *c = a + b, a = b, b = c*. Následne dáta posunieme dátovodu, ktorý aktualizuje atribúty triedy *Pipe* a znovu zavoláme ďalší filter.

```
for each INNN in sfc.pipeIN

data_pipe_print = INNN.receiveData();

if (data_pipe_print.data.start_iteration < data_pipe_print.data.n )
write("fibonacci of =",data_pipe_print.data.start_iteration," == ",data_pipe_print.data.a);
end if;

for each next_f_print in nfc
next_f_print.filter.process(next_f_print,next_f_print.next_filter);
end for;

end for;
```

Obr. 2.21: kód OAL triedy - FN_A_ADD_B

FN_Print je trieda, ktorou by sme nepotrebovali pre správny výpočet, ale roz-

hodli sme sa vypisovať na konzolu priebežné výstupy počítania fibonacciho postupnosti. Zápis na konzolu je pomocou kľúčového slova *write*.

FN_LT_N 2.22 je náročnejšia trieda na implementáciu po stránke OAL kódu, keďže dáta zo vstupného dátovodu musia prejsť dvoma rôznymi filtrami: buď *FN_A_ADD_B* ak je podmienka **LT** (less then) splnená a potrebujeme pokračovať v iterácii, alebo *OutputFilter* ktorého funkcionalitu opíšeme nižšie. Kľúčovým problémom bolo navrhnúť spôsob indexovania, ktorým filtru posunieme dáta. Keďže AnimArch v stave písania tohto textu nepodporuje indexovanie v poli, rozhodli sme sa vytvoriť podmienku s využitím pomocnej premennej *index* ktorá sa počas iterovania cyklu zvyšuje.

```
for each INNN in sfc.pipeIN
data_pipe_compare = INNN.receiveData();
index = 0;
for each next_f_compare in nfc
if (data_pipe_compare.data.n == data_pipe_compare.data.start_iteration AND index == 0)
next_f_compare.filter.process(next_f_compare,next_f_compare.next_filter);
index = index+1;
elif (data_pipe_compare.data.n > data_pipe_compare.data.start_iteration AND index == 1)
next_f_compare.filter.process(next_f_compare,next_f_compare.next_filter);
else
index = index+1;
end if;
end for;
end for;
```

Obr. 2.22: kód OAL triedy - FN_LT_N

Na záver opíšeme OAL kód s obrázka 2.23, kde výstupnú hodnotu posielame metóde zdedeného atribútu *sink.showData()*. Týmto sa ukončí filtrovanie a môžeme pokračovať s novými vstupmi, pre ktoré chceme vypočítať fibonacciho postupnosť.

2.3 Charakteristika architektonického štýlu Blackboard

Túto sekciu venujeme architektonickému štýlu Blackboard. Podrobne si priblížime, ako funguje a aké sú jeho možnosti využitia pri riešení problémov, spolu s

```
for each OUT in sfc.pipeIN

data_pipe_print = OUT.receiveData();

data_pipe_print.sink.data_out.data = data_pipe_print.data.a;
data_pipe_print.sink.showData(data_pipe_print.data.n);

end for;
```

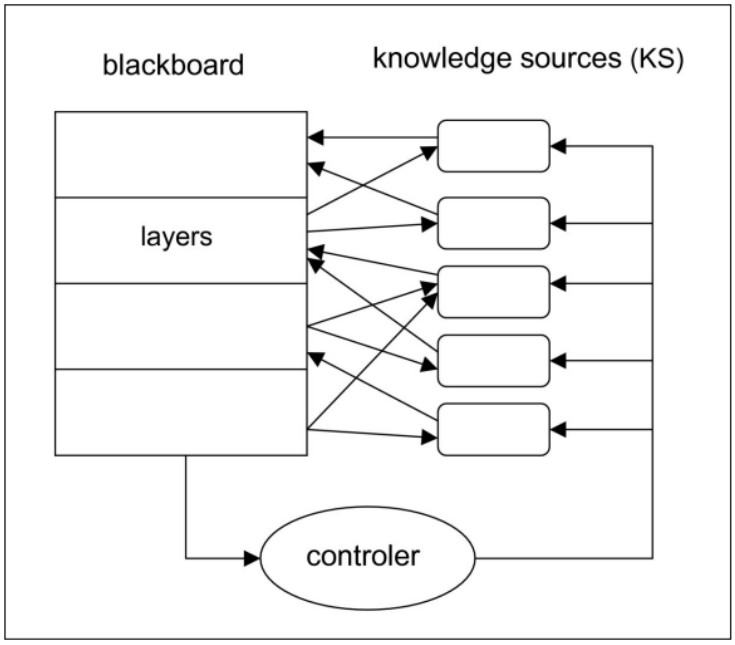
Obr. 2.23: kód OAL triedy - Output_Filter

našimi návrhmi implementácie tohto rámca v AnimArch.

Architektonický štýl Blackboard nám poskytuje určitý spôsob alebo štruktúru inteligentného systému, ktorý umožňuje prácu viacerým nezávislým komponentom. Tieto komponenty môžu spolupracovať na riešení problému. Blackboard slúži ako zdieľaný priestor pre komunikáciu a koordináciu jednotlivých komponentov riešiacich problém vid. obr.2.24. Tento štýl nám v základe podporuje paralelizáciu a adaptivitu celkového systému automaticky. Medzi hlavné výhody tohto systému patrí jeho schopnosť riešiť unikátne problémy, ktoré nemusia byť riešiteľné v polynomiálnom čase. Jeho nedeterministická povaha spôsobuje, že problémy môžeme riešiť len čiastočne, a nie je isté, či budú riešené správne. Preto sme vo výskumnej časti zvolili metódu RJT (relevance judgment techniques) na overenie jeho spoľahlivosti a správnosti implementovaného rámca [13][10].

Tabuľa (z ang. Blackboard) pod týmto názvom označujeme miesto, kde budú dáta uložené. Je to vlastne pracovné miesto známe aj ako zdieľaný pracovný priestor (repozitár), k ktorému pristupujú jednotlivé zdroje poznánia (z ang. Knowledge Source). Tento pamäťový priestor je modifikovaný jednotlivými komponentami počas riešenia problému. Každý z nich upravuje dáta na tabuli podľa svojej implementácie. Dáta sú synchronizované medzi všetkými komponentami, a tento proces zabezpečuje ovládač (z ang. Controller).

zdroje vedomostí (z ang. Knowledge Source) nám predstavujú komponenty ktoré riešia problém, ich implementácia nám určuje ako bude problém ktorí je na tabuli riešení.



Obr. 2.24: všeobecný diagram štýlu Blackboard.[8]

riadiaca jednotka (z ang. Controller) je posledný komponent štýlu ktorí umožňuje pridávanie jednotlivých poznatkových zdrojov. Hlavnou úlohou tohto komponentu je synchronizácia vykonávania funkcii v poznatkových zdrojoch.

Vlastnosti architektonického štýlu Blackboard **Flexibilita** Architektonický štýl Blackboard je známy pre jeho flexibilnú povahu. Vedomostné zdroje sa dajú ľahko pridať alebo upraviť bez ovplyvnenia celého systému. Je teda prispôsobivý na zmenu požiadaviek.

Paralelnosť V jadre tohto štýlu je podporovaná paralelnosť. Viacero vedomostných zdrojov dokážu pracovať súbežne s tabuli. Umožňuje efektívne riešenie problémov a využitie zdrojov v paralelnom alebo distribuovanom prostredí.

2.3.1 Predchádzajúce implementácie

V tejto časti spomenieme len zopár známych možností implementácie tohto štýlu.

Expertné systémy v takomto prípade hovoríme skôr o programe než o systéme. Tento program je vynikajúci v úzkej expertnej doméne, na ktorú sa zameriava. Často je známy aj ako „information-based expert system“. Takýto systém typicky pozostáva z veľkého množstva rozumných komponentov (KS), faktov, heuristík a pravidiel na aplikáciu týchto faktov[14].

Medicínsky systém na určenie diagnózy v takejto implementácii jednotlivé rozumové komponenty predstavujú rôzne medicínske expertízne pohľady, prispievajúc k analýze poskytnutej pacientom.

Systém na predpoveď počasia pri tejto implementácii sa na rozumové zdroje pozerá z iného pohľadu. Niektoré prispievajú aktuálnymi dátami, napríklad zo satelitov alebo iných zdrojov, a iné komponenty spracúvajú tieto dáta a aproximujú výpočty podľa toho, čo sa práve nachádza na tabuli.

2.3.2 Problematika spojená so štýlom Blackboard

Pri implementácii tohto rámca nevzniknú evidentné problémy, keďže rámec ako taký pozostáva z jasne definovaných komponentov a jeho štruktúra sa dá ľahko napodobniť. Medzi hlavné nevýhody tohto rámca patrí jeho celková testovateľnosť, pričom nám pomôže aj táto práca, v ktorej budeme vytvárať animáciu objektového sveta a všetkých komponentov, aby sme mohli bližšie sledovať ich správanie. Medzi ďalšie problémy patria aj nasledovné:

Správnosť riešenia v tomto prípade ide o kľúčovú nevýhodu tohto systému, ale zároveň nemožno povedať, že ide o nevýhodu, lebo pokusmi sa dá dospieť aj k oveľa efektívnejším riešeniam. Keďže jednotlivé myšlienkové komponenty sa vyvíjajú nezávisle od ostatných, ich práca môže nepriamo ovplyvniť úsudky ďalších komponentov v systéme.

Slabá efektivita a škálovateľnosť systém postavený na tejto architektúre je

často veľmi veľký a obsahuje veľa rozumných zdrojov, ktoré vzájomne pracujú na riešení problému. Vzájomná práca týchto rozumných komponentov je veľmi nákladná z výpočtového aj časového hľadiska. Tým, že systém obsahuje veľa týchto komponentov, jeho škálovateľnosť je veľmi obmedzená a limitovaná dostupnými zdrojmi.

2.3.3 Návrh štýlu Blackborad

Pri rozmýšľaní o tom, aké systémy by sme mohli implementovať pomocou tohto architektonického štýlu, nás napadlo zopár implementácií: medicínsky systém na detekciu chorôb, detekcia typu húb podľa opisu a pravidlový expertný systém na báze Prologu.

Pre implementáciu pravidlového systému na báze Prologu potrebujeme pochopiť základné problémy spojené s implementáciou pravidlového systému ako takého. Pravidlá, klauzuly, query - tieto prvky treba implementovať do class diagramu a následne dokázať ich vyhodnocovanie. Preto si definujeme základné pojmy s Prologu[17]:

Atóm je akýkoľvek výraz $P(t_1,\ldots,t_n)$, kde $P \in PL$, arity$(P) = n$ a $t_1,\ldots,t_n$ sú termy.

Pravidlo r je formula v tvare $A_0 \leftarrow A_1, \ldots, A_n$, kde $0 \leq n$ a každé A_i je atóm.

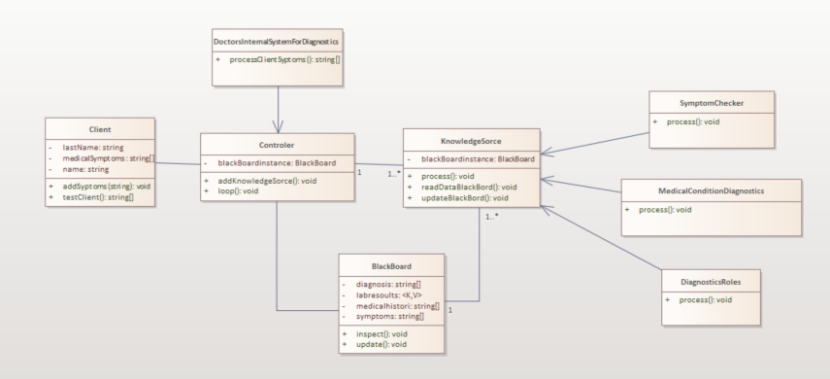
Logický program je konečná množina pravidiel. Ak máme pravidlo $r = A_0 \leftarrow A_1,\ldots,A_n$, tak Hlava$(r)$ =A_0 a Telo(r)=$\{A_1,\ldots,A_n\}$. Pokiaľ $n = 0$, $A_0 \leftarrow$ sa nazýva fakt.

2.4 implementácia rámcu Blackborad

2.4.1 Iterácia 1 - Prvý pohlaď na štýl Blackboard

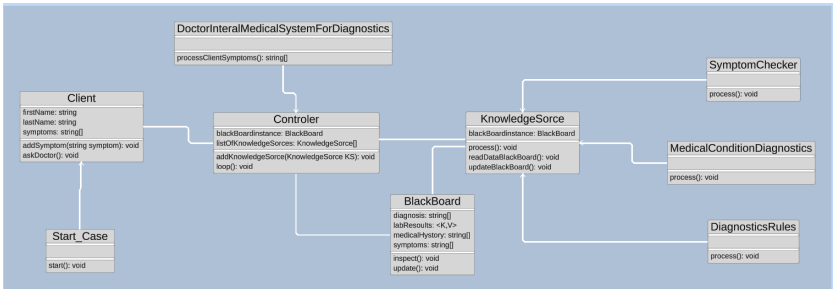
Pri tejto iterácii sme začali s vývojom v prostredí aplikácie **EA**, pričom základný model na detekciu chorôb bol pomerne dobrý výber implementácie tohto štýlu. Pri začiatku sme využili znalosti z kapitoly. 2.3. Tak ako bolo uvedené v tomto

článku [8], sme implementovali jednotlivé triedy pre komponenty tohoto modelu. Ako je možné vidieť z obrázka 2.25 znovu sme použili generické (<K,V>) typy pre vyhodnotenie laboratórnych testov.



Obr. 2.25: Medicínsky systém v rámci Enterprise Architekt

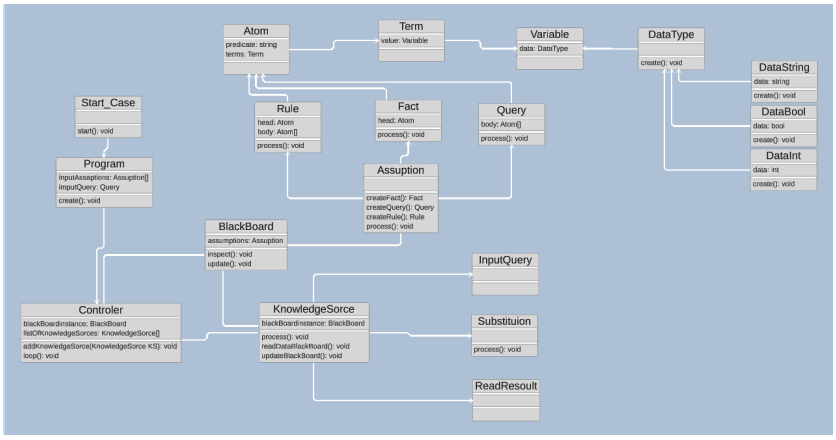
Po vygenerovaní .xml súboru, sme skúsili nahrať tento súbor do AnimArch a jednoducho naanimovať tento model. Opäť sme sa stretli s problémom neschopnosťou interpretovať generické typy v jazyku OAL. Medzi ďalšie problémy patrilo to, že pôvodný diagram nemal štartovaciu triedu, ktorú sme pridali a v neposlednom rade, sme mali zle určené asociačné vzťahy medzi objektami *Client, Controler a DoctorInternalMedicalSystemDiagnostics* ktoré je možno vidieť na obrázku 2.26.



Obr. 2.26: Konverzia medicínskeho systému do Animarch

2.4.2 Iterácia 2 - Pokus o implementáciu pravidlového systému na báze Prologu

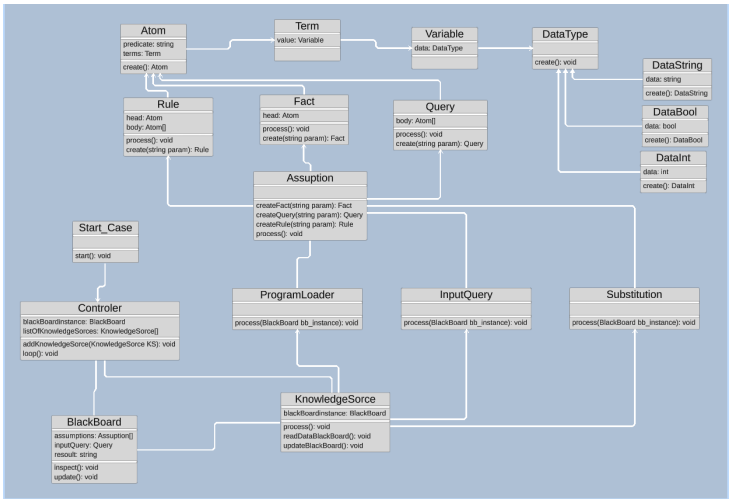
V tejto časti sme sa odchýlili od pôvodného nápadu implementácie medicínskeho systému a presmerovali naše úsilie k systému na báze pravidiel z jazyka Prolog. Počas tejto iterácie sme začali s návrhom z obrázka 2.27, na ktorom je badateľných viacero nedostatkov. Bola zavedená zbytočne rozsiahla trieda *Program*, ktorá mala slúžiť ako trieda na naštartovanie **BB** štýlu, teda riadiacej jednotky, tabule a jednotlivých rozumných zdrojov. Tento prístup nebol správny, keďže aj vstupný program môže byť vnímaný ako ďalší rozumný zdroj sám o sebe. Hlavným problémom tohto modelu bola opačná generalizácia medzi triedou *KnowledgeSource* a rozumnými zdrojmi.



Obr. 2.27: Pravidlový systém 1.pokus

Tento druhý náčrt pravidlového modelu je vlastne ďalšou iteráciou 2.27. Pri jeho tvorbe sme sa snažili opraviť nedostatky predchádzajúceho modelu. Zistili sme, že pri komunikácii potrebujú všetky myšlienkové zdroje prístup k pravidlám jazyka Prolog, tak ako môžeme vidieť na obrázku 2.28. Ich funkcionalita však bude mierne odlišná.

Trieda *ProgramLoader* bude načítavať program, ktorý chce používateľ riešiť. *InputQuery* bude očakávať vstup typu *Query*, ten po overení validity tejto qu-



Obr. 2.28: Pravidlový systém 2.pokus

ery zverejní obsah na tabuľu pre ostatné zdroje. Myšlienkový zdroj *Substitution* následne rieši obchodnú logiku tohto systému a pomocou resolvečných pravidiel sa snaží vyhodnotiť možné substitúcie, ktoré sú potrebné na splnenie vstupnej query.

V tejto podčasti 2.iterácie sa zameriame na samostatné spúšťanie rozumných zdrojov, pričom rozlišujeme dva prístupy. Prvým je lineárne spracovanie procesov (postupne jeden za druhým). Na obrázku 2.29 môžeme vidieť OAL kód s for cyklom, ktorý prechádza cez všetky rozumové zdroje.

```
for each value_KS in self.listofKS
value_KS.process();
end for;
```

Obr. 2.29: lineárne spustenie rozumových zdrojov

V druhom prípade, teda v paralelnom, môžeme pozorovať ten istý cyklus s použitím vlákien. Z kódu na obrázku 2.30 môžeme identifikovať dve kľúčové slová: *thread* pre spustenie vlákna a *end thread* pre ukončenie procesu vlákna.

```
for each value_KS in self.listofKS
thred
value_KS.process();
end thred;
end for;
```

Obr. 2.30: paralelné spustenie rozumových zdrojov

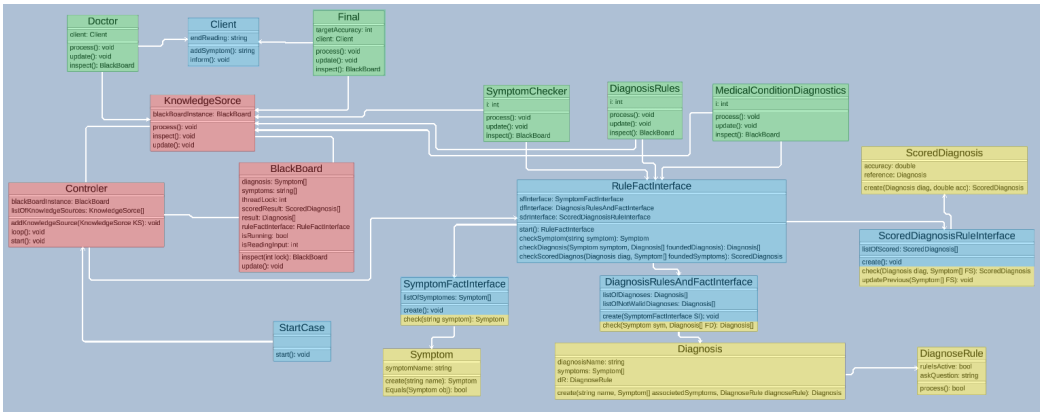
Na záver prezentujem spôsob implementácie metódy *process* pre rozumové zdroje v oboch scenároch (OAL). Ako môžeme vidieť na obrázku 2.31, spôsob vykonávania sa výrazne nezmenil. Hlavným rozdielom je to, že pri paralelnom spracovaní je telo metódy obalené v nekonečnom cykle. Týmto zabezpečíme naozajstnú náhodnosť spúšťania a vykonávania metódy *process*.

```
// pri linearnom volani funkcie process
for each assumption in blackBoardinstance:
if(assumption == "rule")
...
end if;
...
end for;

// pri paralelnom volani funkcie process
while (1)
for each assumption in blackBoardinstance:
if(assumption == "rule")
...
end if;
...
end for;
end while;
```

Obr. 2.31: Implementácia metódy process pri lineárnom vs. pri paralelnom spúšťaní.

2.4.3 Iterácia 3 - Systém na detekciu chorôb

Na záver implementačného procesu sme sa opäť vrátili k medicínskemu systému, keďže AnimArch nepodporoval isté skupiny stringových operácií, ktoré boli potrebné počas predošlej implementácie štýlu BB. Tento expertný systém, ktorý sme implementovali s využitím architektonického štýlu **BB**, bližšie opíšeme z pohľadu tried. Jeho hlavné komponenty sú Controller, BlackBoard a KnowledgeSource (abstraktná trieda), ktoré môžeme vidieť na obrázku 2.32 označené červenou farbou.



Obr. 2.32: Triedny diagram systém na detekciu chorôb

Funkcionalita hlavných komponentov bola opísaná v 2.3. V našom systéme sme sa zameriavali na hlavnú funkcionalitu tohto štýlu, ktorou je podpora multithreadingu (využitie viacerých rozumných zdrojov KS paralelne) s cieľom dosiahnuť náš cieľ. V našom príklade máme niekoľko tried, ktoré implementujú abstraktnú triedu KnowledgeSource, čo je možné vidieť na obrázku 2.32 označenom zelenou farbou. Táto abstraktná trieda má jednu vlastnú premennú *blackBoardInstance*, ktorej je priradená inštancia BB. Teraz sa pokúsime opísať hlavnú funkcionalitu jednotlivých **KS** z pohľadu tried:

Trieda **Doctor** vyvoláva triedu **Client** na pridanie chorobných príznakov do tabule.

SymptomChecker validuje chorobné príznaky typu *string* od používateľa a konvertuje ich na faktory typu *Symptom*, následne ich ukladá do tabuľky v zozname *diagnosis* ako referencie na objekty.

DiagnosisRules vyhľadáva nájdené symptómy a následne hľadá v existujúcich diagnózach, či prienik medzi symptómami a symptómami diagnóz nie je prázdna množina. Ak nie je, pridáva ich do tabuľky v zozname *result*.

MedicalConditionDiagnostics z nájdených diagnóz vytvára ohodnotené diagnózy. Na základe prieniku medzi symptómami, ktoré má **Client**, a symptómami nájdených diagnóz vytvorí nové objekty typu *ScoredDiagnosis*.

Final - ktorý vyhľadáva, či existuje nejaká ohodnotená diagnóza (*ScoredDiagnosis*), ktorej ohodnotenie spĺňa počiatočné podmienky zadané používateľom.

Na obrázku 2.32 označenom modrou farbou je možné vidieť triedy, ktoré sú pomocného typu a umožňujú nám spúšťať animáciu **Startcase**. Ostatné triedy s koncovkou Interface, ako sú *RuleFactInterface*, *SymptomFactInterface*,..., nám umožňujú vytvárať pravidlá a poznámkové fakty, ku ktorým budú pristupovať jednotlivé KS podľa potreby. Pod žltou farbou môžeme vidieť triedy a metódy, ktoré sú reprezentované a kontrolované samotnými faktami a pravidlami.

OAL kód: V tejto pod časti sa budeme zaoberať implementáciou OAL kódu. Na úvod vysvetlíme spôsob ako je implementovaná trieda Controller. Samotná trieda Controller má viacero metód, ktoré bolo nutné implementovať, ale kľúčové pre nás sú metódy loop a start. Na obrázku 2.33 môžeme vidieť, že táto metóda nastavuje všetky potrebné prvky, ktoré sú nutné na beh animácie.

Celý systém sa spúšťa na konci *self.start()*, ktorej implementáciu je možné vidieť na obrázku 2.34. Pričom komentár *par* umožňuje spustenie vlákien, ktoré sú v našom prípade jednotlivé KS. Celková implementácia vlákna je veľmi jasná. *Thread* umožňuje spustenie metódy objektu paralelne. Potom nasleduje metóda, ktorú chceme paralelne vykonávať. V našom prípade je to metóda procesu každého KS *ks0.process();*, a následne ukončenie behu vlákna pomocou *end thread;*. Na záver treba ešte ukončiť paralelizmus príkazom *end par;*.

Ďalej priblížime triedu BlackBoard a jej metódy *update()* a *inspect(int lock)*.

```
create object instance bb of BlackBoard;
self.blackBoardInstance = bb;
create object instance client_1 of Client;
client_1.endReading = "";
//Vytvorenie a nastavenie jednotlivých KS
create list self.listOfKnowledgeSources of KnowledgeSorce;
create object instance KS_Doctor of Doctor;
KS_Doctor.client = client_1;
...
// Nastavenie vlastných atribútov.
self.blackBoardInstance.threadLock = 0;
...
// Nastavenie požadovanej presnosti pre hľadanie diagnózy.
targetAcc = integer(read("Set Targe accuracy <0,100>"));
KS_Final.targetAccuracy = targetAcc;
less = targetAcc >= 0;
max = targetAcc <= 100;
if( less AND max)
    self.start();
end if;
```

Obr. 2.33: OAL kód Controler.loop()

```
par
thread
ks0.process();
end thread;
thread
ks1.process();
...
end par;
```

Obr. 2.34: OAL kód Controler.start()

V prípade metódy *inspect(int lock)* môžeme vidieť kód na obrázku 2.35. Táto metóda je kľúčová pre správnu funkčnosť vlákien, keďže jednotlivé vlákna chcú pristupovať k tabuľke a zapisovať do nej. Podmienka v prvom riadku kódu nám hovorí o tom, že ak je atribút *threadLock* rovný 0, tak zápis parametra metódy *lock* sa nastaví ako aktuálny vlastník tabuľky.

A ďalšou kľúčovou metódou je metóda *update()*, ktorá je uvedená na obrázku

```
if (self.threadLock == 0)
    self.threadLock = lock;
end if;
return self;
```

Obr. 2.35: OAL kód BlackBoard.inspect()

2.36. Tu nuluje dva atribúty triedy BlackBoard. Prvým je *threadLock* a druhým je *isReadingInput*, ktoré sa zamykajú v jednotlivých **KS**, v prípade potreby čítania vstupu z konzoly. Aktuálna implementácia vlákien v Animarch nepodporuje čítanie len pre jedno vlákno, ale pre všetky, čo môže spôsobiť problém v animácii.

```
self.threadLock = 0;
self.isReadingInput = 0;
```

Obr. 2.36: OAL kód BlackBoard.update()

Teraz sa pozrieme na jednu implementáciu **KS**. Princíp implementácie je rovnaký pre všetky, aj keď ich funkcionalita je odlišná. Hlavný princíp zapisovania a pozerania sa na **BB** je rovnaký. Na obrázku 2.37 sú vidieť 3 metódy: *process*, *update*, *inspect*. Ako prvé opíšeme metódu *inspect()*, ktorá sa pokúša zapísať do tabuľky svoj unikátny kľúč na objekt typu *BlackBoard*. V tomto prípade ide o 1. Následne metóda *update* volá metódu *update* na *BlackBoard*, ktorá zasa vynuluje unikátny kľúč nachádzajúci sa v tabuľke. Metóda *process()* je jedinečná a odlišná medzi každým rozumným zdrojom. Ale kľúčové prvky sú vždy rovnaké, keďže ide o prácu s vláknami a jednotlivé príkazy v tejto funkcii chceme vykonávať potenciálne donekonečna, tak metóda začína while cyklom, pričom podmienky ukončenia sú za slovom while. Následne sa pokúšame získať obsadenie tabuľky cez príkaz *self.inspect()*. V prípade, že je to možné, atribút *threadLock* na tabuľke sa zmení na unikátnu hodnotu. V tomto prípade sme museli kvôli prístupu k zápisu do konzoly pridať príkaz *isReadingInput* nejakú hodnotu, keďže nechceme dovoliť iným KS, ktoré môžu chcieť vstup od používateľa, prístup k písaniu. V ďalšom

príkaze pridáme do tabuľky to, čo používateľ napísal do konzoly, a updatujeme objekt *BlackBoard*. Výhodou tejto implementácie je, že sa môže veľmi ľahko modifikovať funkcionalita metódy *process* a upraviť podľa požadovaných vlastností

```
// process()
while (True AND self.client.endReading  != "end")
bbi = self.inspect();
if(bbi.threadLock == 1 AND self.client.endReading  != "end" AND bbi.isReadingInput == 0)
    bbi.isReadingInput = 1;
    add self.client.addSymptom() to bbi.symptoms;
    bbi.update();
end if;
wait for 0.1 seconds;
end while;

// update()
self.blackBoardInstance.update();

// inspect()
return self.blackBoardInstance.inspect(1);
```

Obr. 2.37: Implementácia OAL kódu rozumového zdroja triedy Doctor a metód process(), update(), inspect()

Túto vlastnosť sme si otestovali pri požiadavkách od vedúceho práce na triede **DiagnosisRules**, ktorej implementáciu je možné vidieť na obrázku 2.38. Tento obrázok je útržok kódu bez časti kódu, ktorá je zhodná s *Doctor.process()*.

V pôvodnej časti sme sa pozreli na diagnózy, ktoré boli nájdené, a v prípade, že sme nejaké našli, sme sa pozreli na tabuľu, či sa už na nej nachádzajú. Ak nie, tak sme ich tam pridali. V tejto časti prišli požiadavky, aby pravidlá neboli iba faktami s Prologom, ako napríklad *diagnose(symptom(headache),symptom(fever),cold)*. Mali by sme byť schopní, volať aj ďalšie pravidlá v závislosti od nájdených diagnóz. Toto pochopíme z Prologu ako *diagnose(symptom(headache), symptom(fever),cold) :- t1(L), L = true.* Pravidlo t1(L) predstavuje čítanie z konzoly a priradenie do L, teda v Prologu *t1(L):- read(L).* V závislosti od toho, čo používateľ zadal (1 alebo 2, teda true/false), sa pridá diagnóza na tabuľu alebo sa zaradí do zoznamu diagnóz, ktoré už nechceme kontrolovať. V prípade, že odpovieme na všetky diagnózy nie, proces prehľadávania sa ukončí skôr, čím šetríme výpočtový čas behom celého programu. Toto je jedna z hlavných výhod architek-

```
count = bbi.diagnosis.Count();
index = self.i - 1;
if(count > index)
    resultOfCheckDiagnosis = bbi.ruleFactInterface.checkDiagnosis(bbi.diagnosis[index],bbi.result);
    for each diagnoseResult in resultOfCheckDiagnosis
        foundedDiagnose = diagnoseResult;
        contains = bbi.result.Contains(foundedDiagnose);
        notContains = not contains;
        if (notContains AND foundedDiagnose.diagnosisName != "null")
            add foundedDiagnose to bbi.result;
        end if;
    end for;
    self.i = self.i + 1;
end if;
// Zmena voči originálnej verzii
countMAX = bbi.ruleFactInterface.dfInterface.listOfDiagnoses.Count();
countMAX = countMAX - 1;
countActual = bbi.ruleFactInterface.dfInterface.listOfNotWalidDiagnoses.Count();
if(countMAX == countActual)
    write("Sorry, we have not been able to find the cause of your diagnosis.");
    self.blackBoardInstance.isRunning = False;
end if;
```

Obr. 2.38: Implementácia OAL kódu rozumového zdroja triedy DiagnosisRules pre metódu process()

tonického štýlu **BB**.

Ako posledné, máme ešte OAL kód pre pravidlá. V tomto expertnom systéme sú jednotlivé pravidlá reprezentované triedami *Symptom, Diagnosis, DiagnoseRule, ScoredDiagnosis* a príslušnými metódami *SymptomFactInterface.check(), DiagnosisRulesAndFactInterface.check(), ScoredDiagnosisRuleInterface.check() a .updatePrevious()*, ktoré spúšťajú, kontrolujú a vytvárajú jednotlivé objekty. Teraz opíšeme niektoré z nich.

SymptomFactInterface.check() je metóda, ktorú možno vidieť na obrázku 2.39. Z obrázka opíšem len časti, ktoré sú potrebné pre pochopenie funkcionality, a špeciálne príkazy začínajúce s # opíšeme neskôr. Z obrázka je vidno prechádzanie cez zoznam objektov *self.listOfSymptoms*, pomocou príkazu *for each*. Počas behu príkazu *for each* kontrolujeme, či sa nejaký objekt typu Symptom rovná inému. Toto je zabezpečené špeciálnou metódou **Equals**, ktorá sa môže prepísať podľa potreby. V našom prípade sme ju upravili na porovnávanie podľa podmienky *return self.symptomName == otherSymptom.symptomName*. Na záver tejto časti kódu po príkaze *for each* je skupina príkazov, ktoré zabezpečia, že metóda niečo

vráti, keďže AnimArch nepozná návratovú hodnotu null alebo undefined, ale musí dostať nejaký objekt na výstup. Preto všetky pravidlá majú základnú návratovú hodnotu, a to posledný prvok v zoznamoch, pre *self.listOfSymptoms* je *Symptom('null')*.

```
#NoAnim
#nonewobjs
create object instance jsutSymptom of Symptom;
jsutSymptom.create(symptom);
#donewobjs
for each symptom_db in self.listOfSymptomes
    equals = jsutSymptom.Equals(symptom_db);
    if (equals)
        #DoAnim
        return symptom_db;
    end if;
end for;
#DoAnim
countOfSymptoms = self.listOfSymptomes.Count();
endOfList = countOfSymptoms - 1;
return self.listOfSymptomes[endOfList];
```

Obr. 2.39: Implementácia kontroly faktov (triedy Symptom) v OAL

ScoredDiagnosisRuleInterface.check() v tejto časti sa pozrieme na pravidlá, ktoré nemusia byť len faktami, ako v prípade symptómov opísaných vyššie. Objekty typu **Diagnosis** sú tvorené ľavou a pravou časťou. Ak je hlava pravidla platná, potom voláme ľavú stranu pravidla, ktorá predstavuje ďalšie pravidlo. V OAL kóde to môžeme vidieť na obrázku 2.40, kde je hlava pravidla zoznam symptómov a pravá strana predstavuje ďalšie pravidlo, ktoré, keď je platné, môže byť pridané do tabuľky. Tento proces vykonania a overenia pravidla *T1* sa deje v metóde *check()*, ktorú možno vidieť na obrázku 2.41. Chod programu v OAL kóde môžem v skratke opísať takto: Ak nájdeme prienik symptómov užívateľa a symptómov diagnózy, potom sa pozrieme, či je pravidlo na pravej strane aktívne (*self.listOfDiagnoses[i].dR.ruleIsActive*). Ak je, potom skontrolujeme, či *dR.process()*

== *T1*, a ak je odpoveď od užívateľa true, tak ho pridáme do *returnListOfDiagnoses*, inak ho pridáme do zoznamu *listOfNotValidDiagnoses*, aby sa nabudúce už nekontroloval.

```
headache = SI.check("headache");
fever = SI.check("fever");

// telo pravidla
// pravidlo t1(L) :- read(L).
create object instance diagnoseRule1 of DiagnoseRule;
diagnoseRule1.ruleIsActive = True;
diagnoseRule1.askQuestion = "T1 - We have to swab throat for more information?";

// pravidlo diagnose(symptom(headache),symptom(fever),Cold) :- t1(L), L = true.
// hlava pravidla
create object instance Cold of Diagnosis;
create list x of Symptom {headache, fever};
add Cold.create("cold", x,diagnoseRule1) to self.listOfDiagnoses;
```

Obr. 2.40: Tvorba pravidiel Diagnosis

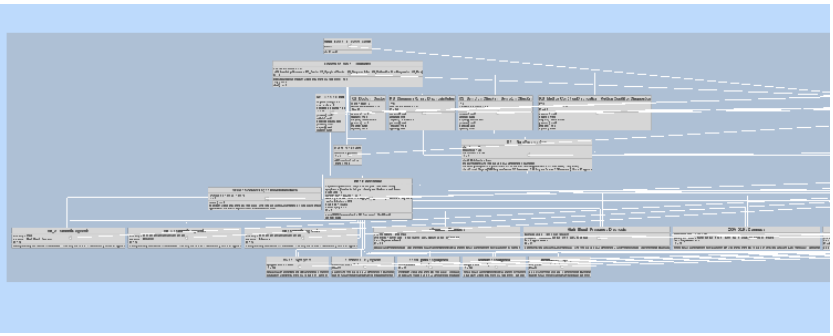
```
#NoAnim
create list returnListOfDiagnois of Diagnosis;
i = 0;
for each diagnose in self.listOfDiagnoses
    containsSymptom = diagnose.symptoms.Contains(sym);
    containsDiagnose = FD.Contains(diagnose);
    invertedContainsDiagnose = not containsDiagnose;
    if (containsSymptom AND invertedContainsDiagnose)
        containsDiagnoseRule = self.listOfNotWalidDiagnoses.Contains(diagnose);
        invertedContainsDiagnoseRule = not containsDiagnoseRule;
        if(self.listOfDiagnoses[i].dR.ruleIsActive AND invertedContainsDiagnoseRule)
            checkPrecondition = self.listOfDiagnoses[i].dR.process();
            invertedCheckPrecondition = not checkPrecondition;
            if(invertedCheckPrecondition)
                add self.listOfDiagnoses[i] to self.listOfNotWalidDiagnoses;
            else
                add self.listOfDiagnoses[i] to returnListOfDiagnois;
            end if;
        else
             add self.listOfDiagnoses[i] to returnListOfDiagnois;
        end if;

    end if;
    i = i + 1;
end for;
#DoAnim
return returnListOfDiagnois;
```

Obr. 2.41: Implementácia kontroly pravidiel Diagnosis v OAL

Problémy s implementáciou: V tejto časti sa budeme venovať kľúčovým problémom, s ktorými sme sa potýkali počas implementácie expertného systému.

Príliš dlhý beh animácie bol jedným z problémov, keďže každý príkaz sa animoval s minimálnou rýchlosťou 0,1 sekundy, čo spôsobilo, že animácia s približne 1800 príkazmi trvala 3 alebo viac minút. Keďže táto animácia mala v závere aj viac ako 3000 vykonaných príkazov, bolo nutné nejako urýchliť vykonávanie. Našli sme dva spôsoby, ako riešiť tento problém. Prvým bolo pridanie ďalších špeciálnych príkazov: **#NoAnim** na zrušenie animovania a **#DoAnim** na znovu spustenie animácie. Dôvodom implementácie týchto príkazov bolo zbytočné sledovanie niektorých kontrolných metód, ktoré sú nezaujímavé a nijako neovplyvňujú hlavný chod animácie. Taktiež nám to prinieslo zrýchlenie približne o 25%, čo bolo nevyhnutné pre ďalší vývoj.

Veľkosť objektového diagramu: Tento problém, ktorý vznikol, si môžeme pozrieť na obrázku 2.42, na ktorom je vidieť iba jednu pätinu objektovej vrstvy diagramu. Tento problém vznikol, keď sme sa rozhodli mať 20 rôznych symptómov a 10 rôznych diagnóz, čo nás priviedlo k extrémnemu množstvu objektov v objektovej vrstve diagramov.



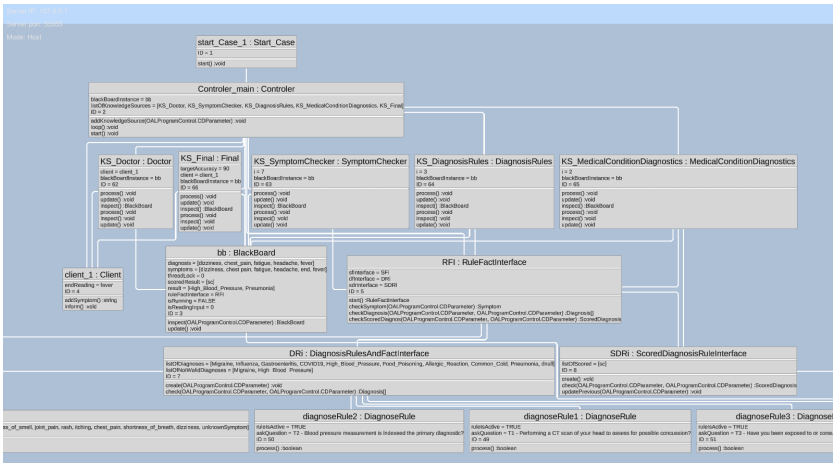
Obr. 2.42: Antipattern objektový diagram

Riešenie tohto problému požadovalo pridanie špeciálnych príkazov: **#nonewobjs**, čo zrušilo vytváranie nových objektov na objektovej vrstve diagramu, a **#donewobjs**, ktorý opäť povolil vytváranie nových objektov na tejto vrstve. Týmto spôsobom sme dokázali ďalej urýchliť vývoj tejto animácie.

Referencie na objekty: V tomto probléme išlo o to, že nám jednotlivé objekty uložené v zoznamoch reprezentovali iba ich *ID*. Tento problém bol vyriešený zmenou reprezentácie na meno objektu pri vytváraní. To znamená, že ak vytvárame objekt takto: *create object instance COVID19 of Diagnosis;*, tak meno bude **COVID19**.

Maximalizovanie zrýchlenia animácie: Keďže animácia bola už dosť rýchla, ak počet diagnóz a symptómov nebol príliš veľký, trvala asi 1,5 minúty. V prípade, že sme sa rozhodli spustiť animáciu so všetkými 20 symptómami a 10 diagnózami, čas sa stále zvyšoval, a počet vykonaných príkazov mohol súbežne dosiahnuť až 20 000. Preto sme potrebovali nejakým spôsobom ďalej zrýchliť spôsob animovania. Tento cieľ sme dosiahli pridaním špeciálneho nastavenia v C# skripte, pričom animovanie jedného príkazu sa zo 0,1 sekundy skrátilo na 0,00001 sekundy. Táto hodnota sa dala zmeniť podľa potreby v C# skripte.

Po všetkých zmenách sme získali objektový diagram, ktorý je vidieť na obrázku 2.43. Na ňom sú zobrazené len potrebné časti pre vysvetlenie funkcionality **BB** architektonického štýlu.



Obr. 2.43: Finálna verzia objektového diagramu pre 20 symptómov a 10 diagnóz

3 Vyhodnotenie a výsledky

Táto kapitola sa zameriava na celkové zhodnotenie dosiahnutých výsledkov. Preskúmali sme, či naša práca posunula AnimArch ďalej vpred a či má ďalší vývoj tejto aplikácie zmysel.

Naša diplomová práca bola zameraná na skúmanie možností OAL jazyka v aplikácii Animarch a slúžila na overenie, či by výučba s využitím tohto softvéru bola možná. Výsledky práce jasne naznačili, že tento cieľ ešte nie je dosiahnuteľný. Preto sme si položili otázku, či je OAL jazyk v Animarchu schopný a dostatočne flexibilný na použitie, a zrýchlenie vývoja komplexnejších štrukturálnych a dynamických modelov alebo v našom prípade architektonických štýlov.

Na záver sme sa pozreli na využitie OAL pri generovaní zdrojového kódu v jazyku Python. Evaluácia prebehla pomocou techniky RJT (Relevance Judgment Technique), pričom princípom tejto metódy je testovať algoritmus z hľadiska správnosti odpovedí, či sú v súlade s očakávaným výstupom testovacieho objektu. V našom prípade sme testovali animáciu a samotný scenár, ktorý bol implementovaný, a zároveň sme sa pozerali na správnosť kódu, ktorý bol vygenerovaný.

3.1 Zhodnotenie použiteľnosti softvéru AnimArch

V nasledujúcich častiach sa budeme bližšie zaoberať výhodami a nevýhodami a ako môžu ovplyvniť používanie softvéru AnimArch v praktických projektoch.

3.1.1 Výhody softvéru Animarchu

Začneme prvými pozitívami, ktoré sme objavili pri tejto aplikácii. Jednou z významných pozitív je schopnosť pozorovať životný cyklus objektov a tried, čo je vhodné pre hlbšie pochopenie správania programu. V rámci tohto bodu mi napadli aj jednotlivé príklady, ktoré som sa pokúsil implementovať, hoci niektoré boli neúspešné. Medzi tie, ktoré by mali väčší význam, by určite patrilo pravidlo expertných systémov pre jazyk Prolog. Toto by mohlo byť veľmi kľúčovým vzdelávacím prvkom pri výučbe logiky.

Ďalším kľúčovým benefitom je možnosť importovať .xml súbory z aplikácie *Enterprise Architect*, ktorú som využíval hlavne na začiatku, keďže práca s EA bola jednoduchšia ako v AnimArchu. Problémom EA je to, že človek musí venovať veľa úsilia pochopeniu tohto systému pre vývoj grafov z hľadiska veľkosti a možností, ktoré ďaleko presahujú možnosti AnimArchu. Ich pochopenie si vyžaduje oveľa rozsiahlejšie znalosti ako AnimArch.

Sila jazyka OAL v AnimArchu vo mne zo začiatku vyvolávala veľké obavy, že to nepôjde, no čas všetko zmenil. V priebehu tejto diplomovej práce OAL jazyk v AnimArchu dosiahol novú úroveň, o ktorej som na začiatku mohol len snívať. Pribudli nám nové možnosti, ako sú:

1. Reťazcové operácie - split, join, at,...
2. Operácie s listami - count, contains a indexOf
3. Špeciálne príkazy - #NoAnim, #DoAnim, #nonewobjs, #donewobjs
4. Špeciálne funkcie - prepisovateľné (overridable)

Operačné funkcie nad reťazcami som nepoužil, z dôvodu oneskorenej implementácie, a bol som teda nútený opustiť moju myšlienku interpretovaného jazyka Prolog alebo spracovania textu pomocou architektonického štýlu P&F. No jedno je isté, všetky ostatné operácie som plne využil a boli veľmi nápomocné pri riešení tejto práce.

Maskovanie tried je jednou z noviniek, ktorá umožňuje modifikáciu názvov jednotlivých triednych objektov, ktoré sa vytvárajú. Túto možnosť som nevyužil, ale viem si predstaviť jej benefity pri ďalšom vývoji.

Chybové hlásenia a opravy uloženia znejú ako jednoduchá vec, ale kvôli tomu, že táto aplikácia žije a je stále vyvíjaná medzi rukami našich študentov, tak niekedy počas vývoja vznikla chyba pri ukladaní animácie a proces ukladania bol veľmi náročný a nekonzistentný. Navyše, bolo nutné kontrolovať .json súbor animácie, aby sme overili jeho konzistentnosť a funkčnosť, resp. dôvod nefunkčnosti animácie. Tento problém bol zväčša vyriešený a vylepšený pridaním chybového hlásenia v prípade nefunkčného OAL kódu.

3.1.2 Nevýhody softvéru Animarchu

Treba zároveň spomenúť aj túto časť, ktorá ma síce mrzí, ale je to fakt, ktorý bude ťažké odstrániť.

Práca v softvéri nie je úplne jasná a sú v nej chyby. Počas práce sme narazili na drobné chyby, ktoré nie je úplne jednoduché odstrániť, pretože ich pôvod nevieme presne identifikovať kvôli komplexnosti kódu. Jednou z možností chyby je prekliknutie cez UI. Táto chyba bola a stále čiastočne pretrváva, vyskytuje sa, keď klikneme do priestoru na písanie OAL kódu. Ak klikneme na triedu pod týmto UI elementom (ak sa tam nachádza), označíme triedu na písanie OAL kódu, ktorá sa nachádzala pod ňou, čo spôsobuje chaos v používaní.

Ďalšou nevýhodou je príliš veľké priblíženie objektového diagramu k diagramu tried. Ak chceme sledovať niečo na objektovom diagrame a chceme sa posunúť ďalej aby sme videli všetko, tak sa presunieme ponad triedny diagram. Tento efekt je nežiadúcim efektom odzoomovania z objektovej vrstvy.

Neprehľadnosť objektových diagramov: keďže svet objektov môže byť veľmi veľký, tak je veľmi ľahké sa stratiť v takomto diagrame, najmä ak je veľký, a tým pádom sa na týchto diagramoch nedá vyčítať ich funkčnosť, pretože používateľ nedokáže sledovať všetko, čo sa deje.

V neposlednom rade sú tu ešte drobnosti, ale drobnosti robia kvalitu života a

dobrý softvér ako taký. Preto spomeniem iba to, že niektoré funkcionality, napríklad načítanie diagramu, by sa mali dať spustiť viackrát, v prípade ak načítame zlý diagram, to isté platí pre animácie. Okrem toho by sa spúšťanie animácie malo dať robiť niekoľkokrát za sebou bez toho, aby bol používateľ nútený znovu načítavať všetky súbory. Spúšťanie animácie niekoľkokrát za sebou je už možne v poslednej verzií AnimArch.

3.2 Evaluácia výslednej animácie a vygenerovaných python kódov pre štýl P&F

V tejto časti si zhrnieme dôležité časti a pozorovania, ktoré sme nadobudli pri evaluácii štýlu P&F, a k akým chybám sme dospeli týmto pozorovaním.

3.2.1 Výsledná animácia z Animarchu pre štýl P&F

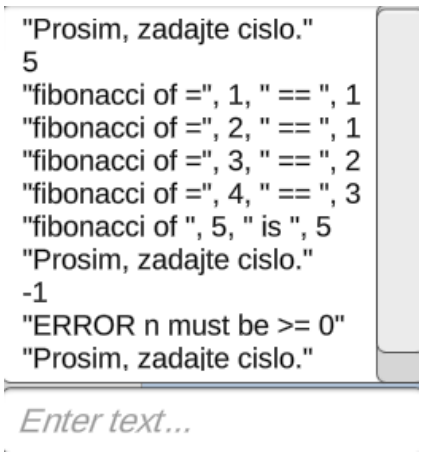
Na úvod je potrebné spomenúť, že príklad Fibonacciho postupnosti bol veľmi jednoduchým, ale zároveň veľmi pekným príkladom, a taktiež bol veľmi ľahko testovateľný. Pri pozorovaní behu tejto animácie sme si všimli nedostatky tohto štýlu, ako bolo zistené v predchádzajúcej časti 2.1.1.

Čo sa týka nadbytku **nadbytok režijného výdaja (Overhead)**, volania pre získavanie dát z jednotlivých dátových potrubí a ich následné spracovanie, predstavuje zbytočnú komplexnosť systému, ktorú by sme vedeli odstrániť zreťazovaním filtrov. Avšak, týmto by sme mohli prísť o niektoré vlastnosti, ktoré sme touto implementáciou získali, hlavne možnosť **rozpojiteľnosti (Decopuling)**.

Z pohľadu **komplexnosti (Complexity)** sme zistili, že jednotlivé komponenty musia mať veľa nadbytočného kódu a zbytočne zložitú štruktúru, s ktorou pracujú. Chcel by som poukázať na znovupoužiteľnosť alebo jednoduchosť zmeny jednotlivých filtrovacích komponentov. V prípade implementácie, ktorej návrh môžeme nájsť na obrázku 2.15, by implementácia vyzerala optimálne pre konkrétny prípad použitia. V prípade zmien v prenosoch dát by vznikol problém nutnosti opraviť

všetkých filtrov pre ich správnu funkčnosť.

Evaluácia prebehla na finálnej implementácii, kde sme si načítali class diagram a OAL kód, ktoré viete nájsť v prílohách diplomovej práce. Po načítaní a spustení sme sledovali správnosť vykonávania príkazov, pričom rýchlosť bola nastavená tak, aby používateľ bol schopný sledovať to, čo sa deje na objektovej vrstve. Toto bolo vhodné pre *Fibonacci(5)*, potom už animácia trvala dlho, čo by bolo náročné na sledovanie. Pri pozorovaní sme zistili, že čísla menšie ako 0 nie sú vhodné pre beh tohto programu. Pre otestovanie programu sme skúsili aj vyššie čísla a výpočet prebiehal správne a bez problémov. Tento priebeh je zachytený v konzole, ktorú môžete vidieť na obrázku.



Obr. 3.1: Výpis v výsledkov Fibonacciho postupnosti piatej úrovne v konzole AnimArch

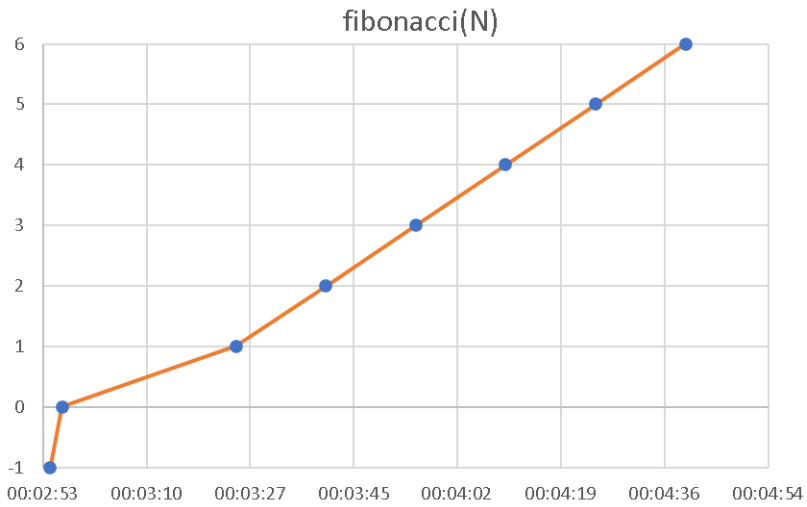
Pre overenie možností a schopností softvéru AnimArch sme chceli spustiť tento systém aj s vysokými rýchlosťami, pričom sme chceli sledovať, ako sa bude systém správať a či bude počítať správne. Nevieme presne, prečo, ale táto animácia nebola schopná bežať v tomto režime, padala s chybovými hláseniami. Preto túto časť evaluácie hodnotím ako zlyhanie systému AnimArch, i keď chyba môže byť spôsobená tým, že táto časť bola vytvorená v čase, keď veľmi rýchly režim nebol ešte prístupný.

Na záver sa pozrieme na výsledky, ktoré možno vidieť na obrázku 3.2, a vysvetlíme si, prečo ukážka samotného výpočtu je tak dlhá. Pod *modrou* farbou je funkcia fibonacci(N), ktorú sme testovali, a pozorovali sme časy, za ktoré animácia vedela byť vykonaná pri rýchlosti, ktorú dokáže sledovať používateľ(0,8) kroku na jeden riadok OAL kódu. Ako možno vidieť z obrázku, pre *N == -1* bol čas takmer 3 minúty. Tento čas je pre nás mierkou toho, za aký čas sme dokázali vygenerovať infraštruktúru, ktorá dokáže počítať Fibonacciho postupnosť. Pre *N == 0* sa čas veľmi nezmenil, keďže filter, ktorý sčítava, berie *N >= 1*. Pre ostatné *N >= 1* sa čas výpočtu konštantne zvyšoval o 15 sekúnd pre každé ďalšie N.

fibonaci (N)	Time	Prediction	Resoult
test(-1)	2:54s	Error	Error
test(0)	2:56s	0	0
test(1)	3:25s	1	1
test(2)	3:40s	1	1
test(3)	3:55s	2	2
test(4)	4:10s	3	3
test(5)	4:25s	5	5
test(6)	4:40s	8	8

Obr. 3.2: Overenie výsledkov Fibonacciho postupnosti s koeficientom 0.8 pre vykonanie jedného OAL príkazu

Zároveň uvádzame aj obrázok grafu 3.3, na ktorom sú jednotlivé výsledky ľahšie pochopiteľné. Pri tomto pozorovaní sme sa pokúsili zistiť, prečo čas narastá lineárne. Skúšali sme to dokázať výpočtom *Celkový počet príkazov OAL (55)* $\times$ *0.8*, čo by malo predstavovať 44 sekúnd, ale v skutočnosti sme zistili 15 sekúnd. Pri sledovaní animácie sme zistili, že niektoré časti kódu nie sú animované, ale predstavujú len kód medzi týmito podmienkami a cyklami. Po úprave vzorca na výpočet len animovaných príkazov (18) sme dosiahli výpočet času 14,4 sekundy, čo sa výrazne priblížilo k očakávanému výsledku.



Obr. 3.3: Graf výpočtu Fibonacciho postupnosti v čase

3.2.2 Evaluácia vygenerovaného zdrojového kódu v jazyku Python vzhľadom na korektnosť pre štýl Pipes and Filters

Po evaluácii animácie sme sa posunuli k evaluácii Python kódu. Tento kód bol vygenerovaný z OAL kódu, ktorý sme napísali. Najprv sa pozrieme na chyby, ktoré sme objavili po vygenerovaní Python kódu. A po oprave chýb sa pustíme do samotnej evaluácie kódu.

Jednou z chýb, ktoré sme objavili, je chyba parsera OAL kódu. V prípade prvej chyby, ktorá je uvedená na obrázku, je zjavné kde problém nastal, keďže premenná sa volá rovnako ako trieda. Tento problém bolo veľmi ľahké definovať a opraviť, takže treba do budúcna implementovať ochranný proces buď do OAL kódu alebo parsera pre eliminovanie tejto chyby.

V prípade tejto druhej chyby je už problém vzniku trochu ťažšie definovať, a ostáva nám možnosť len polemizovať nad tým, prečo vznikla. Ako je možné vidieť z obrázku, chýbali nám tam parametre metódy *n* a *self* pre triednu premennú. Po prvé, táto chyba mohla vzniknúť, keď sme pracovali na tejto implementácii. Keďže v začiatkoch práce bolo nutné zasahovať do zdrojových kódov

```
// Zlý kód
Client= Client()
Client.create()
// Opravený kód
c = Client()
c.create()
```

Obr. 3.4: Problém v triednej metóde StartCase.start() pre animáciu P&F

OAL animácie v prípade chýb. Tento problém bol ale odstránený počas vývoja práce. Preto som skúsil nanovo vygenerovať OAL kód a následne aj nanovo vygenerovať Python kód, avšak tento kód aj tak pretrval.

```
// Zlý kód
def showData(self):
    if n >= 0:
      print("fibonacci of ", n, " is ", data_out.data)
    else:
      print("ERROR n must be >= 0")
// Opravený kód
def showData(self, n):
    if n >= 0:
      print("fibonacci of ", n, " is ", self.data_out.data)
    else:
      print("ERROR n must be >= 0")
```

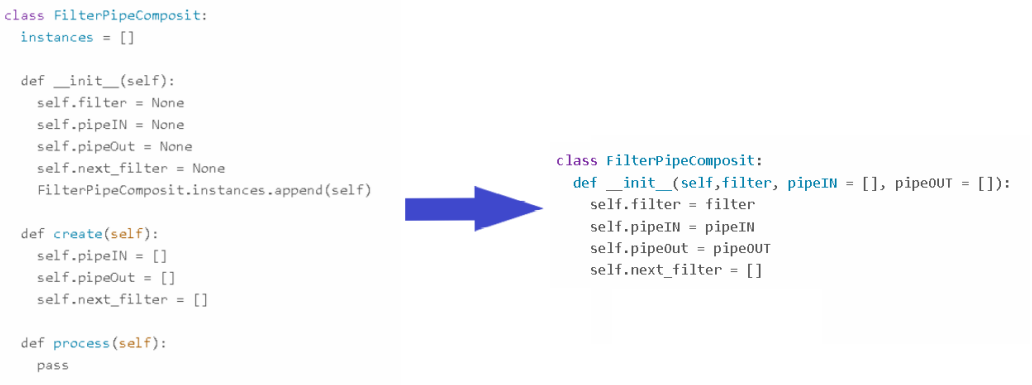
Obr. 3.5: Problém v triednej metóde Sink.showData() pre animáciu P&F

3.2.3 Evaluácia metód Python kódu pre štýl Pipes and Filters vzhľadom na funkcionalitu

Evaluácia vygenerovaných metód pre štýl Pipes and Filters v tomto prípade sa pozrieme na samotné skladanie filtrov v triede *Pipeline.create()* a zároveň na súvisiacu triedu *FilterPipeComposite*, ktoré nie je úplne najlepšie transformovaná

a jej funkčnosť je veľmi neprehľadná. Aj keď by som rád ukázal, ako sa dá transformovať a zjednodušiť zápis komplexnej funkcie *Pipeline.create()*, kód je príliš veľký a jeho transformáciu budete vedieť nájsť v prílohe 3.3.5. V rámci toho, čo môžete nájsť v prílohe, je aj obrázok „priloha_evaluacia_p_and_f_OAL_GENERATED.png“ a nasledujúci obrázok „priloha_evaluacia_p_and_f_BETTER_WAY.png“. V týchto obrázkoch je vidieť kľúčový rozdiel prístupu v programovaní v prostredí OAL a Python. Pričom v Python kóde generovanom s OAL ani nevieme, že sa jedná o nejakú implementáciu **Pipes and Filters**. Keď sa potom pozrieme na obrázok *..BETTER_WAY.png*, tak štruktúrovaním kódu dosiahneme lepšiu čitateľnosť a porozumenie.

Teraz sa pozrieme na to, ako sa dá transformovať trieda *FilterPipeComposite*. Na obrázku 3.6 je možné vidieť hneď niekoľko chýb, ktoré by nespravil ani programátor začiatočník. Začnem najprv veľmi zjavnými vecami a to, že ak metóda triedy je prázdna *FilterPipeComposite.process()*, tak by sa ani nemala generovať. Hlavným rozdielom je aj to, že pri generovaní sa v kóde k všetkým atribútom priradí hodnota *None*. Toto je spôsobené tým, že pri generovaní kódu nevyužívame informácie z Triedneho diagramu, kde poznáme jednotlivé typy atribútov triedy.



Obr. 3.6: Porovnanie vygenerovaného a očakávaného zdrojového kódu pre triedu FilterPipeComposit

3.3 Evaluácia výslednej animácie a vygenerovaných python kódov pre štýl Blackboard a ich vzájomné porovnanie

V tejto časti sa zameriame na vyhodnotenie funkčnosti animácie v časti 2.4.3, v ktorej sme implementovali expertný pravidlový systém na určovanie chorôb. Pod slovom ëxpertný pravidlový systém"nemáme na mysli systém ako taký, keď že uvedený príklad animácie nie je v pravom slova zmysle expertný, keď že určovanie chorôb v prípadoch reálneho sveta nie je také priamočiare, ako pozrieť sa na bázu faktov a porovnať hodnoty. Často je nutné zistiť omnoho viac faktorov a vykonať ďalšie testy pred určením prítomnosti choroby. V našom prípade sa skôr pozrieme na systém ako na bázu faktov a jednoduchých pravidiel s výstupnou hodnotou true alebo false.

3.3.1 Prvý príklad evaluácia animácie pre štýl Blackboard

Evaluácia prebieha na dvoch animáciách, kde sme pozorovali validitu nájdenia chorôb. Pri prvej sme riešili, či dokážeme prejsť celý strom možností a otestovať, či odpovede sedia vzhľadom na to, čo sme si stanovili ako náš výsledok. Na Obrázku 3.7 v *modrej* farbe možno vidieť diagnózu s priradením symptómov. Ďalej je vidieť postup, akým sme testovali animáciu: v prvom stĺpci (*zelení*) sme na vstupe zadali hodnotu presnosti, podľa ktorej sme chceli usúdiť nájdenie diagnózy. V *žltých* stĺpcoch je možno vidieť symptóm, ktoré užívateľ zadal na vstup do konzoly. Číslovanie v týchto stĺpcoch hovorí o poradí zadania do konzoly. Následne v *červenej* časti je možné vidieť, ako sme odpovedali na Otázky (Pravidlá) T1 a T2. Hodnota *null* predstavuje, že do systému sme nezadali symptómy alebo systém po nás nepožadoval odpoveď na pravidlá.

Záver evaluácie pre jednoduchú animáciu, ako je vidieť v Obrázku 3.7, v posledných stĺpcoch (*Prediction* a *Result*), ukazuje, že moje očakávania (*Prediction*) správania systému neboli úplne správne. Prečo je to tak, skúsim vysvetliť tým,

diagnosis							
cold(headache, fever) + T1 stomach_infections(fever, nausea) == SI + T2							
Test(ACC%)	headache	nausea	fever	T1	T2	Prediction	Result
Test1(100%)	1.	2.	null	1	1	null	null
Test2(100%)	1.	null	2.	1	null	cold	cold
Test3(100%)	1.	2.	3.	1	1	cold || SI || cold,SI	cold || SI || cold,SI
Test4(100%)	null	1.	2.	1	1	SI	SI
Test5(50%)	null	null	1.	1	null	cold || SI || cold,SI	cold

Obr. 3.7: Evaluácia animácie štýlu Blackboard

že správanie sa vlákien nie je úplne jednoznačné. Viacero KS sa môže spustiť naraz a niektoré sa nemôžu spustiť počas behu iných (blokujúce alebo neblokujúce). Tak sa stáva systém veľmi ťažko testovateľným, dokonca aj v prípade pri málo symptómoch, čím sme potvrdili tvrdenie v časti 2.3.2, kde hovoríme o správnosti riešenia a jeho nemožnosti presne určiť, či je výsledok je očakávaným alebo efektívnejším riešením z hľadiska vyhľadávania za pomoci viacerých paralelných KS.

3.3.2 Druhý príklad evaluácia animácie pre štýl Blackboardd

Pri druhej animácii sme sa pokúsili zistiť to isté, ale s rozdielom, že počet symptómov bol 20 a počet potenciálnych diagnóz bol 10. Teda strom možností bol oveľa väčší, preto sme testovali animáciu pre všetky symptómy a diagnózy, kde presnosť bola stanovená na 100%. Týmto sme chceli overiť správnosť navrhnutého systému v optimálnom stave, pretože dostaneme všetky potrebné údaje na tabuľu(**BB**). Exemplárny príklad: máme diagnózu *Migraine(headache, nausea, vomiting, dizzines)* a do vstupu zadané všetky symptómy: headache, nausea, vomiting, dizzines. Ako výstup očakávame Migrénu. Počas evaluácie týchto výstupov sme narazili na nečakaný problém tohto systému. A to v prípade, ak sú napríklad diagnózy definované následovne: diagnóza_A (fever) a diagnóza_B (fever, cough, congestion). Užívateľ zadal túto sadu symptómov: fever, cough, congestion. V procese prehľadávania stromu nájdeme prvý výsledok, ktorý spĺňal podmienku na 100% a uzavrel ďalšie prehľadávanie možností. Tým pádom sme dostali výsledok diagnóza_A, no naše očakávanie bolo diagnóza_B. Toto sme dokázali obísť, keď

sme poradie symptómov zadávali naopak, teda: congestion, cough, fever, keďže *congestion* je symptóm, ktorý sa nachádza v menej diagnózach ako horúčka, tým pádom sme vedeli donútiť systém prehľadať najprv vetvy, v ktorých sú menej časté choroby. Ďalšou možnosťou bolo, že na výstupe sme dostali obe možnosti, teda diagnóza_A a diagnóza_B, ktoré sa vyhodnotili v jednom cykle obe na 100%. Tento príklad možno vidieť na Obrázku 3.8.

```
create object instance Influenza of Diagnosis;
create list x2 of Symptom  {fever, sore_throat, muscle_pain, fatigue, cough, congestion};
add Influenza.create("Influenza", x2,diagnoseRuleNull) to self.listOfDiagnoses;

create object instance Common_Cold of Diagnosis;
create list x8 of Symptom {sore_throat, cough, congestion};
add Common_Cold.create("Common Cold", x8,diagnoseRuleNull) to self.listOfDiagnoses;
```

Obr. 3.8: Chyba zhodných symptómov v podmnožinách

Záver evaluácie pre druhej animáciu bol uskutočnený bez ďalších ťažkostí a výsledky boli konzistentné s predpokladmi, ktoré sme si stanovili. Riešením tohto problému bolo pridať ďalšie pravidlá, ktoré môžu vylúčiť diagnózy, ako v prvom príklade, kde diagnózy obsahovali ďalšie kontrolné otázky T1 a T2. Túto opravu môžeme vidieť na obrázku 3.9. Pre ďalšie zlepšenie by bolo potrebné pridať ďalšie pravidlá a spôsoby ich kontroly.

```
create object instance Influenza of Diagnosis;
create list x2 of Symptom  {fever, sore_throat, muscle_pain, fatigue, cough, congestion};
add Influenza.create("Influenza", x2,T1) to self.listOfDiagnoses;

create object instance Common_Cold of Diagnosis;
create list x8 of Symptom {sore_throat, cough, congestion};
add Common_Cold.create("Common Cold", x8,T2) to self.listOfDiagnoses;
```

Obr. 3.9: Oprava chýb zhodných v podmnožinách symptómov

3.3.3 Evaluácia vygenerovaného zdrojového kódu v jazyku Python vzhľadom na korektnosť pre štýl Blackboard

Spôsob hodnotenia bude dodržovať rovnaký postup ako v prípade štýlu Pipes and Filters 3.2.2, zahŕňajúci všetky kroky od opravy chýb po samostatné testovanie funkcionality. V tomto prípade sme vygenerovali kód pre komplexnejšiu animáciu 3.3.2.

Chyba parsera OAL kódu V tomto prípade nemôžeme hovoriť o probléme ako takom, ale skôr o unáhlenom uvoľnení netestovanej verzie. Chyba, ktorú je možné vidieť na obrázku 3.10, je skôr kategorizovaná ako prehliadnutie než ako chyba samotného parsovacieho nástroja. Vzhľadom na to, že bolo pridaných niekoľko nových metód a funkcií do Animarchu, nie všetky zmeny sa dostali aj do parsovacieho nástroja. V tomto prípade hovoríme o kombinácii funkcie *contains* a triednych metód *Equals*. Keď že funkcia *contains(List, Element)* kontroluje jednotlivý prvok $x \in List, x.Equals(Element)$, jednalo sa o prehliadnutie, pretože nie všetky triedy, na ktorých bola vykonaná operácia *contains*, obsahujú metódu *Equals*.

```
class Diagnosis:
  ...
  def Equals(self, obj):
    return self == obj
```

Obr. 3.10: chýbajúca metóda Equals v triede Diagnosis pre štýl Blackboard

3.3.4 Evaluácia metód Python kódu pre štýl Blackboard vzhľadom na funkcionalitu

V tejto časti sa budem odkazovať na prílohu 3.3.5 v ktorej sa nachádza pôvodná vygenerovaná verzia a príloha 3.3.5, v ktorej nájdete vylepšenú verziu Python

kódu, ktorú sme očakávali. V tomto prípade spomeniem, že som nezaznamenal ďalšie výrazné zmeny okrem tých, ktoré boli uvedené v sekcii pre štýl P&F v časti 3.2.3. S vygenerovaným kódom súhlasím a nevykonal by som ďalšie výrazné zmeny, pokiaľ ide o funkčný aspekt v porovnaní s tým, ako bol kód vytvorený.

3.3.5 Evaluácia výstupov animácie a python kódu

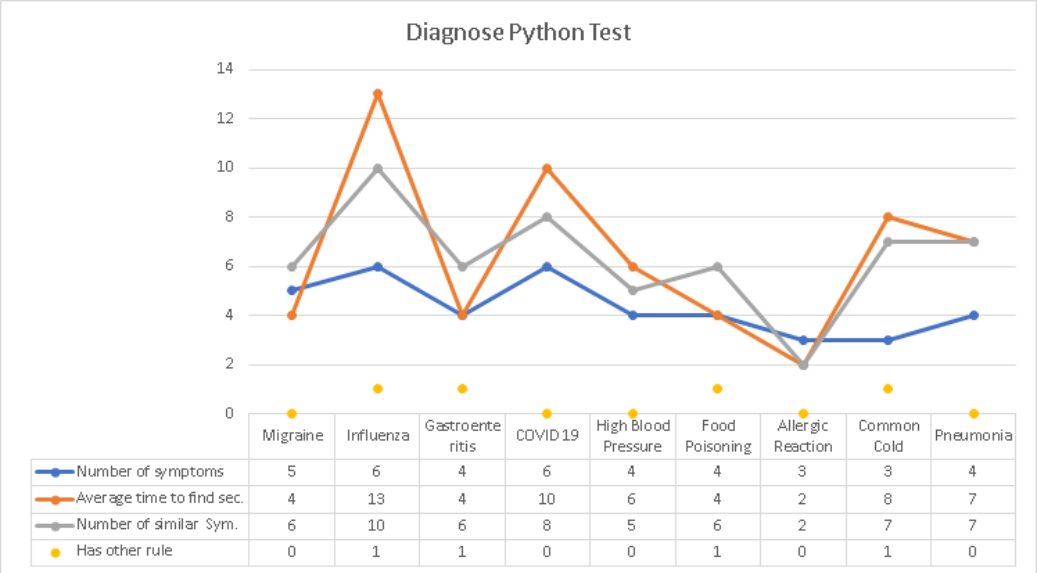
V tejto časti budem odkazovať na prílohy kódov 3.3.5 súbor„bb original.py“ a prílohu 3.3.5, v ktorej nájdete kód „bb original vylepsena.py“, ktorý sme očakávali a zároveň prílohu 3.3.5 kde je testovací súbor „bb original vylepsena test.py“. V tomto prípade spomeniem, že som nezaznamenal ďalšie výrazné zmeny okrem tých, ktoré boli uvedené v sekcii pre štýl P&F v časti 3.2.3.

Ďalej sme skúsili vykonať rovnaký test ako v prípade uvedenom v časti 3.3.2, pričom nastala neočakávaná zmena. Výsledky testovania sa líšili od výsledkov zo softvéru AnimArch. Ako bolo spomenuté v obrázku 3.8, kde sme na výstup dostali diagnózu, ktorej podmnožina symptómov bola obsiahnutá v inej diagnóze, alebo sme dostali obe diagnózy naraz. V tomto prípade došlo k neočakávanému zisteniu, keď sme kontrolovali presnosť na 100%, dostali sme vždy len jednu výslednú diagnózu. Našou hypotézou bolo, že použitie príkazu *sleep* vo vláknach v Pythone sa správa odlišne od príkazu *wait* v prostredí AnimArch. Túto hypotézu sme sa pokúsili overiť zmenami času *sleep* a následne úplným odstránením. Aj po zmene času *sleep* sa nám stále nepodarilo dosiahnuť stav systému ako v animácii v prostredí AnimArch. Po odstránení príkazu *sleep* sa nám podarilo dosiahnuť stav, kedy na výstup dostaneme obe diagnózy naraz. V žiadnom prípade sa nám nepodarilo dostať diagnózu s podmnožinou symptómov na výstup samostatne.

Vyhodnotenie výsledkov hľadania diagnóz po rovnakom teste sme vykonali ďalší test, ktorý je uvedený v prílohe 3.3.5. Tento test sme nechali bežať 10-krát za sebou, aby výsledky boli čo najmenej ovplyvnené funkčnosťou vlákien. V tomto teste sme pozorovali vplyv počtu symptómov na diagnózy, počtu zhodných symptómov s inými diagnózami a pridaných pravidiel na časovú efektivitu hľadania diagnóz. Vyhodnotenie výsledkov sme následne zobrazili na grafe 3.11. Naše

očakávania boli zhodné s tými, ktoré sme pozorovali na grafe. Z grafu sme pozorovali:

1. Čím viac symptómov, tým dlhší čas hľadania (Infuenza a COVID 19).
2. Čím viac podobných symptómov, tým dlhší čas hľadania (Allergic Reaction a Common Cold).
3. Pridanie pravidla nemá výrazný vplyv na čas hľadania.



	Migraine	Influenza	Gastroenteritis	COVID 19	High Blood Pressure	Food Poisoning	Allergic Reaction	Common Cold	Pneumonia
Number of symptoms	5	6	4	6	4	4	3	3	4
Average time to find sec.	4	13	4	10	6	4	2	8	7
Number of similar Sym.	6	10	6	8	5	6	2	7	7
Has other rule	0	1	1	0	0	1	0	1	0

Obr. 3.11: graf výsledkov pre druhu animáciu BB

Záver evaluácie Python kódu celkovo som nenašiel žiadne výrazné problémy v kóde počas evaluácie, okrem drobných chýb, ktoré by sa mali dať opraviť v prostredí AnimArch. Kód, ktorý bol vytvorený po finálnej implementácii štýlu BB, je jasnejší a pridaním funkcionalít nad zoznamami sa výrazne rozšírili možnosti funkcionality. Vo všeobecnosti mám hlavne výhrady ku konštruktorom tried, kedy nevyužívame vstavanú metódu __init__, ktorú nám ponúka Python. Naopak, odkazujeme na triedne parametre, čo považujem skôr za zlé praktiky v kóde (bad

smells). Treba ešte spomenúť, že celá animácia P&F by bola kódovo omnoho prehľadnejšia, ak by sme ju vytvárali teraz na záver súbežne s novými možnosťami, ktoré nám ponúka AnimArch. Kód je použiteľný a chyby detegované v kóde boli minimálne. Veľa chýb, ktoré vznikli pri generovaní kódu, hlavne v P&F, by už teraz nevznikli, ak by sme diagram vytvárali od začiatku, keďže ukladanie súborov bolo opravené. Na záver ešte poviem, že kód veľmi odzrkadľuje ten, ktorý bol písaný v OAL.

Záver

V tejto diplomovej práci sme sa primárne venovali vývoju rámcov s využitím metodológie MDD. Konkrétne vývojom riadeného modelu, na ktorom je založený softvér AnimArch. Naším cieľom bolo demonštrovať, že tento softvér je vhodný nielen pre vývojové účely, ale aj pre výučbu. Zistili sme, že softvér je efektívny pre vývoj, s veľkým potenciálom, avšak existuje ešte priestor na ďalšie vylepšenia.

Funkcionality softvéru sme rozšírili o operácie nad reťazcami a zoznamami, čo umožňuje ich lepšie využitie v rámci ďalších implementácií štýlov a vzorov. Okrem toho sme softvér obohatili o mechanizmus maskovania názvami, čo prispieva k lepšiemu pochopeniu jeho fungovania na objektovej vrstve. Súčasne sme detailne vysvetlili princípy fungovania jednotlivých štýlov Pipes and Filters a Blackboard a ich aplikácií vo vhodných scenároch. Na záver sme tieto scenáre spolu s triednym diagramom preniesli do programovacieho jazyka Python, čím sme vyhodnotili schopnosti a nedostatky tohto nástroja.

Samotný softvér AnimArch nie je dokonalým nástrojom a vyžaduje ďalší vývoj, ale jeho budúcnosť vidíme vo veľmi pozitívnom svetle. Po zlepšení jeho funkcionality máme pozitívny pohľad na jeho perspektívnu budúcnosť. Pri práci s Animarch sme identifikovali možnosti na jeho ďalšie vylepšenia. Jednou z nich je implementácia vlastného konštruktora pre vytváranie objektov, podobne ako pri konštruktore listu objekt1, objekt2, ..., objektN. Ďalšou vhodnou zmenou by bola možnosť farebného zvýraznenia triedneho diagramu podľa výberu používateľa. Tieto farby by sa následne premietali do objektovej vrstvy, čo by výrazne zlepšilo prehľadnosť, či už v kontexte kľúčových tried patriacich k štýlu, alebo objektov

vytvorených pomocou OAL kódu.

Ďalšie možné zmeny by sa mohli týkať vylepšenia procesu písania OAL kódu, buď priamo v konzolovej aplikácii, alebo prostredníctvom používateľskej príručky, kde by boli zhrnuté jednotlivé ukážky použitia a nebolo by potrebné detailne študovať zdrojové kódy a ich funkčnosť. Takéto zmeny by zvýšili efektivitu a použiteľnosť softvéru AnimArch. Ako posledné dodáme ešte rozšírenia OAL jazyka o funkcionality, medzi ktorými by sme uvítali funkciu random a randomInt. Funkcia random by fungovala aj nad zoznamami, napríklad x = random(List). Funkcia randomInt(Od, Do) by vrátila náhodné celé číslo v rozsahu <Od, Do>.

Literatúra

[1] Akka kniznica pre java and scala. Accessed on November 27, 2023.

[2] implementacia v jaziky pyhon. Accessed on November 27, 2023.

[3] microsoft azure implementacia pipes and filter. Accessed on November 27, 2023.

[4] Object action language. OAL. Accessed on November 18, 2023.

[5] Omg unified modeling language (omg uml), version 2.5.1. OMG, December 2017. Accessed on November 16, 2023.

[6] Hakan Burden, Rogardt Heldal, and Toni Siljamaki. Executable and translatable uml – how difficult can it be? In *2011 18th Asia-Pacific Software Engineering Conference*, pages 114–121, 2011.

[7] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, and Michael Kircher. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 2007.

[8] Greg Butler. *Blackboard Architecture*. Accessed on November 30, 2023.

[9] Alexander S. Gillis. Model-driven development. TechTarget, June 2018. Accessed on November 13, 2023.

[10] Swapna S Gokhale and Sherif M Yacoub. Reliability analysis of pipe and filter architecture style. In *SEKE*, pages 625–630. Citeseer, 2006.

[11] J.J. Gutiérrez, M.J. Escalona, and M. Mejías. A model-driven approach for functional test case generation. *Journal of Systems and Software*, 109:214–228, 2015.

[12] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[13] Vasudevan Jagannathan. *Blackboard architectures and applications*. Elsevier, 1989.

[14] Durgaprasad Janjanam, Bharathi Ganesh, and L Manjunatha. Design of an expert system architecture: An overview. *Journal of Physics: Conference Series*, 1767(1):012036, feb 2021.

[15] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[16] HongXing Liu, YanSheng Lu, and Qing Yang. Xml conceptual modeling with xuml. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 973–976, New York, NY, USA, 2006. Association for Computing Machinery.

[17] Júlia Pukancová Martin Homola. Lecture 4: Logic programming. Matriáli s 11 Oct 2022.

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

[19] Jorge L Ortega-Arjona. The parallel pipes and filters pattern. *A Functional Pa rallelism Architectural Pattern for Parallel Programming*, 2005.

[20] Terence Parr. The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, pages 1–326, 2013.

[21] Abhilash Ponnachan. Architecture styles versus architecture patterns, May 2018. Accessed on November 22, 2023.

[22] Terry Quatrani and UML Evangelist. Introduction to the unified modeling language. *A technical discussion of UML*, 6(11):03, 2003.

[23] Mark Richards. *Microservices vs. service-oriented architecture*. O'Reilly Media Sebastopol, 2015.

[24] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[25] Anubha Sharma, Manoj Kumar, and Sonali Agarwal. A complete survey on software architectural styles and patterns. *Procedia Computer Science*, 70:16–28, 2015.

[26] Nenad Ukić, Josip Maras, and Ljiljana Šerić. The influence of cyclomatic complexity distribution on the understandability of xtuml models. *Software Quality Journal*, 26:273–319, 2018.

Príloha A: obsah elektronickej prílohy

priečinok	URL adresa
Obrázky v plnej kvalite	https://drive.google.com/obrazky_v_pl...
P&F animácia	https://drive.google.com/pipes_and_fi...
Blackboard animácia	https://drive.google.com/Blackboard...
Evaluácia obrázky	https://drive.google.com/Evaulacia_ob...
Python kód vygenerovaní	https://drive.google.com/Python_kod_v...
Python kód upravení	https://drive.google.com/Python_kod_u...
Python kód BB test	https://drive.google.com/Python_kod_B...