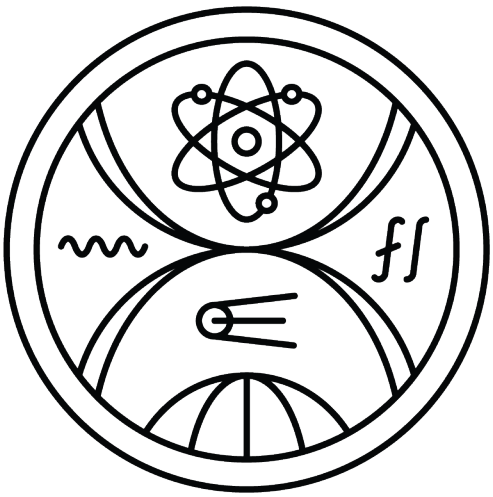
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



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 mojemu školiteľovy doc. Ing. Ivan Polášek, PhD., za vedenie práce, cenné rady, konzultácie a množstvo trpezlivosti a motivácie pri tvorbe Diplomovej práce. Takztie z radou spomeniem este zopar mien ktore mi pomohly pri tvorbe tejto prace L.Radovansky,...

Abstrakt

V rámci diplomovej práce sme vytvorili exemplarne moduly(Rámce). Hlavnou naplnou prace bolo vitvorenie spoosbu na reprezentaciju 2 zvolenich stylov ako ”Pipes and filtersänd ”BlackBoard”. Pri tvorbe tychto stylov sme sa zamerali na prakticku reprezentaciju a pochopenie vseobecnej verjnosti nie len pre vybranich ludi ktori maju pokrocile vzdelanie v pochopeni architekturovvich stylov.

V style Pipes and Filters sme sa pozreli na jednoduchsi priklad ”Fibonaci”v ktorom sme sa snazili zachitit vsetky podsatne casty stylu ako Filtre,Pipe. V pripade Filtrov sme sa zamerali na linerne spracovanie. Zarovnen sme chceli ukazat vsetky mozne vyuziatia filtorv ako moznoti Tranfare, Marge and Split.

Nasledne praca pokracuje spracovanim stylu BlackBorad v ktorom sme sa pokusili naimplementovat o nieco zlozitejsi priklad iterpreter pre programovaci jazik Prolog.

V praci sa dozvieme Blizsie informacie o vyvojovom procesi v softvery AnimArch ako aj o uskaliach spojenich s vyvojom ako takim, problemami a rieseniami ktorimi sme zlepsili software AnimArch.

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

In the context of the diploma thesis, we created exemplary modules (Frameworks). The main focus of the work was to create a way to represent two chosen architectural styles, namely ”Pipes and Filtersänd ”BlackBoard.”When developing these styles, we aimed at practical representation and understanding for a general audience, not just for those with advanced education in architectural styles.

In the ”Pipes and Filters”style, we examined a simple example, ”Fibonacci,ïn which we attempted to capture all essential aspects of the style, such as Filters and Pipes. In the case of Filters, we focused on linear processing. Additionally, we wanted to demonstrate all possible uses of filters, such as Transfer, Merge, and Split.

Subsequently, the work continues with the exploration of the ”BlackBoard”style, in which we attempted to implement a slightly more complex example, an interpreter for the Prolog programming language.

The thesis provides detailed information about the development process in the AnimArch software, as well as the challenges associated with development itself, problems, and the solutions that have improved the AnimArch software.

Keywords: first, second, third (or fourth, fifth)

Obsah

Zoznam obrázkov

Zoznam tabuliek

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 **Pipes and Filters** to bude overenie hypotéz a pre **Black board** to bude kvalitativne ohodnotenie relevancie pomocou standardu pre vyvoj (RJT).

1 Základná terminológia a použité nástroje

1.1 Základná terminológia / terminológia

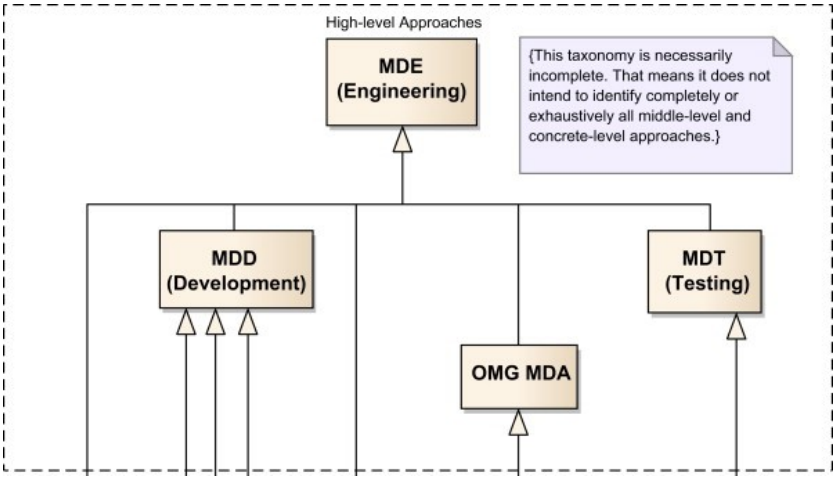
V tejto časti kapitole si podrobnejšie vysvetlíme a oboznámime sa so základnými pojmy, ktoré sú kľúčové pre našu prácu.

1.1.1 Modelovo orientované prístupy/ Prístupy Modelovej Orientácie

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



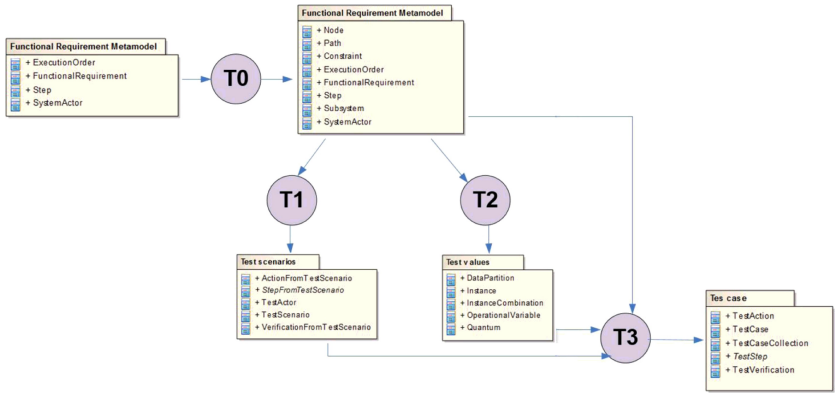
Obr. 1.1: najvyššia abstrakcia MD [24]

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.

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 prvkov. Ponúkajú nám jasný a štruktúrovaný pohľad na plánovanú funkcionalitu systému. Na rozdiel od tradičných testovacích metodológií, ktoré pozostávajú hlavne z



Obr. 1.2: meta model pre MDT[11]

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.

MDD - Model Driven Development / modelovo riadené vývojárstvo

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].

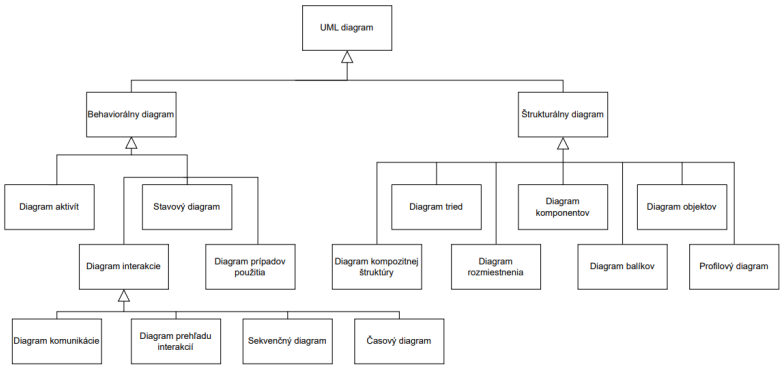
Úplný potenciál MDD spočíva v závere, teda automatickom generovaní kompletného a overiteľného kódu z modelu, pričom štandardy UML určujú najlepšie postupy, pre kompatibilitu a znovu použiteľnosť častí systémov.

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].



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

1.1.2 UML- Unified Modeling Language / Zjednotený 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.

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 / Spustiteľný zjednotený 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 / Spustiteľný a prekladateľný 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.

2 Analýza domény

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.

2.1 Návrhové štýly a vzory

V tejto časti si priblížime zásadné odlišnosti medzi štýlmi a vzormi.

2.1.1 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.

2.1.2 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.

2.1.3 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

2.2 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.

2.2.1 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].

2.2.2 Generovanie kódu

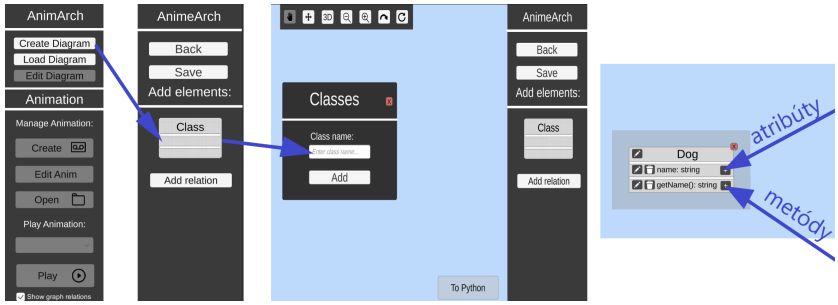
V súčasnej verzii AnimArch je možnosť generovať kód, ktorý z OAL kódu vytvorí ekvivalentný kód v jazyku Python. Toto tlačidlo je dostupné priamo po spustení prostredia, ako môžete vidieť ??. Pred kliknutím na tlačidlo je nevyhnutné načítať diagram a príslušnú animáciu (OAL kód). Po kliknutí má užívateľ 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**.

2.3 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.

2.3.1 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 Diagramä 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 2.1. 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 Diagramä následne Ëdit Diagram”, po skončený editovania je potrebne znovu uložiť

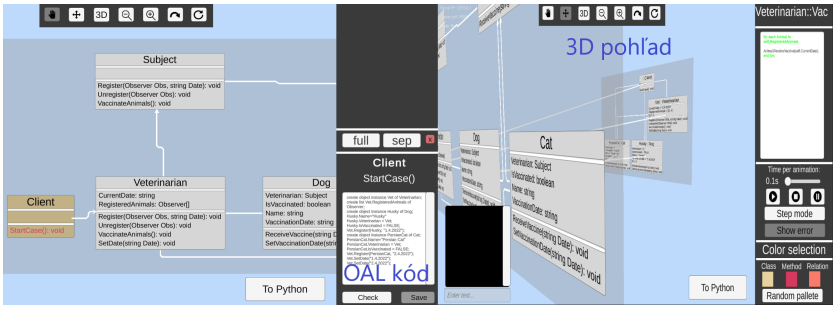


Obr. 2.1: práca v 2D priestore aplikácie AnimArch

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.

2.3.2 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 ”Createä 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 2.2 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. 2.2: pohľad na AnimArch v 3D

fakulty MATFYZ.

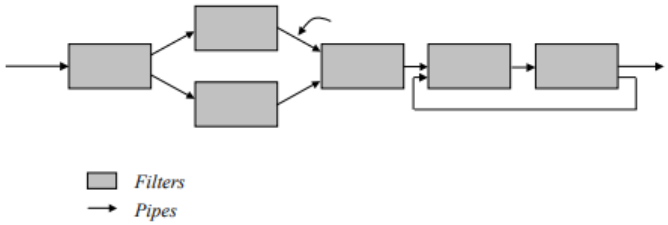
3 Návrh softwarových rámcov a ich implementácia

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.

3.1 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 3.1, 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. 3.1: 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.

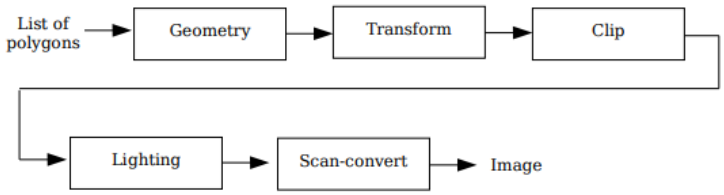
Opakované použitie - reusability Každý filter je navrhnutý tak, aby mohol byť opätovne použiteľný, pretože majú špecifickú funkcionalitu. To podporuje ich znovu použiteľnosť a zjednodušuje vývoj nových systémov s využitím už existujúcich filtrov.

3.1.1 Identifikácia problémov / 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.



Obr. 3.2: exemplár spracovania obrazu s využitím topológie P&F [19]

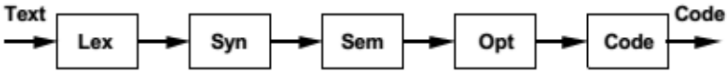
Problémy spojené s architektonickým štýlom Pipes and Filters Ďalej bližšie opíšeme problémy, s ktorými sme sa potykali pri implementácií architektonického štýlu Pipes and Filters. Snažili sme sa navrhnúť ľahko škálovateľný a zároveň udržateľný rámec, ktorý by reprezentoval tento architektonický štýl. 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

3.1.2 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 äko dátovodov a expressions sú zreťazené do podoby ëxp — 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 3.2 od autorov môžeme vidieť ich prístup na sériové spracovanie obrazu.



Obr. 3.3: exemplár dávkového spracovania [7]

Na obrázku3.2 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.

3.1.3 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.

Použitie dávkového spracovávania na vstupy používateľa Spôsob ako dávkové spracovanie funguje sme už opísali v sekcii3.1.2 Toto spracovanie využíva lineárne aplikovanie filtrov jedného za druhým obrázok3.3. 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.

```
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. 3.4: kód rekurzívna implementácia fibonacciho postupnosti

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-1ä ”n-2”. Na obrázku 3.4 môžeme vidieť implementáciu v kóde python. Teraz túto ukážku môžeme prerobiť do jednotlivých filtrov, kde ”n¡0”, predstavuje jeden nezávislý celok. Potom ”n == 0”predstavuje ďalší a ”n == 1 or n == 2”posledný a následne rekurzívne volanie s návratom.

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 3.5. 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. 3.5: kód lineárnej implementácia fibonacciho postupnosti

3.2 Implementácia rámcu Pipes and Filters

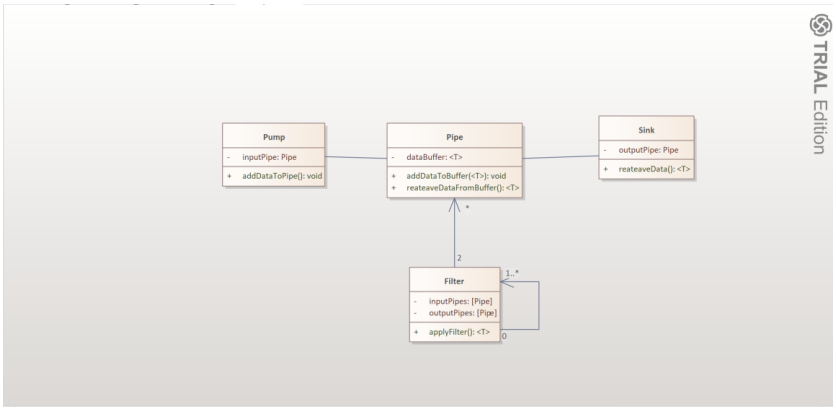
V nasledujúcich podkapitolách sa budeme venovať jednotlivým iteráciám, ktorými sme prešli počas vývoja.

3.2.1 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.3.6: 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.

Class sink má atribút outPipe typu Pipe(dátovod) a pomocou metódy retrieve data 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 dáta vieme vyberať. Class filter pozostáva z dvoch atribútov prislúchajúcich vstupným a výstupným dátovodom.

Na obrázku 3.7 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.



Obr. 3.6: všeobecná implementácia štýlu P&F v Enterprise Architect

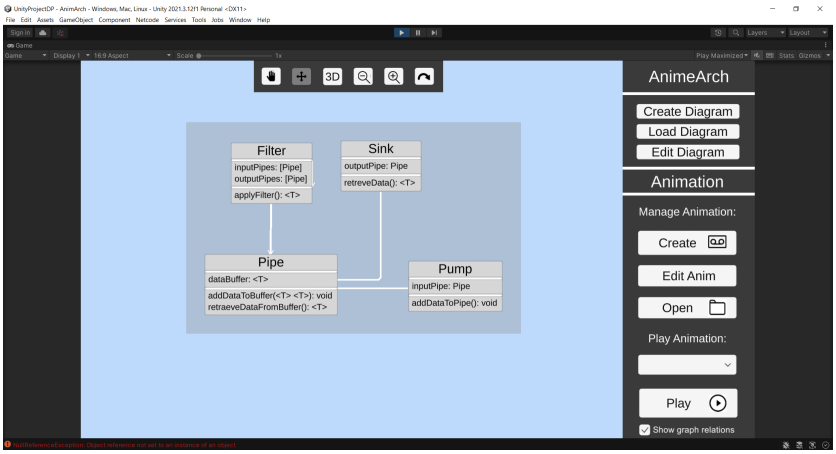
Rovnako ako v prvej časti sme sa pokúsili pracovať so softvérom EA a snažili sme sa implementovať rekurzívnu fibonacciho postupnosť, kde filterfibonacci vracal n-1 a n-2 ako vstupné dáta do dátovodu. V tomto príklade môžeme vidieť z obr.3.8 čítanie vstupu ako string, následnú konverziu do integeru pomocou filtra filterConvertInt a výstupné dáta sa dostanú do konečného dátovodu outputPipe a následne sa zobrazia v Sink.

Tak ako v predošlom prípade sme aj tento model uložili do xML súboru a následne nahrali do aplikácie AnimArch obr.3.9. Pri tejto implementácii sme evidovali stratu vzťahov medzi triedami ”filterCounter, pipeIntData”.

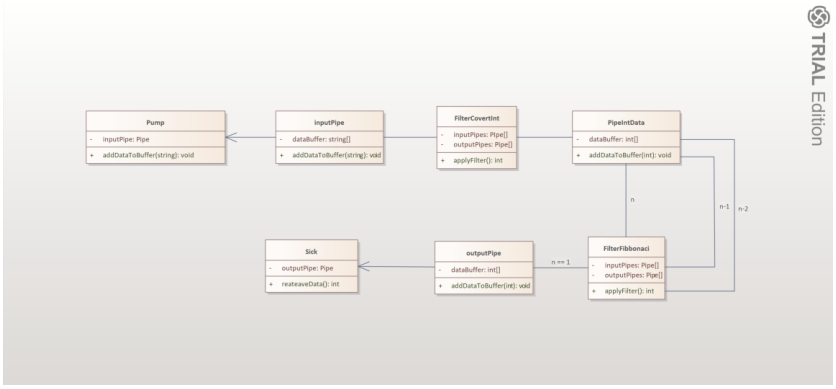
3.2.2 Iterácia 2 - dávkové spracovanie P&F

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.

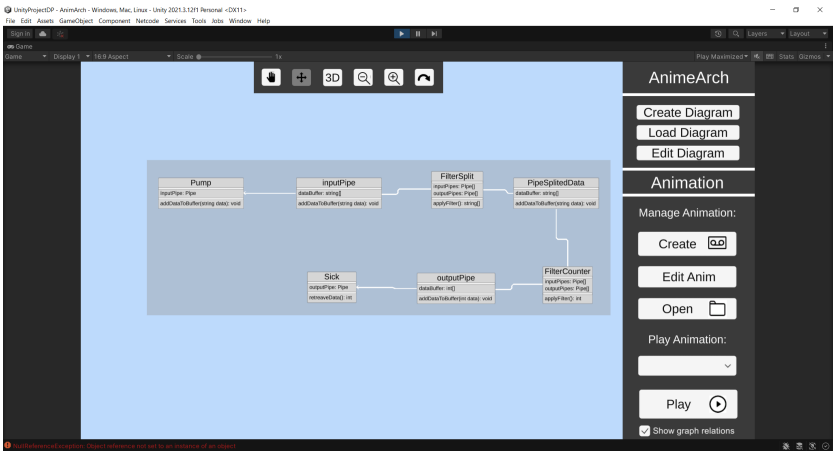
Na obrázku3.10 môžeme vidieť implementáciu dátového spracovania textu, pričom sme sa pokúsili filtrovať veľké písmená. Na tomto diagrame môžeme vidieť filter ”Filter_Capital_Letter”, ktorého metóda filter() prijíma vstupné dáta typu string a pozná výstupné miesto, kam majú byť dáta posielané. Samotná



Obr. 3.7: konverzia všeobecnej implementácie 3.6 do AnimArch



Obr. 3.8: implementácia fibonacciho v Enterprise Architekt



Obr. 3.9: konverzia implementácie 3.8 do AnimArch

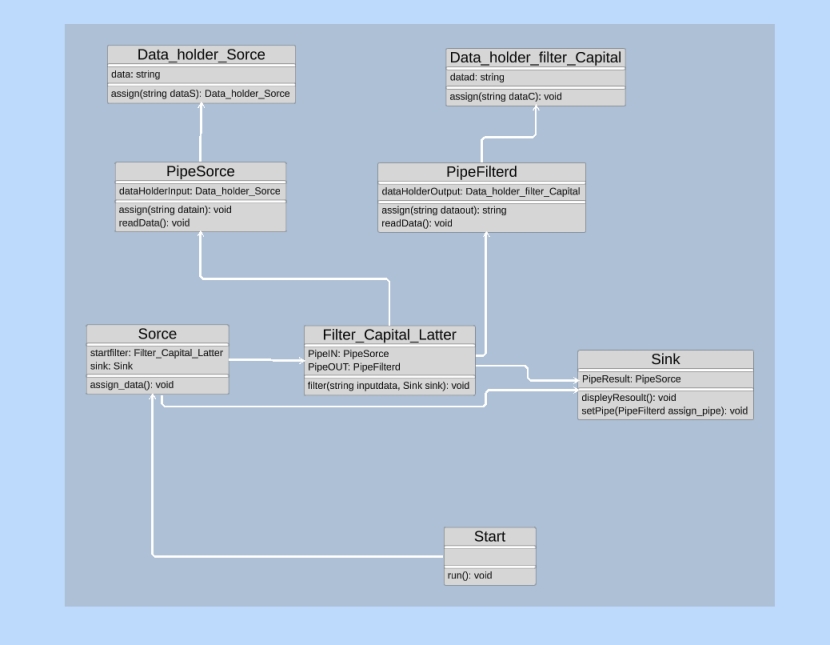
metóda prejde každé písmeno zo stringu a zapíše ho do výstupného dátovodu ”PipeFiltered”. Tento diagram bol vytvorený v prostredí AnimArch.

Na obrázku3.11 môžeme vidieť implementáciu OAL kódu pre metódu filter z class filter_capital_letter pomocou ”create object instance”vytvoríme objekty ”PipeSourceä ”PipeFilteredä ďalšie. Pomocou volania metódy ässign”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 spracovania veľkých písmen to vyzerá takto obr. 3.12 Z obrázka môžeme vidieť jasné nedostatky v OAL kóde, keďže niektoré objekty nemajú priradené iné objekty.

3.2.3 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.3.13. 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 (ĬsCorrectResultFilter”). Tento filter by vyhodnotil, či je

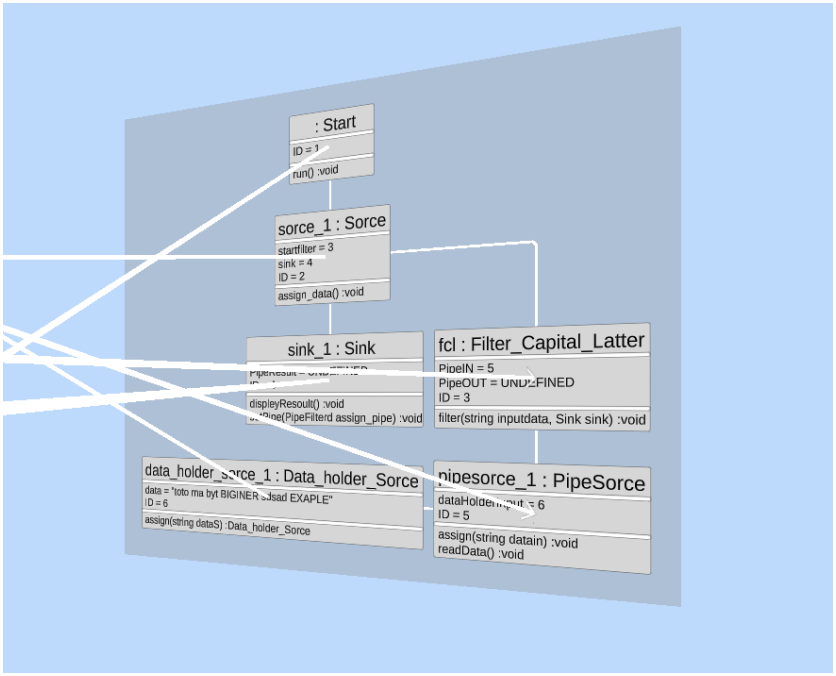


Obr. 3.10: triedny diagram pre filtrovanie veľkých písmen

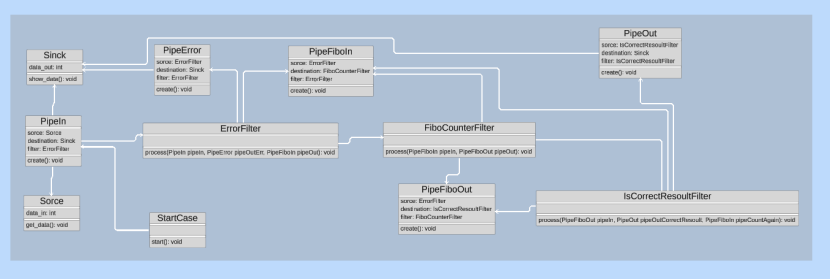
```
create object instance pipesorce_1 of PipeSorce;
self.PipeIN = pipesorce_1;
create object instance pipefilterd_1 of PipeFilterd;
self.PipeOUT = pipefilterd_1;

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. 3.11: kód OAL k filtrovanie veľkých písmen



Obr. 3.12: objektový diagram pre filtrovanie veľkých písmen



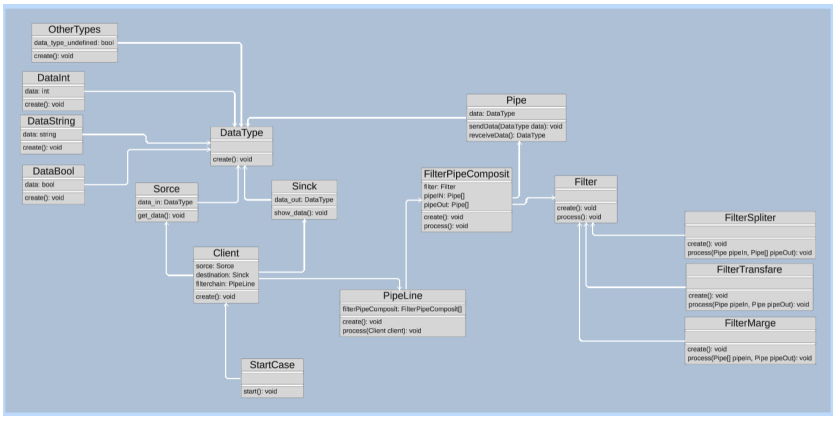
Obr. 3.13: diagram tried pre fibonacciho postupnosť

fibonacciho iterácia už v konečnom stave, alebo ešte pokračuje vo výpočte. V prípade ak je iterácia konečná, dáta sa posunú do triedy ”Pipeoutä 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.

Na obrázku3.14 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 ”Sourceä ”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 obrázku3.15 celkový triedny diagram sme zúžili na nutné minimum, aby sme odstránili zbytočnú komplikovanosť celkového systému.



Obr. 3.14: všeobecná implementácia štýlu P&F

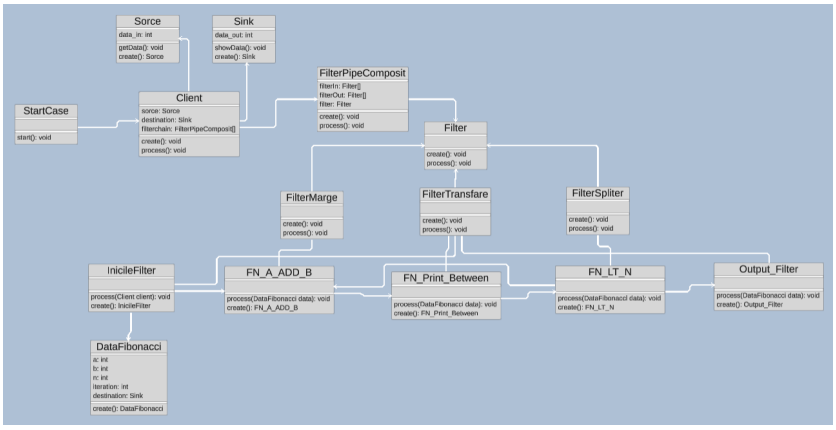
Zároveň na obrázku je možno vidieť, že naša implementácia fibonacciho, začína iniciálnym filtrom, ktorého cieľom je nastavenie hodnôt, následne sčítacím filtrom, ďalej filtrom vypisujúcim medzi výpočty fibonacciho postupnosti počas behu programu a nakoniec porovnávacím filtrom, ktorý buď znova opakuje výpočet, alebo posúva dáta do výstupného filtra.

3.2.4 Iterácia 4 - Finálna implementácia P&F

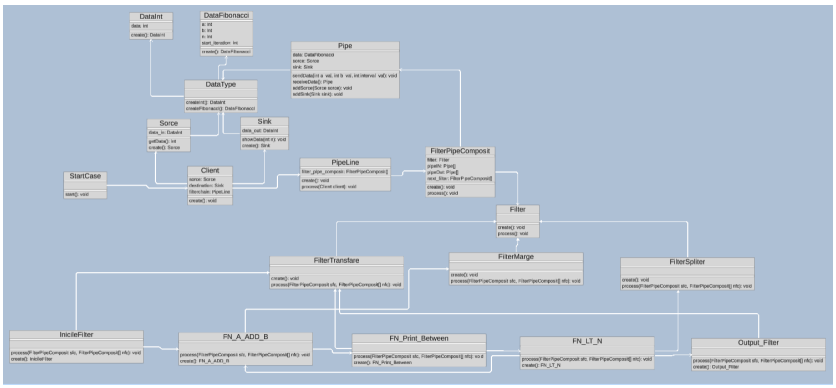
Táto iterácia je poslednou, ktorú tu opíšeme v časti určenej pre implementáciu rámca P&F. Na obrázku 3.16 prezentujem class diagram celého rámca, ktorý zahŕňa aj implementáciu fibonacciho postupnosti pomocou využitia komponentov štýlu.

Pri pohľade na obrázok 3.17 je možné pozorovať prepojenie inštancií objektu filtra. Zároveň môžeme všimnúť aj kľúčovú vlastnosť - znovupoužitia už existujúceho filtra pre ďalšie iterovanie výpočtov. Táto vlastnosť je definovaná pomocou abstraktných tried filtrov.

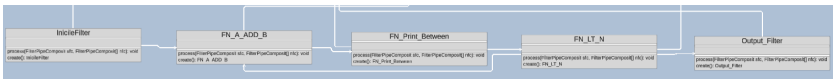
Pre našu implementáciu sú to triedy ”FilterTransfare,FilterMarge,ä ”FilterSpliter”. Každá z týchto tried implementuje určitú vlastnosť filtrov, ktorú sme opísali v časti 3.1. Vlastnost filtrov uvedených na obr.3.18, okrem spracovania dát, ktorú



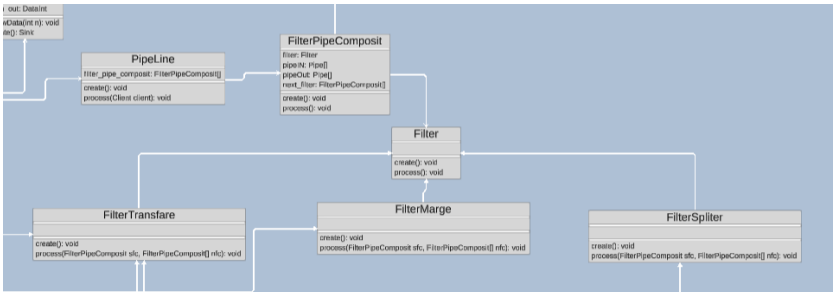
Obr. 3.15: implementácia štýlu P&F bez dátovodov



Obr. 3.16: úplné zachytenie rámca P&F



Obr. 3.17: fibonacciho postupnosť priblížený pohlaď na triedny diagram



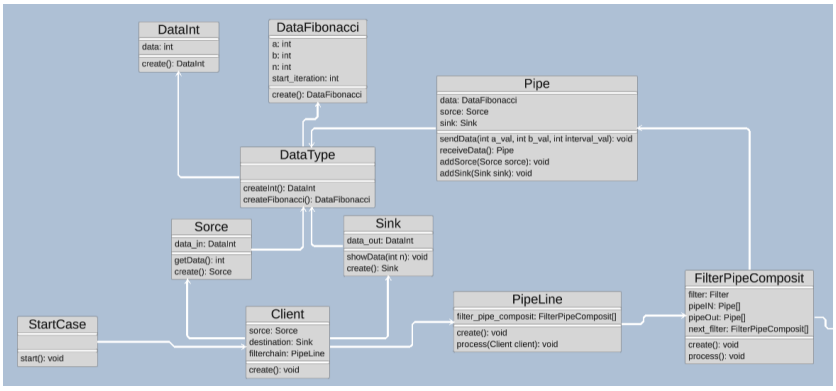
Obr. 3.18: priblížený pohlaď na triedy spojené s componenotm filter

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

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 ĬnicialFilter”, ktorej OAL kód je na obrázku 3.20. Prechádzame všetky vstupné dáta pomocou for cyklu. Vyberieme dátovod, ktorý používa táto trieda, a nastavíme hodnoty na úvodné hodnoty pre výpočet fibonacciho postupnosti, teda ä = 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.



Obr. 3.19: priblížený pohlaď na implementáciu štýlu P&F

```
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. 3.20: kód OAL triedy - InicileFilter

```
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. 3.21: kód OAL triedy - FN_A_ADD_B

V obrázku 3.21 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,ä znovu zavoláme ďalší filter.

”FN_Print”je trieda, ktorou by sme nepotrebovali pre správny výpočet, ale rozhodli sme sa vypisovať na konzolu priebežné výstupy počítania fibonacciho postupnosti. Zápis na konzolu je pomocou kľúčového slova ”write”.

”FN_LT_N”3.23 je náročnejšia trieda na implementáciu po stránke OAL kódu, keďže dáta zo vstupného dátovodu musia prejsť dvoma rôznymi filtrami: buď ”FN_A_ADD_B,äk je podmienka **LT** (less then) splnená a potrebujeme pokračovať v iterácii, alebo ÖutputFilter,”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 ïndex,”ktorá sa počas iterovania cyklu zvyšuje.

Na záver opíšeme OAL kód s obrázka 3.24, 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.

vygenerovani Python kod tuna budeme pisat Pyhon kode spusteni a celkovej funkcnosti alebo problemom spojenim s vygenerovanim kodom, VIANOCE.

```
for each INNN in sfc.pipeIN

data_pipe_print = INNN.receiveData();

if (data_pipe_print.data.start_iteration < data_pipe_print.data.n )

write("fibonaci 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. 3.22: kód OAL triedy - FN_Print

```
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. 3.23: kód OAL triedy - FN_LT_N

```
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;
```

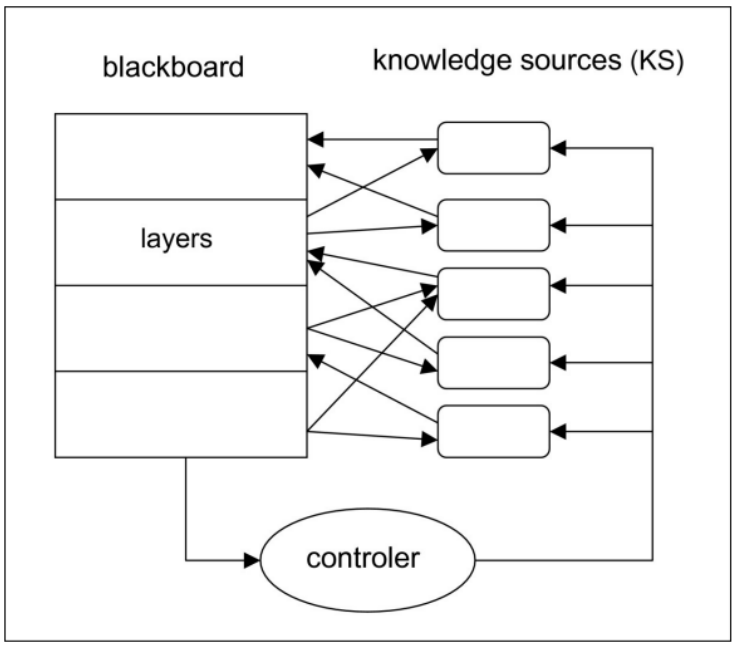
Obr. 3.24: kód OAL triedy - Output$_{F}ilter$

3.3 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 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.3.25. 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 / Nástenka (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.



Obr. 3.25: všeobecný diagram štýlu Blackboard.[8]

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).

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

ovládač / riadiaca jednotka / ovládacia 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.

3.3.1 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 ïnformation-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, prispie-

vajúc 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.

3.3.2 Identifikácia problémov / 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.

3.3.3 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.

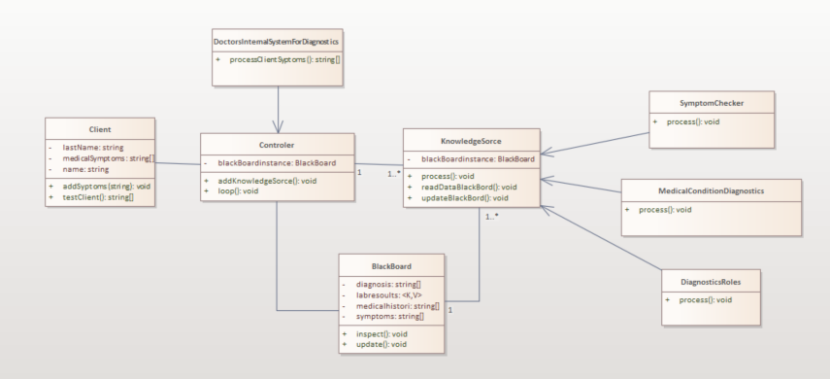
Pre implementáciu pravidlového systému na báze Prologu potrebujeme pochopiť základné problémy spojené s implementáciou pravidlového systému ako takého. Pravidlá, klauzuly, query - tieto prvky treba implementovať do class diagramu a následne dokázať ich vyhodnocovanie. Preto si definujeme základné pojmy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

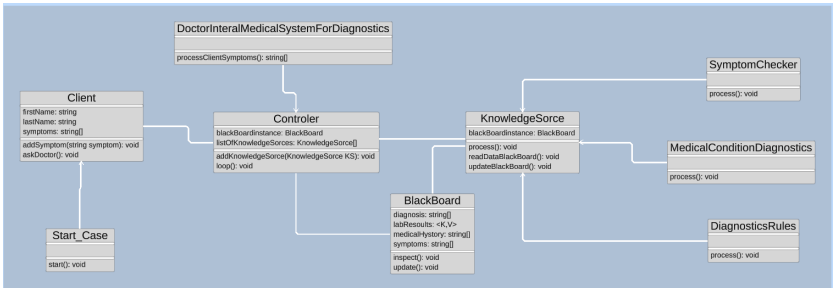
3.4 implementácia rámcu Blackborad

3.4.1 Iterácia 1 - Prví pohlaď na štýl Blackboard

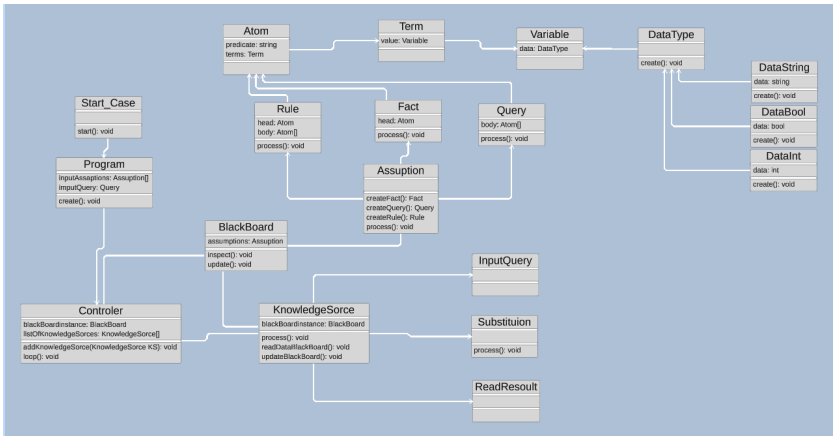
Pri tejto iterácii sme začali s vývojom v prostredí aplikácie **EA** pričom základni model na detekciu chorob bol pomerne dobrý výber implementácie tohto štýlu. Pri začiatku sme využili znalosti s kapitoly 3.3. Tak ako bolo uvedené v tomto článku [8], sme implementovali jednotlivé triedy pre komponenty tohoto modelu. Ako je možné vidieť s obrázka **??** znovu sme použili generické (¡K,V¿) typi pre vyhodnotenie laboratórnych testov. Po vygenerovaní .xml súboru, sme skúsili nahrať tento súbor do AnimArch a jednoducho zanimovať tento model. Opäť sme sa stretli s problémom nemožnosti interpretovať generické typi v jazyku OAL. Medzi ďalšie problémy patrilo to že pôvodný diagram nemal stravovaciu triedu ktorú sme pridali a v neposlednom rade sme mali zle určenú asociačné vzťahy medzi objektmi "Client, Controler a DoctorInternalMedicalSystemDiagnostics"ktoré je možno vedieť na obrázku 3.27.



Obr. 3.26: medicínsky systém v rámci Enterprise Architekt



Obr. 3.27: konverzia medicínskeho systému do Animarch



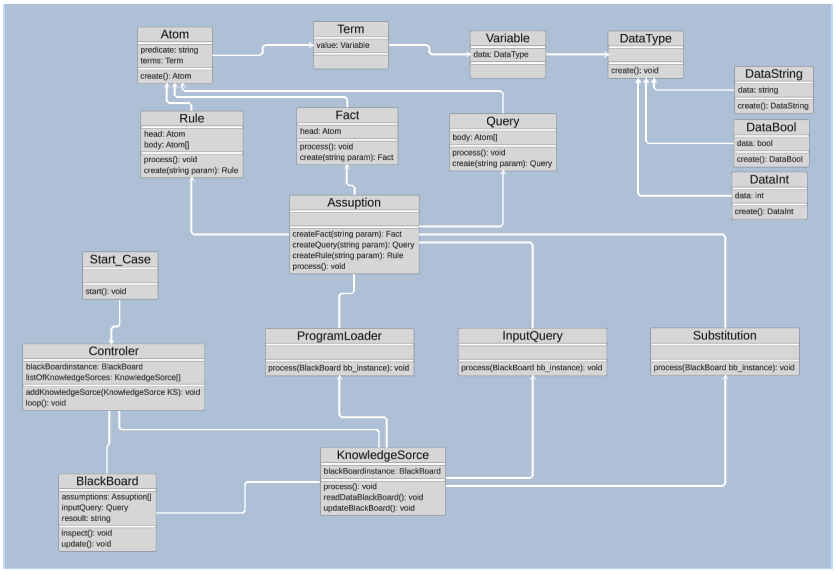
Obr. 3.28: pravidlový systém 1.pokus

3.4.2 Iterácia 2 - Pokus o implementáciu plavidlového 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 3.28, na ktorom sme zistili 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 ”KnowledgeSourceä rozumnými zdrojmi.

Tento druhý náčrt pravidlového modelu je vlastne ďalšou iteráciou 3.28. Pri jeho tvorbe sme sa snažili opraviť nedostatky predchádzajúceho modelu. Zistili sme, že pri komunikácii potrebujú všetky myšlienkové zdroje prístup k pravidlám jazyka prolog, ako môžeme vidieť na obrázku 3.29. Ich funkcionalita však bude mierne odlišná.

Trieda ”ProgramLoader”bude načítavať program, ktorý používateľ chce riešiť. ĬnputQuery”bude očakávať vstup typu ”Query”, tento po overení validity tejto



Obr. 3.29: pravidlový systém 2.pokus

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

V tejto podcasti 2.iterácie sa zameriame na samostatné spúšťanie rozumných zdrojov, pričom rozlišujeme dva prístupy. Prvým je lineárne spracovanie procesov postupne jeden za druhým. Na obrázku 3.30 môžeme vidieť OAL kód s for cyklom, ktorý prechádza cez všetky rozumové zdroje.

V druhom prípade, teda v paralelnom, môžeme pozorovať ten istý cyklus s použitím vlákien. Z kódu na obrázku 3.31 môžeme identifikovať dve kľúčové slová: ”thread”pre spustenie vlákna a ënd thread”pre ukončenie procesu vlákna.

Na záver prezentujem spôsob implementácie metódy ”process”rozumných zdrojov v oboch scenároch (OAL). Ako môžeme vidieť na obrázku 3.32, spôsob vykonávania sa výrazne nezmenil. Hlavným rozdielom je to, že pri paralelnom spracovaný je telo metódy obalené v nekonečnom cykle. Týmto zabezpečíme naozajstnú náhodnosť spúšťania vykonávania metódy ”process”.

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

Obr. 3.30: lineárne spustenie rozumových zdrojov

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

Obr. 3.31: paralelne spustenie rozumových zdrojov

```
// 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. 3.32: Implementácia metódy ”process”v lineárnom vs. paralelnom spúšťaní.

3.4.3 Iterácia 3 - Systém na detekciu hríbu / Systém na detekciu chorob

Jeden z týchto modelov implementujeme počas VIANOC.

4 Vyhodnotenie a výsledky

tuna je to co prave robim

4.1 Vyhodnotenie hypotézy Pipes and Filters

4.1.1 H1

Tu je nejaký text.

4.1.2 H2

Tu je nejaký text.

4.1.3 H3

Tu je nejaký text.

4.1.4 H4

Tu je nejaký text.

4.2 Relevance judgement technique (RJT) pre architektonický štýl BB

Tu je nejaký text.

Záver

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

Príloha B: Používateľská príručka