Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Implementácia návrhových vzorov v xUML a OAL

Bakalárska práca

2023
Michelle Gavlák

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Implementácia návrhových vzorov v xUML a OAL

Bakalárska práca

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

Bratislava, 2023

Michelle Gavlák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Implementácia návrhových vzorov v xUML a OAL
Implementing design patterns in xUML and OAL

Anotácia: Zložitosť zdrojového kódu rozsiahlych softvérových systémov nás núti k výskumu a pokusom zaviesť do oblasti softvérového inžinierstva nové metódy vizualizácie a modelovania, ktoré by podporili ľahšie porozumenie, rozširovanie a znovupoužitie funkcionality a softvérových znalostí v zdrojovom kóde.

Analyzujte vybraté metódy modelovania v softvérovom inžinierstve (napríklad Executable UML a Object Action Language), interaktívnu grafiky v Unity a náš prototyp animácie modelu v xUML z roku 2021/2022.

Platforma Unity umožňuje prácu v 2D a 3D priestore ale aj migrovať do virtuálnej (VR) alebo rozšírenej reality (AR).

Cieľ: Cieľom práce je nový návrh a implementácia dvoch vzorov v xUML diagrame tried a v jazyku OAL a kontrola jeho funkcionality vo vygenerovanom zdrojovom kóde jazyka Python.

Výstupom BP bude obohatenie bázy návrhových vzorov, ktorý by sa dal používať pri modelovaní a vizualizácii softvérovej štruktúry a funkcionality alebo na testovanie funkčnosti a animácie architektonických štýlov a vzorov. Pomohol by aj pri výučbe softvérového inžinierstva na vysvetľovanie modelov, štýlov a vzorov a na podporu experimentovania.

Literatúra: Schreiber A. et al: Visualizing Software Architectures in Virtual Reality and Augmented Reality. In IEEE Aerospace Conference. Big Sky, MT, USA, 2019, pp. 1-12, doi: 10.1109/AERO.2019.8742198.
Stanček, M.: Kolaboratívne modelovanie softvéru vo virtuálnej realite. Diplomová práca, FIIT STU, Bratislava, 2021.

M. Ferenc, I. Polasek and J. Vincúr. Collaborative Modeling and Visualisation of Software Systems Using Multidimensional UML. In Proceedings of the fifth IEEE Working Conference on Software Visualization VISSOFT 2017, Shanghai.

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Poďakovanie: Rada by som týmto vyjadrila vďaku svojmu školiteľovi, doc. Ing. Ivanovi Poláškovi, PhD., za rady a pomoc, ktoré mi poskytoval. Taktiež by som sa chcela poďakovať kolegom vo výskumnej skupine, ktorí mi neváhali pomôcť pri ťažkostiach a rodine a priateľom, ktorí ma celý čas podporovali. Títo ľudia mi boli neskutočnou pomocou pri vypracovaní tejto práce.

Abstrakt

Táto práca sa venuje implementácii troch návrhových vzorov, dekorátor, zreťazenie zodpovedností a most v xUML diagrame tried a jazyku OAL. Pre každý vzor sme opísali diagram tried, implementáciu pomocou jazyka OAL a analyzovali sme zdrojový kód v jazyku Python, ktorý bol pomocou nástroja ANTLR a špeciálnej gramatiky vygenerovaný z kódu napísaného v jazyku OAL. Tieto vzory môžu poslúžiť ako učebné pomôcky pre lepšie pochopenie daných návrhových vzorov. Taktiež obohacujú bázu návrhových vzorov, ktoré sa môžu používať na testovanie funkcionality programov, ktoré sa zameriavajú na xUML a OAL. V práci sme pomocou týchto vzorov otestovali funkcionalitu nástroja AnimArch, ktorý pomáha pri xUML modelovaní.

Kľúčové slová: návrhový vzor, OAL, dekorátor, zreťazenie zodpovedností, most, AnimArch

Abstract

This thesis dedicates itself to the implementation of three design patterns, decorator, chain of responsibility and bridge in xUML class diagrams and the OAL language. For each pattern, we described the class diagram, the implementation using the OAL language and we analyzed the source code in Python, that was generated from the code written in the OAL language, using the ANTLR library and a special grammar. These patterns can be used as teaching materials to help with the understanding of the design patterns. They also expand the base of design patterns, which can be used to test the function of programs, that focus on xUML and OAL. In this work, we tested the function of the tool AnimArch, which helps with xUML modeling, using these design patterns.

Keywords: design pattern, OAL, decorator, chain of responsibility, bridge, AnimArch

Obsah

Zoznam kódov

Zoznam obrázkov

Zoznam tabuliek

Úvod

Vznik softvéru je zložitý proces. Neskladá sa len zo samotnej implementácie a napísania kódu, treba aj najprv navrhnúť a pochopiť interakciám samotných komponentov. Tomuto kroku, pochopeniu softvéru, sa venujeme v našej práci.

Návrh softvéru sa zvyčajne reprezentuje pomocou UML diagramov. Pridaním dynamickosti pomocou animácie interakcií komponentov v diagrame sa pridá vrstva chápania. Hlavne keď sa jedná o komplexnejšie interakcie, kde je viacero komponentov zodpovedných za vykonávanie rozličných úloh.

Možnosť automatickej generácie zdrojového kódu z navrhnutej animácie by mohla byť veľkým prínosom, najmä pre ušetrenie času pri implementácii a lepšie pochopenie situácie, ktorú vygenerovaný kód reprezentuje, vďaka dynamickému modelu.

Preto by mohlo byť prospešné doplniť bázu modelov a návrhových vzorov, ktoré sa môžu využívať pri takomto prístupe. Navyše pri kombinácii statického diagramu tried spolu s jej dynamickou animáciou, tieto tri návrhové vzory by sa mohli použiť pri výučbe, pričom študenti by na konkrétnych príkladoch mohli lepšie pochopiť princípy týchto vzorov.

V kapitole 1 sme opísali základné pojmy týkajúce sa problematiky a nástroje, ktoré sme pri práci využívali. Taktiež tam vysvetľujeme spôsob akým sa z OAL kódu generuje zdrojový kód v jazyku Python. V ďalších troch kapitolách sa venujeme trom návrhovým vzorom. Kapitolu 2 venujeme návrhovému vzoru dekorátor, kapitolu 3 vzoru zreťazenie zodpovedností a v poslednej kapitole 4 vzoru most. Vždy na začiatku vysvetlíme zmysel a využitia daného vzoru, potom opíšeme diagram tried príkladu a jeho OAL kód, spolu s ukážkou ako by vyzeralo dynamické spustenie tohto kódu. Nakoniec rozoberáme vygenerovaný Python kód a jeho funkcionalitu.

Kapitola 1

Základné pojmy a nástroje

Na začiatok priblížime pojmy, ktoré budeme neskôr v práci využívať a ktorým sa budeme venovať. Sú to najmä výrazy týkajúce sa metódy vývoja softvéru zhora nadol, teda sa postupuje od diagramu smerom ku výslednému zdrojovému kódu. Taktiež opíšeme jazyky, ktoré sme využívali pri implementácii návrhových vzorov. Priblížime ich syntax a možnosti využitia. S týmito jazykmi a metódou vývoja softvéru súvisia nástroje, ktoré umožňujú vytvárať alebo rôzne prezentovať diagramy. Pre tieto nástroje opíšeme ich funkcionalitu. Nakoniec opíšeme spôsob, ktorým sa generuje zdrojový kód v jazyku Python.

1.1 Základné pojmy

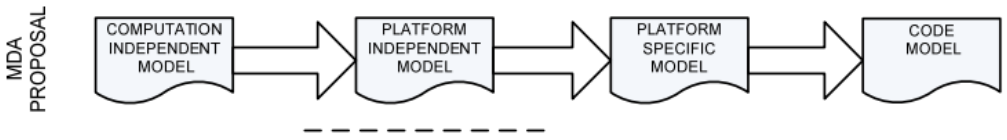
V tejto sekcii vysvetlíme pojmy súvisiace s našou prácou. Zameriame sa na metódu vývoja softvéru, *vývoj riadený modelom*, a samotné jazyky, ktoré budeme používať v praktickej časti. Tieto jazyky sú *UML*, *xUML* a *OAL* a súvisia so spomínanou metódou vývoja softvéru.

1.1.1 Model Driven Development

Vývoj riadený modelom (skrátene **MDD**, z anglického "Model Driven Development") je taký prístup, ktorý sa vyznačuje tým, že sa najskôr zameriava na výstavbu modelu softvéru, týmto sa zabezpečí zjednodušenie procesu vývoja softvéru a výsledná funkcionalita sa prejavuje ešte pred začiatkom implementácie [7].

Táto paradigma navrhuje vytvorenie viacerých modelov v rôznych úrovniach abs-

trakcie. Modely, ktoré sa vyskytujú v tejto paradigme sú **výpočtovo nezávislý model** (z anglického Computation Independent Model, ďalej označované **CIM**), **platformovo nezávislý model** (z anglického Platform-Independent Model, ďalej označované **PIM**), **platformovo špecifický model** (z anglického Platform Specific Model, ďalej označované **PSM**) a nakoniec **model kódu** (anglicky "code model", označovaný **CM**) [12]. Proces je znázornený na obrázku 1.1.



Obr. 1.1: Modely v MDD [12]

Prvý model, ktorý je vytvorený v procese vývoja softvéru je **CIM**. Je zameraný na prostredie a požiadavky systému a nerozlišuje časti modelovaného systému, ktoré budú spracované počítačom.

Ďalej nasleduje **PIM**, ktorý už berie v úvahu súčasti systému, ktoré budú spracúvané počítačom, ale nerieši technologické platformy, ktoré budú podporovať implementáciu.

Nasleduje úroveň, kde sa zameriava na platformovo špecifické záležitosti. Toto zaručuje **PSM** a ten opisuje systém, ktorý využíva konkrétne charakteristiky platformy, ktorá ho bude podporovať.

Nakoniec, **CM** je odvodený z **PSM**, tento už opisuje výsledný kód. Proces od prvého modelu až po výsledný kód môže byť náročný a zdĺhavý vďaka početnému množstvu modelov, ktoré sa musia vytvoriť.

Pri tomto prístupe modelovania je potrebné, aby boli dostupné nástroje, ktoré sú zamerané na modely softvéru a ich dôslednú analýzu, čo docieli, že sa už pri úrovni modelovania môžu skontrolovať bezchybnosť a správnosť funkcionality softvéru.

Okrem nástrojov existujú aj jazyky, pomocou ktorých sa dá vytvoriť model softvéru a následne ďalšie jazyky, ktoré umožňujú s vytvoreným modelom dynamicky pracovať. Najvýznamnejším jazykom na vytvorenie modelu je jazyk *UML* a dynamickosť sa rozvíja pomocou *xUML* a *OAL*.

1.1.2 UML

Unified Modeling Language (skrátene „**UML**“) je univerzálny jazyk určený na vizuálne modelovanie. Pomocou neho sa dajú špecifikovať, vizualizovať, konštruovať a dokumentovať artefakty softvérového systému. Je používaný na pochopenie, dizajn, prezeranie, konfigurovanie, udržiavanie a kontrolovanie informácií o danom systéme [13].

Pomocou **UML** sa dajú vytvárať rôzne diagramy. Rozdeľujú sa na *štrukturálne* a *behaviorálne*. *Štrukturálne* reprezentujú statickú štruktúru systému a jeho častí, ich interakcie na rôznych úrovniach abstrakcie. *Behaviorálne* zobrazujú dynamické správanie objektov systému v čase [4].

Pre nás je najdôležitejší *diagram tried*. Ten patrí medzi štrukturálne diagramy a zobrazuje štruktúru navrhnutého systému, jeho triedy, rozhrania a ako spolu navzájom súvisia, ako na seba vplývajú. Možné vzťahy v diagrame tried sú asociácia, generalizácia alebo závislosť.

Asociácia je sémantický vzťah, ktorý znázorňuje skutočnosť, že inštancie tried môžu byť spojené alebo skombinované logicky alebo fyzicky. Generalizácia znázorňuje dedenie medzi nadtriedou a podtriedou. Nakoniec závislosť opisuje potrebu výskytu iného elementu v modeli pre špecifikáciu alebo implementáciu **UML** elementu.

Pri komplikovanejších systémoch bývajú aj komplexné **UML** diagramy. Z tohto dôvodu niektoré nástroje na zobrazovanie takýchto diagramov používajú vrstvy na zníženie zložitosti takétoho modelu [5]. Presun diagramov z dvojrozmerného do trojrozmerného priestoru umožňuje ešte väčšie zníženie komplexnosti a lepšiu vizualizáciu modelov, pričom sa štruktúra rozloží na konkrétne komponenty, vrstvy typov, času a autorské verzie [8].

1.1.3 xUML

XtUML alebo **xUML** ako jazyk (celým významom **Executable UML**), spája podmnožinu grafických notácií z jazyka **UML** so spustiteľnou sémantikou a pravidlami časovania. Tým rozvíja možnosti **UML** diagramov, a to tak, že sa dajú spúšťať, testovať, kontrolovať a dá sa im merať výkon [14]. Na rozdiel od **UML**, čo je len súbor štandardných notácií, jazyk **xUML** je plne funkčný programovací jazyk. Umožňuje vytvoriť platformovo nezávisle modely, ktoré možno automaticky pretransformovať na

platformovo špecifické modely pomocou kompilácie, pričom sa udržiava konzistencia [2]. Skladá sa z troch konceptov, prvý abstrahuje skutočný alebo hypotetický svet a organizuje ho do formálnej štruktúry, túto reprezentujeme pomocou UML diagramov tried. Druhý popisuje životný cyklus objektov, správanie v čase, ktoré je znázornené pomocou UML stavového diagramu. Pre zmenu stavu objektov sa vykonajú nejaké procedúry, tieto sa skladajú z množiny akcií, ktoré sú popísané jazykom akcie [9]. Tieto tri koncepty sú popísané v tabuľke 1.1.

Koncept	**Nazvaný**	**Modelovaný ako**	**Vyjadrený ako**
svet obsahuje veľa vecí	dáta	triedy atribúty asociácie obmedzenia	UML diagram tried
veci majú životný cyklus	kontrola	stavy udalosti prechody procedúry	UML stavový diagram
veci niečo vykonávajú v každej etape	algoritmus	akcie	jazyk akcií

Tabuľka 1.1: Koncepty v **xUML** modeli [9]

My v práci využívame len dva tieto koncepty, abstrahované dáta vyjadrené v UML diagrame tried a akcie objektov, popísané pomocou jazyka akcií OAL.

1.1.4 OAL

Jazyk **OAL** (celým menom **Object Action Language**) je programovací jazyk, pomocou ktorého sa dajú znázorňovať dynamické vzťahy medzi triedami a ich metódami vrámci diagramu tried. Je to súčasť **xtUML**, ktorá má na starosti modelovanie spracovania, teda akcie. Jeho syntax je jednoduchá, intuitívna a podobá sa na mnoho rozšírených programovacích jazykov, ako sú Java, C++ alebo Python. Celkovo sa **OAL** snaží byť jednoduchý, abstraktný, vedomý si modelu a prekladateľný. Vďaka týmto

vlastnostiam je veľmi zrozumiteľný aj programátorom, ktorí s ním nemajú skúsenosti a dá sa rýchlo naučiť [3].

Detailovo pracuje na úrovni modelu, na ktorom sa spúšťa, je si vedomý daného modelu, aj syntaxovo, aj významovo. Spracúvajú sa mená elementov modelu, ako sú triedy, atribúty, parametre, správy. Jazyk **OAL** je nezávislý od cieľa a môže byť preložený do jazykov závislých od cieľa pomocou modelového kompilátora [3].

Ďalej popíšeme podmnožinu príkazov, ktoré ponúka **OAL** a ktoré sme využívali v implementačnej časti našej práce. Sú to príkazy, ktoré podporuje nástroj **AnimArch**, ktorý sme využívali na zobrazenie spusteného **OAL** kódu. Okrem podmnožiny štandardných príkazov [1], v minulosti boli pridané ďalšie príkazy.

V podmnožine príkazov sa nachádzajú:

- Dopyty do inštančného priestoru - je možné vytvárať a mazať inštancie tried a vytvárať a odstraňovať vzťahy medzi nimi. Následne možno manipulovať s vytvorenými objektmi.
- Výrazy - aritmetické aj logické výrazy, ktoré po vyhodnotení vrátia hodnotu, ktorú možno priradiť alebo použiť pri riadiacich konštrukciách.
- Priradenia - vytvoreným lokálnym premenným možno priradiť hodnoty, podporované typy sú integer (celé číslo), real (reálne číslo), boolean (logická premenná TRUE/FALSE), string (reťazec) a objekty tried
- Riadiace konštrukcie - možno pomocou nich riadiť tok vykonávania, patria medzi ne cykly, podmienky a riadené cykly.
- Zoznam - vytvorenie zoznamu prvkov jedného typu a možnosť pridávať a odstraňovať prvky.
- Vstup a výstup - možno interaktívne získať hodnotu od používateľa. Taktiež možno poskytnúť výstup vypísaním hodnoty na konzolu.
- Animačný krok - predstavuje volanie medzi metódami.

Príklady príkazov sme opísali v tabuľke 1.2.

Konštrukcia	**Zápis v OAL**
Dopyty do inštančného priestoru	create object instance cube1 of Cube; select many cubes from instances of Cube; delete object instance cube1;

Výrazy a priradenia	colorcube1.object = cube1;
	ziphandler1.extension = "zip";
	x = 3 + 7/(5 - 8);

Riadiace konštrukcie	if (file.format == self.extension)
	  self.handle(file);
	end if;
	while (x < 50);
	  a = a + x;
	end while;

Zoznam	create list cubeList of Cube; add cube1 to cubeList; remove cube1 from cubeList;
Vstup a výstup	x = int(read("Please write a number."); write("Your number was ", x);
Animačný krok	colorcube1.addColor("red"); rotationcube1.show();

Tabuľka 1.2: Príklady príkazov v **OAL**

1.2 Návrhové vzory

Vývoj objektovo orientovaného softvéru je v niektorých situáciách zložitý, vyskytujú sa všelijaké ťažkosti a problémy. Tieto prekážky ale často bývajú podobného charakteru, tak na ne existujú podobné, až v podstate rovnaké riešenia. Preto už netreba vymýšľať nijaké originálne riešenie, ale stačí použiť už overené šablóny, ktoré sa volajú **návrhové vzory** [6]. Tieto vzory poskytujú jednotné riešenie pre neustále sa opakujúce problémy. V skutku pozostávajú zo štyroch prvkov:

- Meno vzoru – pomenúva dizajnový problém, jeho riešenia a následky v jednom slove alebo slovnom spojení. Zjednodušuje rozprávanie o konkrétnej problematike

tým, že problém aj riešenie označuje jedným slovom.

- Problém – opisuje, kedy sa má aplikovať daný vzor. Vysvetľuje situáciu, v ktorej by sa mal využiť konkrétny *návrhový vzor*. Môže to byť reprezentované ako zoznam podmienok, ktoré musia byť splnené predtým ako sa môže vzor použiť.

- Riešenie – popisuje samotný vzor. Objekty, ktoré sú jeho súčasťou, vzťahy medzi jednotlivými objektmi, ich zodpovednosti a ako medzi sebou spolupracujú. Neopisuje konkrétnosti, ale abstraktný pohľad na to ako daná množina prvkov spolu pracujú a riešia problém, keďže vzory sú univerzálne pre mnoho rôznych situácií.

- Následky – sú to výsledky a kompromisy za využitie *návrhového vzoru*. Poskytujú pohľad na obmedzenia po využití rôznych vzorov, v zmysle priestorových alebo časových nárokov, čo má dopad na flexibilitu, rozsiahlosť a prenosnosť celého systému.

Keďže *návrhové vzory* sú šablónou riešenia častých problémov, tak ich výhodou je, že sú univerzálne pre rôzne platformy a aj programovacie jazyky. Takže sa netreba obmedzovať rôznymi osobitosťami konkrétnych programovacích jazykov pri návrhu softvéru, treba ich brať v úvahu až pri implementácii. Navyše sa navzájom dopĺňajú a prelínajú, tak na riešenie jedného problému sa môže vyberať z viacerých možností [6].

Návrhových vzorov je veľký počet, tak sa delia do štyroch kategórií (v minulosti boli len tri, ale s pribúdaním vzorov sa rozšírili o ďalšiu). Sú rozdelené podľa ich účelu:

1. Generatívne (creational)

   - zameriavajú sa na proces vytvorenia objektov
   - patria sem abstraktná továreň, abstraktná metóda, prototyp, unikát (singleton), staviteľ (builder) a ďalšie

2. Štrukturálne (structural)

   - rozoberajú zloženie tried alebo objektov
   - patria sem kompozit, dekorátor, fasáda (facade), zástupca (proxy), most, adaptér a ďalšie

3. Behaviorálne (behavioral)

    - upresňujú akým spôsobom si majú triedy a objekty deliť zodpovednosti a interagovať medzi sebou
    - patria sem iterátor, mediátor, memento, stratégia, stav, zreťazenie zodpovedností, pozorovateľ (observer) a ďalšie

4. Súbežné (concurrency)

    - zaoberajú sa viacvláknovými aplikáciami
    - patria sem reaktor, proaktor (proactor), zámok na čítanie a zápis, plánovač úloh (scheduler) a ďalšie

V práci sa bližšie venujeme trom vzorom, to sú *dekorátor*, *zreťazenie zodpovedností* a *most*. Tie sú samostatne popísané vo vlastných kapitolách. *Dekorátoru* je venovaná kapitola 2, *zreťazeniu zodpovedností* kapitola 3 a *mostu* kapitola 4.

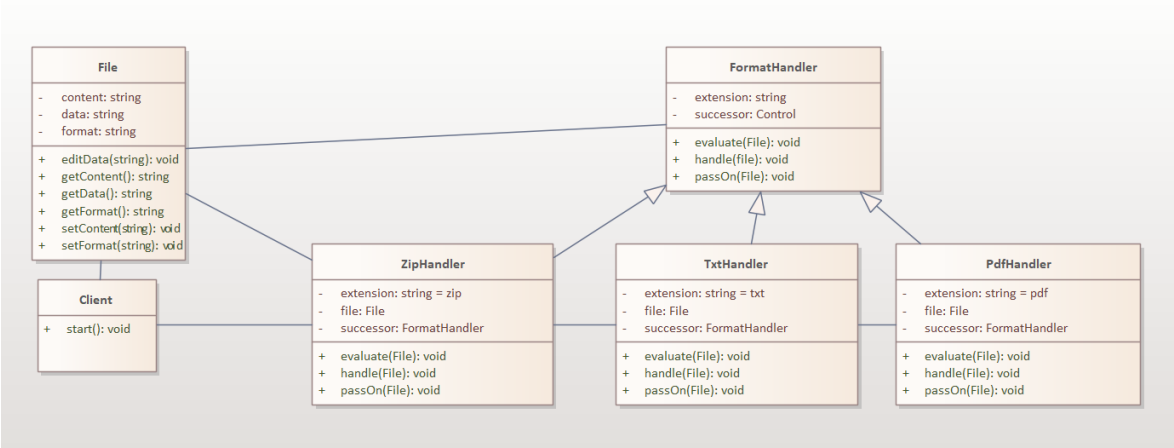
1.3 Použité nástroje

Počas výroby implementácie sme používali niekoľko nástrojov zameraných na vývoj pomocou **MDD**. Vďaka ním sme vedeli vytvárať diagramy *návrhových vzorov* a následne zobraziť diagramy spolu s takzvanou animáciou vytvorenou pomocou jazyka **OAL** na danom diagrame.

1.3.1 Enterprise Architect

Prvý nástroj, ktorý sme v práci použili je **Enterprise Architect**. Je to integrovaná modelovacia platforma od firmy Sparx Systems. Slúži na vizualizáciu, analýzu, modelovanie, testovanie a udržiavanie systémov, softvéru, procesov a architektúr v podniku [15]. Poskytuje mnoho funkcionalít, ale my sme využili malú podmnožinu.

My sme tento nástroj využili na vytváranie diagramov tried našich vzorov, príklad je na obrázku 1.2. Nástroj poskytuje možnosť pridať elementy, ktoré obsahujú aj triedy a rozhrania. Tieto elementy možno pospájať rôznymi vzťahmi, medzi ktoré patria aj asociácia a generalizácia. Triedam a rozhraniam možno pridať atribúty s dátovými typmi a funkcie s návratovými hodnotami a parametrami.



Obr. 1.2: Diagram tried vytvorený v **Enterprise Architect**

Diagram sa dá extraktovať vo viacerých formátoch, pre nás je podstatný formát XMI 2.1, ktorý je podporovaný nástrojom **AnimArch**. Diagramy vytvorené v nástroji **Enterprise Architect** sme otvorili a upravovali v tomto nástroji.

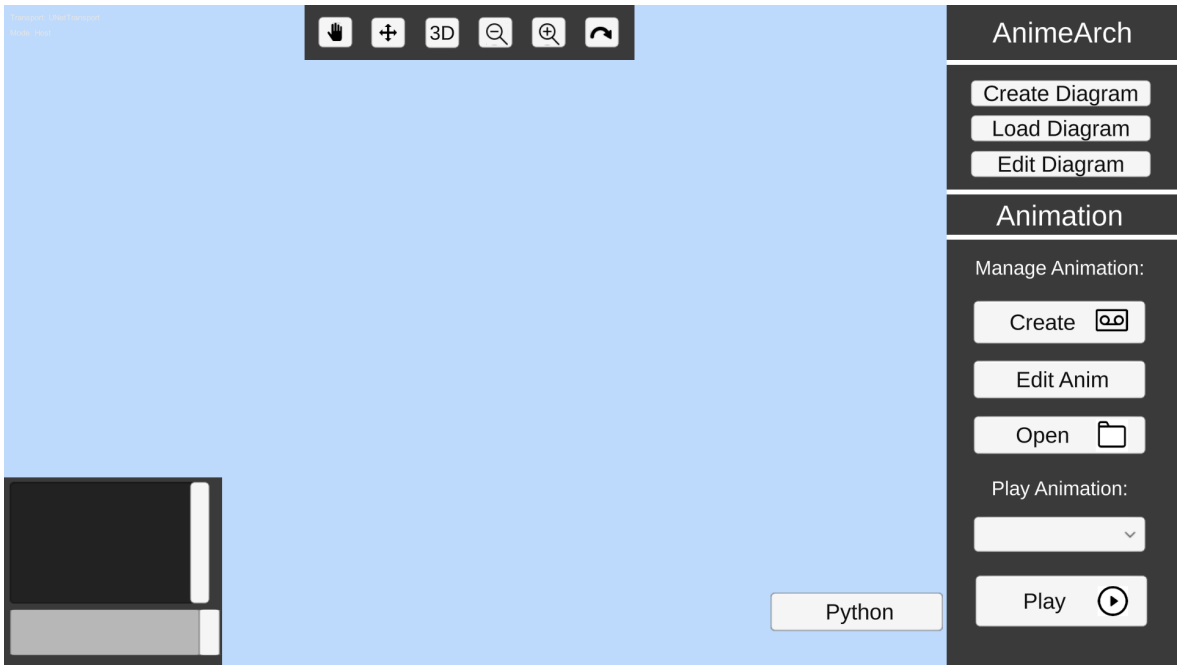
1.3.2 AnimArch

V práci sme prevažne využívali funkcionality nástroja **AnimArch**, jeho používateľské rozhranie je na obrázku 1.3.

Je to Unity aplikácia napísaná v jazyku C#. Jeho hlavným cieľom je, aby používateľovi pomohol pri pochopení štruktúry navrhnutého kódu, pričom umožňuje uskutočniť prístup **MDD**. Funguje tak, že zobrazuje diagram tried a animácie, ktoré znázorňujú interakcie medzi metódami daných tried [10].

Diagramy tried vie importovať z nástroju **Enterprise Architect**, ktoré sa dajú upravovať alebo taktiež ponúka možnosť vytvoriť jednoduché diagramy, pozostávajúce z tried a vzťahov asociácia, agregácia, generalizácia. Triedam umožňuje priradiť aj atribúty a funkcie s parametrami. Táto funkcionalita je jemne obmedzená, ale rozvíja sa a zatiaľ postačí na vytvorenie väčšiny základných diagramov.

Animácie znázorňujú spustený **OAL** kód. **AnimArch** umožňuje vytvárať jednoduché animácie pomocou grafického prostredia. Tieto dokážu zobraziť najmä vytváranie objektov a spúšťanie metód. Jednoducho stačí označiť metódu v triede a nasledujúcu metódu v inej triede, ktorú spúšťa. Ďalej pomocou priamej editácie kódu zobrazeného v okienku na obrazovke, používateľ dokáže manuálne doplniť aj komplexnejšie príkazy. Keďže jazyk **OAL** je veľmi intuitívny, nejaví sa to ako veľký problém, ani pri prvých



Obr. 1.3: Rozhranie nástroja **AnimArch**

použitiach. **AnimArch** potom dokáže zobrazovať aj komplexnejšie **OAL** programy, používajúce logiku a viacej objektov. Animácia je uložená vo formáte JSON, pričom je rozdelená podľa tried, ktoré majú svoje atribúty a metódy. Ku metódam sú priradené parametre a kód, ktorý je napísaný v jazyku **OAL**. Tento formát umožňuje kompatibilitu, jednoduché spúšťanie a pracovanie s vytvoreným kódom.

Aj diagramy, aj animácie je možné vytvárať, ukladať a neskôr načítať zo súboru. Nástroj poskytuje katalóg *návrhových vzorov*, na ktorých si používateľ môže vyskúšať jeho používanie. Zatiaľ obsahuje príklad mediátora, pozorovateľa a abstraktnej továrne. Túto databázu diagramov a animácií v tejto práci rozširujeme o tri ďalšie vzory. Používateľ si potom bude môcť pre konkrétny *návrhový vzor* otvoriť diagram a animáciu, a pri jej spustení pochopiť interakcie medzi komponentami systému implementovaného podľa daného *návrhového vzoru*. Toto umožní rýchlejšie a detailnejšie pochopenie zmyslu vzťahov, základné myšlienky a možnosti využitia ponúkaných *návrhových vzorov*.

1.4 Generovanie Python kódu

Po implementovaní *návrhových vzorov* v jazyku **OAL** vygenerujeme zdrojový kód v jazyku Python. Postup, ktorým sa toto dosiahne a použité nástroje teraz popíšeme.

Využili sme už existujúce nástroje, čím sú nástroj **ANTLR** a špeciálna gramatika na preklad **OAL** kódu do Python kódu. Kedže parser, ktorý v práci využijeme už existuje, tak len zhruba popíšeme proces ako funguje a ako bol vytvorený.

1.4.1 Nástroj ANTLR a gramatika jazyka OAL

ANTLR v4 je generátor *syntaktického analyzátora* (parsera), ktorý sa môže použiť na čítanie, procesovanie, vykonanie alebo prekladanie štruktúrovaných textov alebo binárnych súborov. Z gramatiky, ktorá formálne opisuje jazyk, nástroj **ANTLR** generuje parser pre daný jazyk, ktorý automaticky vyrobí *parsovací strom*, ktorý opisuje ako sa gramatika zhoduje so vstupom. Taktiež automaticky generuje prechádzače stromov, ktoré sa môžu použiť na navštívenie uzlov v strome a vykonanie kódu aplikácie [11].

Na využitie je potrebná vstupná gramatika. Gramatika, ktorá sa využila pozostáva z dvoch typov pravidiel: **pravidlá lexera** a **pravidlá parsera**.

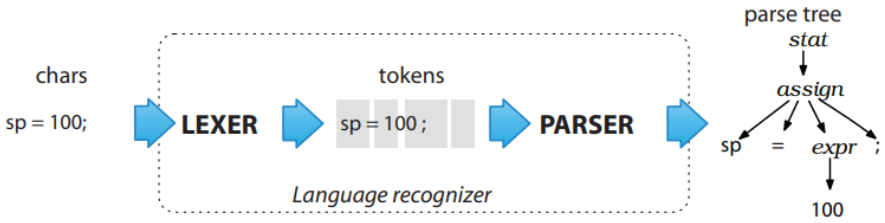
Najprv nastane proces lexikálnej analýzy, pri ktorom sa vstupný reťazec prechádza po znakoch a tie sa zoskupujú do zmysluplných celkov, nazývaných *tokeny*. **Pravidlá lexera** definujú *tokeny*, pričom pozostávajú z typu *tokenu* a textu, ktorý reprezentuje daný *token* [10]. Príklad takétoho pravidla je:

$$NAME : [a - zA - Z\_\#][a - zA - Z0 - 9\_\#]*;$$

Token NAME reprezentuje mená premenných, tried, metód a atribútov. Tieto sa môžu skladať z písmen malej alebo veľkej abecedy, podčiarkovníka (_), mriežky (#) a čísiel, pričom nimi nemôže začínať.

Druhý typ pravidiel sú **pravidlá parsera**. Počas parsovania sa využívajú vytvorené *tokeny* a výsledkom je dátová štruktúra, *parsovací strom*, reprezentujúca štruktúru vstupnej vety [11]. Na obrázku 1.4 je reprezentovaný postup ako sa zo vstupu *sp = 100;* vytvorí *parsovací strom*.

Táto gramatika reprezentuje jazyk **OAL**. Pomocou nej ako vstup do nástroja **ANTLR** sa vygenerujú súbory reprezentujúce *lexer*, *parser* a *visitor*. Tento *visitor* umožňuje prechádzať po vygenerovanom *parsovacom strome* a vykonávať kód aplikácie, v našom prípade generovať kód v jazyku Python z kódu napísaného v **OAL** [11][10].



Obr. 1.4: Tok lexera a parsera na vstupe *sp = 100*[11]

1.4.2 Parsovanie do Python

Proces vygenerovania kódu teraz popíšeme. Vykonáva sa pomocou jednoduchej aplikácia napísanej v C#, ktorá využíva súbory vytvorené nástrojom **ANTLR**.

Najprv sa vo vhodnom formáte zapíše OAL kód. Ten je rozdelený podľa tried, ktoré obsahujú aj atribúty triedy. Triedy obsahujú metódy, ktoré majú parametre a samotný kód.

Pre vhodné spracovanie OAL kódu, sa vytvoria mapy, do ktorých sa ukladajú tieto údaje. Tieto sú *Classes*, ktorá mapuje meno triedy s mapou vytvorenou pre každú triedu, ktorá mapuje meno metódy s jej OAL kódom. Ďalšia mapa je *ClassesAttributes*, ktorá priraďuje meno triedy k jej atribútom. Nakoniec je mapa *MethodsParameters*, ktorá ku menu metódy priraďuje jej parametre. Takáto štruktúra slúži na zjednodušené parsovanie kódu. Kód sa zapisuje do bežného *StringBuilder* v jazyku C#, pomocou ktorého výsledný kód zapíšeme do súboru, ktorý bude reprezentovať spustiteľný Python kód.

Postupne sa prechádza triedami v mape *Classes*. Najprv sa zapíše definícia triedy a základný konštruktor, ktorý obsahuje deklarovanie atribútov triedy s priradenou hodnotou *None*. Tieto atribúty sa získajú prechádzaním hodnôt mapy *ClassesAttributes* pre konkrétnu triedu.

Nakoniec sa riešia metódy. Tieto sú uložené v mape vytvorenej pre triedu, ktorou sa práve prechádza. Každá metóda sa najprv definuje, pričom sa berú v úvahu parametre uložené v *MethodsParameters*. Po definovaní sa zapíše preložené telo metódy. Ak je telo metódy prázdne, tak sa zapíše *" pass"*, pre správnu reprezentáciu prázdnej metódy v Pythone. Ak obsahuje **OAL** kód, tak ten sa postupne upraví a preparsuje na ekvivalentný Python kód. Kód popisujúci tento proces je na obrázku 1.5.

Najprv sa pomocou *AntlrInputStream* zo základného textu vytvorí pole znakov.

```
{
    Code.AppendLine("        pass\n");
}
else
{
    AntlrInputStream inputStream = new AntlrInputStream(methodItem.Value);
    NewGrammarLexer lexer = new NewGrammarLexer(inputStream);
    CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);
    NewGrammarParser parser = new NewGrammarParser(commonTokenStream);
    OALToPythonVisitor visitor = new OALToPythonVisitor(new List<string>());
    String result = visitor.Visit(parser.lines());

    Code.AppendLine(result);

}
```

Obr. 1.5: Preklad kódu v jazyku OAL do jazyka Python

Výstup tohto prejde cez *NewGrammarLexer*, ktorý prejde cez *CommonTokenStream*. Tento vráti postupnosť *tokenov*, ktoré nakoniec prejdú cez *NewGrammarParser*. Premenná *parser* reprezentuje vytvorený *parsovací strom* vstupného textu. *OALToPythonVisitor* postupne navštívi uzly v strome a jeho výstupom je kód v jazyku Python.

Celý tento kód, ktorý sa postupne generoval sa nakoniec zapíše do súboru, ktorý je v jazyku Python a možno ho spustiť.

Kapitola 2

Implementácia návrhového vzoru Dekorátor

Prvý návrhový vzor, ktorý sme implementovali je *dekorátor*. Najprv popíšeme na čo je užitočný a jeho základnú podstatu, potom opíšeme diagram triedy nášho príkladu, nakoniec popíšeme dynamický príklad, jeho animáciu a preklad do Python kódu.

2.1 Návrhový vzor dekorátor

Návrhový vzor *dekorátor* patrí medzi štrukturálne vzory, teda rieši problémy súvisiace so zložením tried a ako majú byť rozdelené zodpovednosti. V tomto prípade sa konkrétne jedná o pridávanie zodpovedností triedam dynamicky [6].

V aplikácii môžeme potrebovať rôzne inštancie jednej triedy s rozličnými funkcionalitami. Na to, aby sme nemali veľké množstvo podtried, pričom každá by sa len líšila v časti kódu, ktorá rieši pridanú funkcionalitu, tak pre každú zodpovednosť vytvoríme podtriedu len pre ňu samotnú. Týmto spôsobom sa triedy z veľkej časti neopakujú, každá má na starosti len jednu vec a poskytuje to možnosť aj viacnásobne pridať nejakú vlastnosť.

Pridaná zodpovednosť môže byť nejaká vlastnosť, ako je napríklad nejaký atribút, nejaká funkcionalita alebo vykonanie nejakej akcie.

Dekorátor sa využíva v situáciách, keď treba pridať zodpovednosti jednotlivým objektom dynamicky a transparentne, bez obmedzenia iných objektov danej triedy, pri zodpovednostiach, ktoré môžu byť odobrané alebo keď je nepraktické rozširovanie cez

vytváranie podtried, napríklad v situácii kedy by malo vzniknúť nemalé množstvo tried [6].

Používa sa tak, že sa vytvorí objekt triedy, ktorej chceme pridať zodpovednosť, následne sa vytvorí konkrétny dekorátor s funkčnosťou, ktorú chceme pridať, pričom mu odovzdáme predošlý vytvorený objekt. Keďže tieto objekty majú rovnakú nadtriedu alebo rozširujú rovnaké rozhranie, tak možno nabaľovať funkcionality pomocou rôznych *dekorátorov*.

Výhody tohto sú, že to poskytuje vyššiu flexibilitu, než keby sa mala pridať zodpovednosť dedením, vyhýba sa pomocou nemu triedam, ktoré sú preplnené funkcionalitami, ktoré sa nemusia využiť. Nevýhody sú, že sa vytvára množstvo menších objektov, vďaka ktorým sa systém môže stať neprehľadným. Ďalšou nevýhodou je, že dekorátor s jeho komponentom nie sú identické a treba s nimi rôzne pracovať [6].

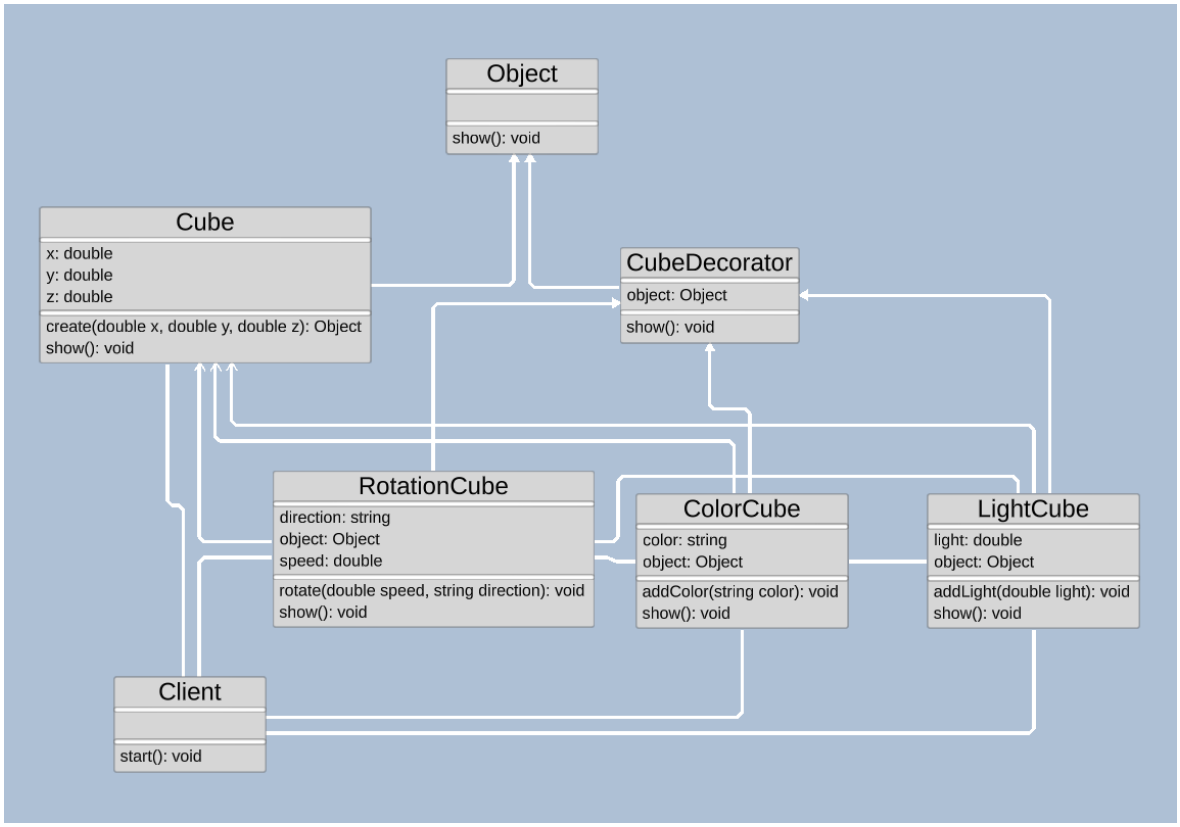
2.2 Diagram tried Cube

My sme si vybrali *dekorátor* ako jeden z návrhových vzorov, ktoré ideme implementovať, pretože pomocou animácie z **OAL** kódu dobre vidno, ktorá trieda kedy vykonáva operácie. Po vytvorení, objekt má „vrstvovú“ štruktúru, a pri vykonaní metódy sa najprv vykoná najvrchnejšia trieda, tá zavolá metódy svojho komponentu, čo je nižšia vrstva, až kým je komponent základný objekt, ktorý nevolá nijaký iný objekt. Túto postupnosť možno pekne znázorniť v animácii na diagrame tried.

Pri implementácii *dekorátoru* sme zvolili aplikáciu, ktorá vytvára 3D objekty. Jeho diagram triedy je na obrázku 2.1. Pozostáva z rozhrania **Object**, triedy **Cube** a dekorátorov **CubeDecorator**, **RotationCube**, **ColorCube** a **LightCube**. Pomocou týchto *dekorátorov* sa použitím ich pridanej metódy pridávajú atribúty a pomocou rozvíjanej metódy *show()* sa dá ukázať ako pracuje objekt vytvorený pomocou *dekorátoru*.

Tento príklad je grafický, pričom *dekorátory* budú pridávať vytvoreným kockám atribúty farby, rotácie a osvietenia. Konkrétny príklad môže navodzovať predstavu, že *dekorátor* je iba grafický vzor, čo nie je pravda. Pomocou *dekorátora* sa môže pridať vykonanie nejakej akcie alebo funkcionality triede.

Rozhranie **Object** opisuje 3D objekty, pričom poskytuje metódu *show()*, ktorá zobrazí vytvorený objekt. Pre obmedzenia v príkazoch jazyku **OAL**, my budeme objekty



Obr. 2.1: Diagram tried v *AnimArchu* pre návrhový vzor dekorátor

reprezentovať textovo, pomocou atribútov základného objektu *x*, *y*, *z* a ďalších ktoré pridáme pomocou *dekorátorov*. Metóda *show()* teda vypíše na konzolu textovú reprezentáciu objektu.

Trieda **Cube** implementuje metódu z rozhrania **Object**. Navyše poskytuje metódu *create(double x, double y, double z)*, ktorá priradí súkromným premenným triedy *x*, *y*, *z* dané hodnoty a tým vytvorí objekt reprezentovaný týmito hodnotami. Metóda *show()* vypíše na výstup práve hodnoty týchto parametrov.

Dekorátor **CubeDecorator** tiež implementuje rozhranie **Object**, mohla by to byť abstraktná trieda, ktorá slúži len pre ďalšie *dekorátory* alebo by tam ani nemusela byť. Samotná neimplementuje žiadne z metód, ale obsahuje objekt *object*, ktorý reprezentuje objekt, ktorý rozširuje rovnaké rozhranie a ku ktorému sa bude pridávať funkcionalita.

Od triedy **CubeDecorator** dedia triedy **ColorCube**, **RotationCube** a **LightCube**.

ColorCube okrem metódy *show()* obsahuje aj metódu *addColor(string color)*. Atribútu triedy *color* priradí farbu špecifikovanú v parametri metódy *color*, čo sa následne prejaví pri metóde *show()*, kde sa táto farba vypíše pri opísaní objektu.

RotationCube má pridanú metódu *rotate(double speed, string direction)*, ktorá priradí objektu dve vlastnosti, rýchlosť a smer otáčania, ktoré sa uložia do atribútov triedy *speed* a *direction*. Taktiež sa prejavia pri metóde *show()*, kedy sa vypíšu ako reprezentácia objektu.

Nakoniec, **LightCube** má podobne naviac jednu metódu, ktorá sa volá *addLight(double light)*, s reálnym parametrom *light*, ktorý reprezentuje aké osvietenie sa má pridať objektu. Pridá sa ako atribút reprezentujúci objekt a vypíše sa pri zavolaní metódy *show()*.

V zhrnutí, *dekorátory* v našom príklade pridajú atribúty, ktoré sa zobrazia pri zavolaní metódy *show()* príslušnej triedy.

2.3 OAL kód a animácia využitia dekorátorov

Naša implementácia v **OAL** popisuje situáciu vytvorenia objektu a následným pracovaním s daným objektom. Toto považujeme za najčastejšiu situáciu, s ktorou sa stretneme pri návrhovom vzore *dekorátor*. Na začiatku postupne vytvorí jednoduchý objekt, potom sa pridáva vybraná funkcionalita a pracuje sa so zloženým objektom. Implementácia metódy *start()* v triede **Client** je napísana v kóde 2.1. Ďalej sme bodovo opísali kroky, ktoré sa v príklade vykonávajú.

```
1  create object instance cube_1 of Cube;
2  cube_1.create(5, 6, 4);
3
4  create object instance colorcube_1 of ColorCube;
5  colorcube_1.object = cube_1;
6  colorcube_1.addColor("red");
7
8  create object instance rotationcube_1 of RotationCube;
9  rotationcube_1.object = colorcube_1;
10 rotationcube_1.rotate(3.2, "left");
11 rotationcube_1.show();
```

Kód 2.1: Implementácia metódy *start()* triedy **Client** v **OAL** kódev *dekorátore*.

Animácia pozostáva z nasledujúcich krokov a je znázornená na obrázku 2.2, ktorý vystihuje najdôležitejšie kroky, celá sa nachádza v prílohe.

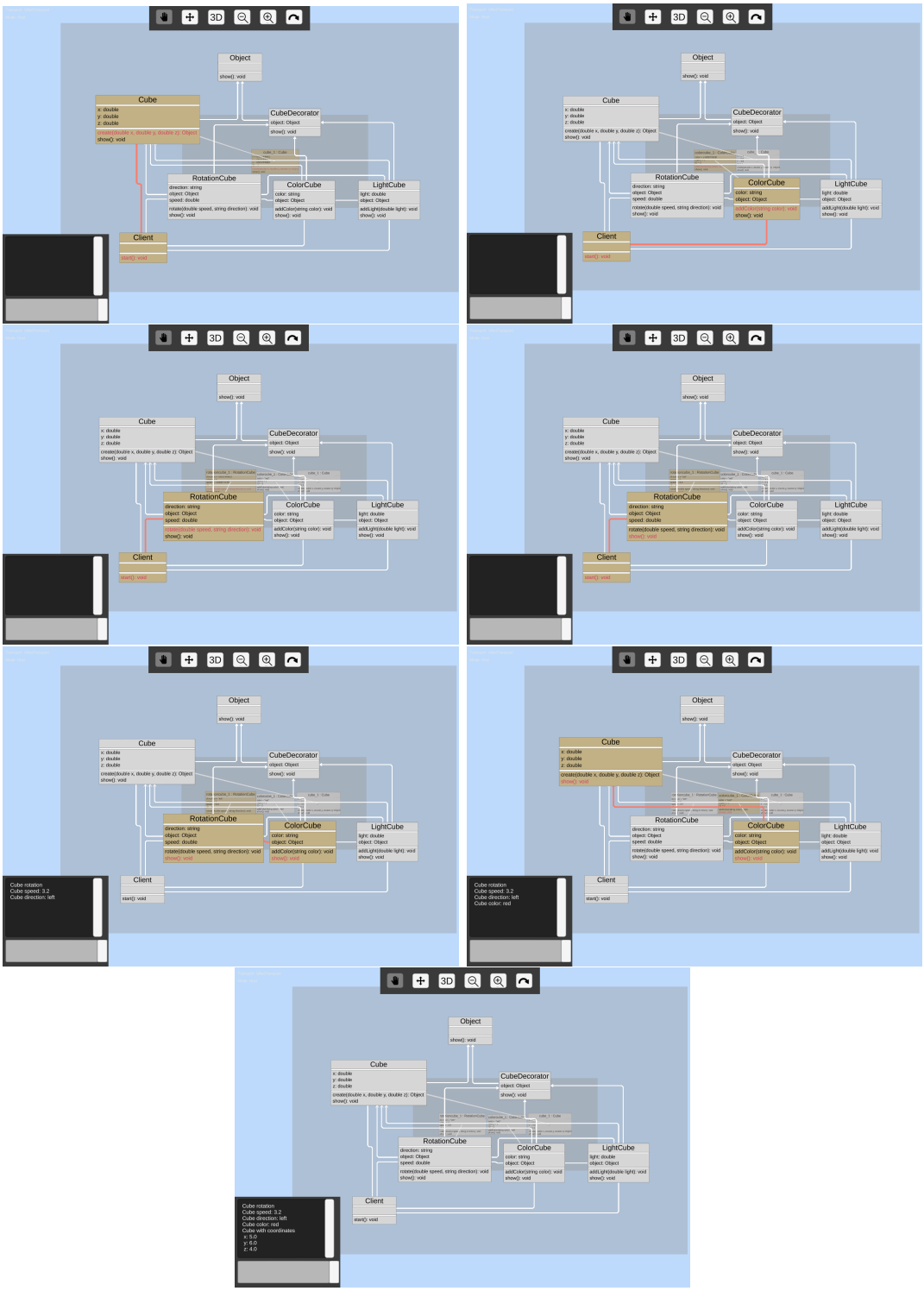
1. Vytvorenie inštancie triedy **Cube** a pridelenie hodnôt rozmerov
2. Postupné vytváranie dekorátorov:
   - Inštancia triedy **ColorCube** s určenou farbou
   - Inštancia triedy **RotationCube** s prideleným smerom a rýchlosťou
3. Vypísanie vytvoreného objektu:
   - Vypísanie objektu **RotationCube**
   - Vypísanie objektu **ColorCube**
   - Vypísanie základného objektu **Cube**

Situácia sa vyvíja tak, že sa najprv vytvorí objekt *cube_1* triedy **Cube**, s rozmermi $x = 5, y = 6, z = 4$, jemu sa pridá červená farba pomocou *dekorátora*. To vyzerá tak, že sa vytvorí objekt *colorcube_1* triedy **ColorCube** pomocou už vytvoreného *cube_1* a zavolá sa jeho metóda *addColor(„red“)*. Ďalej sa ešte objektu *colorcube_1* pridá rotácia s rýchlosťou *3.2* smerom vľavo. Toto sa znovu docieli pomocou vytvorenia objektu *rotationcube_1* triedy **RotationCube** a zavolaním jeho osobitej metódy *addRotation(3.2, „left“)*. Teraz možno pracovať s ľubovoľným z týchto troch objektov v podstate rovnako. My sme zvolili pracovať s najkomplexnejším z nich, teda s *rotationcube_1*.

Animácia tohto postupu sa prejaví tak, že najprv sa vysvieti počiatočná metóda *start()* v triede **Client**. Táto reprezentuje používateľa, ktorý sa rozhoduje, že aký objekt chce vytvoriť a ako s ním následne chce pracovať. Keďže najprv vytvárame objekt triedy **Cube**, tak sa vysvieti väzba medzi týmito triedami. Toto sa docieli tým, že sa volal **OAL** príkaz *create instance cube_1 of Cube;*. Následne sa vysvieti metóda *create(5, 6, 4)* tejto triedy **Cube**.

Podobne sa znázorní vytvorenie objektov *colorcube_1* a *rotationcube_1* a pridanie jednotlivých atribútov. V **OAL** kóde je to implementované ekvivalentne. Vždy trieda **Client** volá vytvorenie objektu určenej triedy, čo sa znázorní vysvietením väzby medzi triedami a volanej metódy.

Toto obsahuje fáza vytvorenia objektov, nasleduje fáza ich zobrazenia.



Obr. 2.2: Animácia v *AnimArchu* pre návrhový vzor dekorátor

Po vytvorení objektu ho teraz zobrazíme. Na inštanciu *rotationcube_1* zavoláme metódu *show()*. V tomto kroku vidno ako naozaj funguje návrhový vzor *dekorátor*. Najprv sa zavolá metóda *show()* v triede **RotationCube**, čo vypíše na konzolu smer a rýchlosť rotácie a zavolá sa metóda *show()* triedy jeho komponentu, čo je vykonané kódom 2.2.

```
1 write("Cube rotation");
2 write("Cube speed: ", self.speed);
3 write("Cube direction: ", self.direction);
4 self.object.show();
```

Kód 2.2: Implementácia metódy *show()* triedy **RotationCube** v **OAL** kóde.

Animácia týchto krokov vyzerá tak, že sa vysvieti trieda **RotationCube** a jej metóda *show()*, od nej sa vysvieti väzba k objektu uloženému v atribúte triedy **RotationCube**, *object*. V tomto prípade je to inštancia triedy **ColorCube**, kedže sa objekt *rotationcube_1* vytváral z objektu *colorcube_1*. V tejto triede sa vysvieti metóda *show()*, ktorá vypíše farbu a zavolá metódu *show()* už základnej triedy **Cube**, z ktorej bol objekt vytvorený. Táto metóda v triede **Cube** vypíše rozmery kocky a ďalej už nemá čo volať, tak sa ukončí beh metódy *show()*. Tieto kroky sa v animácii preukážu podobne ako v triede **RotationCube**.

Na tomto príklade vidno, že pri práci s komplexným objektom vytvoreným pomocou *dekorátora* sa pracuje v „opačnom“ poradí ako sa vytváral. Všetky kroky nerieši len jedna trieda, ale *dekorátory* sú zodpovedné za svoje časti.

Ako sme už spomínali v podkapitole 1.1.4, náš jazyk **OAL** obsahuje obmedzené množstvo príkazov, medzi ktoré patria aj príkazy na vytváranie, logické podmienky, aritmetické operácie a boli pridané aj príkazy na vypísanie na výstup a čítane zo vstupu.

V našej implementácii väčšina metód priradí hodnoty parametrov metódy parametrom triedy. Toto sa vykoná pomocou jednoduchého príkazu priradenia *self.class_param = method_param;*. Toto je odzrkadlenie našej reprezentácie objektu, keďže sa reprezentuje pomocou atribútov, ktoré sa vypíšu v textovej forme.

Na implementáciu príkazu *show()* sa využíva pridaný príkaz v **OAL** *write(„text“,*

self.atr).

2.4 Vygenerovaný Python kód

Po tom, čo sme vytvorili diagram tried, ktorý popisuje statickú štruktúru softvéru a **OAL** kód, ktorý pomocou animácie dynamicky ukazuje beh programu, sme vygenerovali zdrojový kód v programovacom jazyku Python, ktorý sa dá spustiť.

Tento kód sme vygenerovali postupom, ktorý sme podrobne opísali v podkapitole 1.4. Využili sme *OALToPythonVisitor* vytvorený nástrojom **ANTLR**. Postupovali sme tak, že sme vytvorili jednoduchý program v jazyku C#. Po spustení programu sa vygeneroval zdrojový kód do súboru.

Nie všetko sa dalo vyriešiť len parserom. Problém sa naskytol pri prázdnom tele metódy v **OAL** kóde. Po parsovani zostali prázdne, ale toto sa v jazyku Python má zapisovať pomocou príkazu *" pass"*, tak sa nedal spustiť kód. Vyriešili sme to pridaním podmienky pri parsovaní, že ak je telo metódy prázdne, tak sa v Pythone zapíše *" pass"*. Taktiež sme pridali príkaz na spustenie kódu *Client.start()*. Po týchto úpravách bol kód rovno po vygenerovaní spustiteľný.

Výsledný Python kód bol celkom priamočiarym prekladom **OAL** kódu, čo vidno na obrázku 2.3.

```
create object instance cube_1 of Cube;
cube_1.create(5, 6, 4);
create object instance colorcube_1 of ColorCube;
colorcube_1.object = cube_1;
colorcube_1.addColor("red");
create object instance rotationcube_1 of RotationCube;
rotationcube_1.object = colorcube_1;
rotationcube_1.rotate(3.2, "left");
rotationcube_1.show();
```

```
def start(self):
    cube_1=Cube()
    cube_1.create(5, 6, 4)
    colorcube_1=ColorCube()
    colorcube_1.object = cube_1
    colorcube_1.addColor("red")
    rotationcube_1=RotationCube()
    rotationcube_1.object = colorcube_1
    rotationcube_1.addRotation(3.2, "left")
    rotationcube_1.show()
```

Obr. 2.3: *Vľavo:* napísaný OAL kód. *Vpravo:* kód vygenerovaný prekladom z OAL do Pythonu

Keďže v jazyku **OAL** sa vždy vytvárajú inštancie pomocou príkazu *create object instance object_1 of class Object;*, tak sa v Pythone vždy vytvárali objekty pomocou volania *object_1 = Object()*, teda sa využíval iba základný konštruktor a treba dodatočne priradiť funkcie na priradenie atribútov. Týmto sa predlžoval možný kód.

Vygenerovaný kód mal funkcionalitu ekvivalentnú tej implementovanej v **OAL**

kóde. Teda sa dá postupom od diagramu tried s **OAL** kódom vytvárať funkčný kód. Rozdiel spočíva v tom, že sa naraz vypíšu výsledky, pričom pri animácii sa zreteľne znázorňuje, že ktorá trieda čo v ktorom kroku vykonáva. Dobre sa dopĺňajú, **OAL** kód je dobrý na pochopenie softvéru a Python kód na skutočné použitie vytvorenej aplikácie.

Kapitola 3

Implementácia návrhového vzoru Zreťazenie zodpovedností

V tejto kapitole sa budeme venovať návrhovému vzoru *zreťazenie zodpovedností*. Najprv tento návrhový vzor priblížime, potom opíšeme diagram tried pre príklad implementácie tohoto vzoru. Potom vysvetlíme implementáciu pomocou OAL kódu a ako sa to prejaví pomocou animácie. Nakoniec rozoberieme vygenerovaný Python kód tohoto príkladu.

3.1 Návrhový vzor zreťazenie zodpovedností

Návrhový vzor *zreťazenie zodpovedností* patrí medzi behaviorálne návrhové vzory, ktoré upresňujú akým spôsobom komponenty spolu v systéme interagujú a ako si delia zodpovednosti [6]. Už z názvu je jasné, že rieši pridelenia zodpovedností v reťazovej forme.

Niekedy nastáva situácia, že je viacero komponentov systému a každý má schopnosť riešiť osobitný podnet. Tento podnet sa musí dostať ku správnemu komponentu a môže, a nemusí sa dostať k ostatným, čo by umožnilo spracovanie viacerými komponentami. Toto sa rieši tak, že sú komponenty nadviazané jeden za druhým a prvému sa pošle požiadavka. Ten buď vyrieši danú požiadavku, alebo ju ďalej pošle svojmu nasledovníkovi. Takto sa každému naskytne možnosť vyriešiť žiadosť, navyše v hierarchickom poradí. Dobre sa týmto vzorom riešia situácie, keď komponenty neriešia rovnaké množstvo požiadaviek, tak s vhodným usporiadaním sa nezaťažujú irelevantné komponenty [6].

Výhody zreťazenia zodpovedností sa prejavia v situáciách, keď by viacero objektov mohlo spracovať zaslanú požiadavku a nie je prednostne jasné, že ktoré by mohli riešiť danú zodpovednosť. Pri získaní požiadavky objekt automaticky rozpozná, či sa jedná o požiadavku, ktorú vie spracovať alebo ju má poslať nasledovníkovi, respektíve zahodiť. Taktiež sa tento vzor využíva pri poslaní podnetu množine objektov, pričom jeden z nich má na starosti spracúvanie podnetov typu podnetu. Navyše táto množina objektov môže byť špecifikovaná aj dynamicky [6].

Užívateľ pošle žiadosť na spracovanie komponentu, ktorého pozná. Tento komponent príjme požiadavku a môže nastať viac situácii: buď bola žiadosť preňho určená a spracuje ju, alebo mu nebola určená a nespracuje ju. Potom bez ohľadu na to, či ju spracoval, ju môže poslať na skutočne prvé alebo opätovné spracovanie iným komponentom. Tento objekt, ktorému bola poslaná požiadavka, je prvým článkom v reťazi komponentov, tak týmto spôsobom môže požiadavka postupne precestovať cez celú reťaz, až na konci sa zahodí.

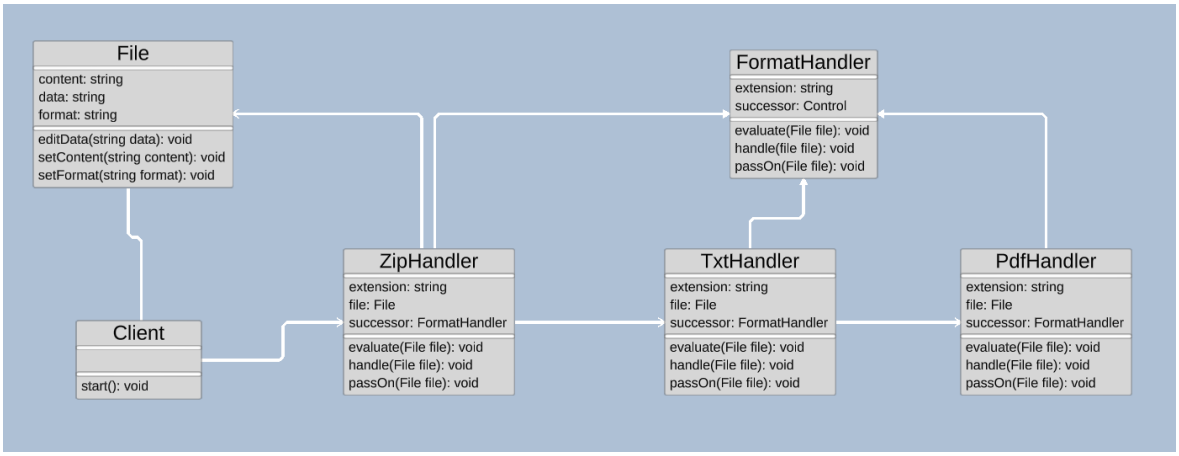
Následky využitia tohto návrhového vzoru sú, že sa znížia závislosti objektov. Pre vyriešenie požiadavky stačí znalosť, že sa nejak vyrieši a nezameriava sa na to, že ktorý objekt to má na starosti. Taktiež objekty v reťazi nepoznajú štruktúru, jediné, čo riešia sú priamy nasledovník. Taktiež sa pridáva flexibilita riešenia zodpovedností. Za behu sa môžu pridávať alebo odoberať komponenty do reťaze, čím sa upravuje ako budú objekty spracované [6]. Nevýhody takéhoto riešenia ležia vo voľnosti štruktúry reťaze. Tým, že nie sú stanovené konkrétne komponenty, tak pri nesprávnej kompozícii sa môže stať, že požiadavka nebude spracovaná ani jedným objektom. O tomto fakte sa nemožno z reťaze dozvedieť a mohlo by sa to stratiť.

3.2 Diagram tried Files

Na našom príklade sme chceli predviesť, že ako vyzerá riešenie požiadavky potom ako je poslaná klientom. Je jasne vidno ako postupne cestuje po reťazi a pri splnení určitého kritéria, je obslúžená objektom zodpovedným za ňu.

Príklad reprezentuje systém zodpovedný za spracovanie súborov s rôznymi príponami. Systém obsahuje samotné súbory v triede **File** a potom objekty zodpovedné za prácu so súbormi konkrétneho typu, ktoré tvoria *reťaz zodpovedností*. Tieto objekty

rozširujú rozhranie **FormatHandler** a pozostávajú zo **ZipHandler**, **TxtHandler** a **PdfHandler**. Všetky triedy sú na obrázku diagramu tried 3.1.



Obr. 3.1: Diagram tried v *AnimArchu* pre návrhový vzor *zreťazenie zodpovedností*

Trieda **File** reprezentuje súbor s nejakou príponou. Obsahuje atribúty *data*, *format* a *content*. Atribút *data* textovo reprezentuje obsah súboru, *format*, opisuje typ súboru. Pre nás je to najdôležitejší atribút, podľa ktorého sa komponenty budú rozhodovať, či majú obslúžiť daný súbor. Posledný je atribút *content*, ktorý bude podstatný v prípade, že sa jedná o súbor typu „zip“. Tento atribút opisuje príponu súboru po rozbalení, pre jednoduchosť a potrebu nášho príkladu nám postačí predpokladať, že obsah zabaleného súboru je jeden súbor. Zvolili sme osobitný atribút pre príponu súboru pre zjednodušenie práce s objektami typu **File**. Metódy v tejto triede slúžia iba na nastavovanie atribútov a volajú sa *setFormat(string format)*, *editData(string data)* a *setContent(string content)*.

Rozhranie **FormatHandler** predstavuje komponenty, ktoré spracujú súbor s príponou určenou v triede. Relevantná prípona sa uchováva v *extension* a v zmysle návrhového vzoru *zreťazenie zodpovedností*, nasledujúci komponent je uchovaný v premennej *successor*. Obsahuje hlavnú metódu *evaluate(File file)*, ktorá rozhodne, či sa má spracovať súbor *file*, alebo posunúť nasledovníkovi. Tento súbor by sa spracoval zavolaním metódy *handle(File file)* v rovnakej triede a posunul ďalej pomocou metódy *passOn(File file)*. Presné kritéria ako sa rozhoduje, že ktoré vykoná opíšeme v ďalšej podkapitole 3.3.

Keďže pre rôzne formáty sú iné akcie, ktoré sa vykonajú pre súbor, tak je niekoľko tried, ktoré rozvíjajú toto rozhranie a opisujú konkrétnu príponu. Reťaz začína triedou

ZipHandler, ktorá spracúva súbory s príponou „.zip“. Je prvá v reťazi, lebo poskytuje možnosť poslať ďalej rozbalený súbor na spracovanie ďalším komponentom. Nasleduje jeho nasledovník **TxtHandler**, zodpovedný za obyčajné textové dokumenty s príponou „.txt“. Nakoniec reťaz končí triedou **PdfHandler**, ktorá má na starosti dokumenty typu „.pdf“. Ak nastane situácia, že by súbor triedy **File** nebol ani jedným z vyšších formátov, tak sa v našom systéme neobslúži a zahodí sa.

Každá z tried obsahuje príponu *extension*, ktorú kontroluje a svojho nasledovníka v atribúte *successor*. Taktiež tri vyššie spomínané metódy *evaluate(File file)*, *handle(File file)* a *passOn(File file)*, ktoré implementujú podľa vlastných kritérií.

3.3 OAL kód a animácia spracovania súboru

V našom **OAL** príklade popisujeme základnú funkcionalitu tohto návrhového vzoru. Reprezentuje poslanie požiadavky a jej spracovanie správnym komponentom, na to zameraným. Požiadavka je v prvom príklade súbor s koncov „.txt“ a správny komponent je objekt typu **TxtHandler**. Vytvorenie objektov a poslanie súboru na spracovanie je vyjadrené v implementácii 3.1.

```
1 create object instance file_1 of File;
2 form=read("Please insert file format (zip/txt/pdf/other)");
3 file_1.setFormat(form);
4
5 create object instance ziphandler_1 of ZipHandler;
6 ziphandler_1.extension = "zip";
7 ziphandler_1.evaluate(file_1);
```

Kód 3.1: Implementácia metódy *start()* triedy **Client** v OAL kóde v *zreťazení zodpovedností*.

Príklad obsahuje nasledujúce kroky a animácia je na obrázkoch 3.2 a 3.3. Priebeh celej animácie je v prílohe, tu zobrazujeme najpodstatnejšie kroky.

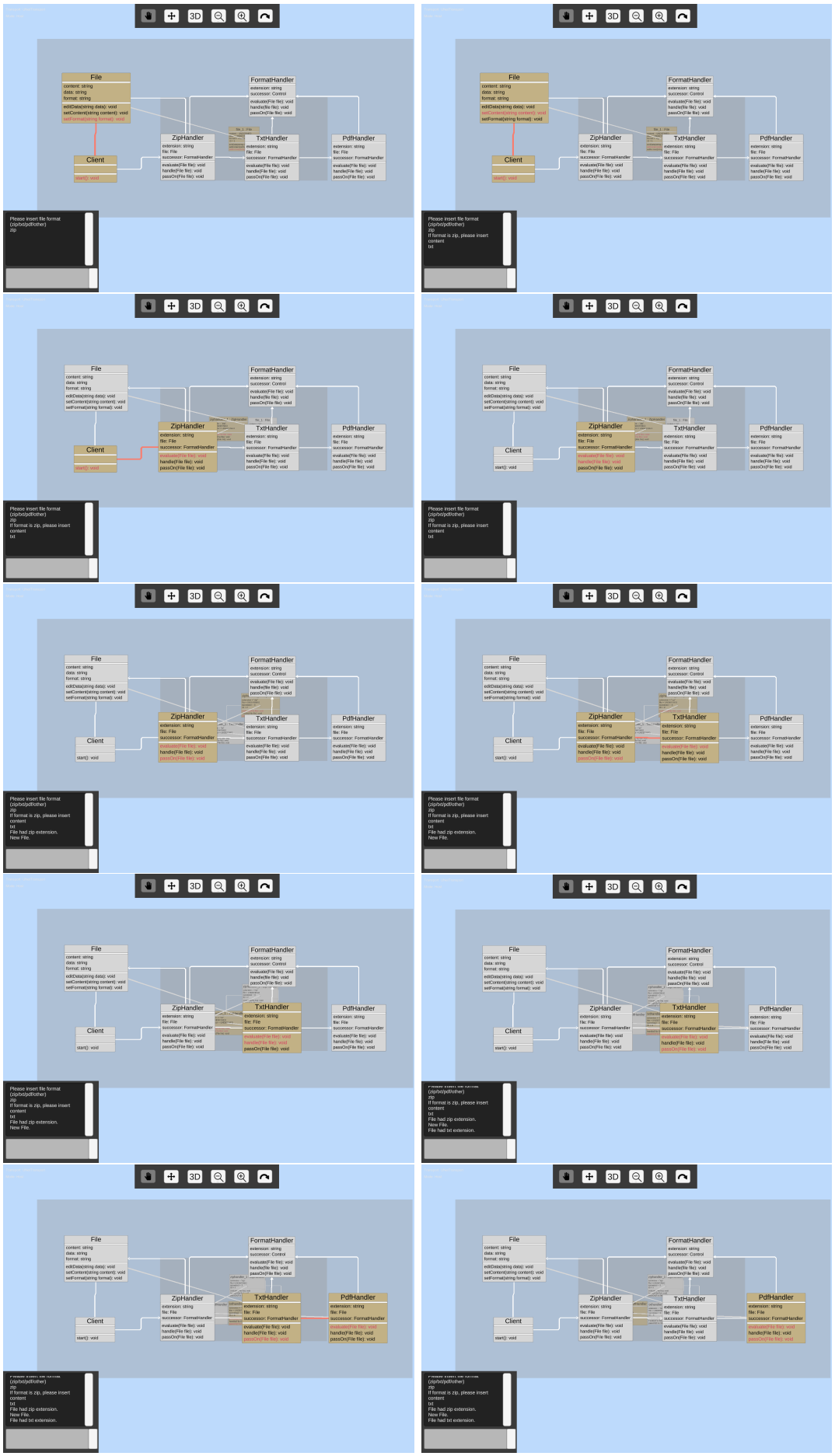
1. Vytvorenie súboru typu **File**
    - Získanie hodnoty formátu súboru zo vstupu

2. Vytvorenie inštancie triedy **ZipHandler**

3. Poslanie súboru na spracovanie po reťazi začínajúcou objektom triedy **ZipHandler**

    - Odmietnutie alebo spracovanie triedou **ZipHandler**
    - Odmietnutie alebo spracovanie triedou **TxtHandler**
    - Odmietnutie alebo spracovanie triedou **PdfHandler**

Obr. 3.2: Animácia v *AnimArchu* pre návrhový vzor zreťazenie zodpovedností



Obr. 3.3: Animácia v *AnimArchu* pre návrhový vzor zreťazenie zodpovedností

Najprv vytvoríme inštanciu *file_1* triedy **File** a potom jej pridelíme formát, ktorý zapíšeme do konzoly po reagovaní na výzvu, formát sa nastaví pomocou metódy *setFormat(string format)*.

Následne tento objekt pošleme na spracovanie inštancii *ziphandler_1* triedy **ZipHandler**, ktorá je prvá v reťazi. Zavoláme jej metódu *evaluate(File file)* s parametrom *file_1*. Keďže prípona súboru, „.txt“, nesedí s nutnou príponou pre to, aby *ziphandler_1* obslúžil daný súbor, čo je „.zip“, tak sa pošle nasledovníkovi, ktorý je triedy **TxtHandler**, pomocou metódy *passOn(File file)*, ako je to popísané v kóde 3.2. Táto metóda zavolá metódu *evaluate(File file)* inštancie *txthandler_1* triedy **TxtHandler**, ktorá rieši súbory s príponou „.txt“, teda náš súbor *file_1* obslúži. Vykoná tak zavolaním svojej vlastnej metódy *handle(File file)*. Pri tomto príklade sme zvolili ten spôsob obslúženia, že sa objekt „pohltí“ po vyriešení a nepošle sa ďalej po reťazi.

```
1 if (file.format == self.extension)
2         self.handle(file);
3 end if;
4 if (file.format != self.extension)
5         create object instance txthandler_1 of TxtHandler;
6         txthandler_1.extension = "txt";
7         self.successor = txthandler_1;
8         self.passOn(file);
9 end if;
```

Kód 3.2: Implementácia metódy *evaluate()* triedy **ZipHandler** v **OAL** kóde.

Animácia sa prejaví tak, že najprv ukáže ako sa z triedy **Client** a jej metódy *start()* volá metóda *setFormat(string format)* v triede **File**. Postupne sa rozsvieti metóda *start()*, vzťah medzi týmito dvoma triedami a nakoniec metóda *setFormat(string format)*.

Poslaním súboru na spracovanie sa rozsvieti vzťah medzi triedami **Client** a **ZipHandler**. V triedach, ktoré rozširujú rozhranie **FormatHandler** sa po rozvietení metódy *evaluate(File file)*, rozsvieti buď metóda *handle(File file)* alebo *passOn(File file)* v tej istej triede. Toto závisí od toho, či je súbor *file* vhodného formátu.

V druhom príklade ukážeme ako sa jednoduchou zmenou v pár riadkoch kódu metódy *evaluate(File file)* zmenil spôsob fungovania nášho *zreťazenia zodpovedností*. Namiesto toho, aby sa po spracovaní súbor pohltil, sa po zmene pošle po reťazi na ďalšie možné spracovania. Tento raz sa súbor spracuje v triede **ZipHandler** a po poslaní ďalej sa znovu obslúži, tento raz triedou **TxtHandler**. Metóda *start()* triedy **Client** je popísaná v kóde 3.3.

```
1  create object instance file_1 of File;
2  form=read("Please insert file format (zip/txt/pdf/other)");
3  file_1.setFormat(form);
4
5  cont=read("If format is zip, please insert content");
6  file_1.setContent(cont);
7
8  create object instance ziphandler_1 of ZipHandler;
9  ziphandler_1.extension = "zip";
10 ziphandler_1.evaluate(file_1);
```

Kód 3.3: Implementácia metódy *start()* triedy **Client** v **OAL** kóde v *zreťazení zodpovedností*.

Znovu sa bude posielať súbor, tento raz používateľ vyberá okrem formátu aj zabalený súbor, ak sa jedná o súbor typu ".zip". My sme zvolili súbor s formátom „zip“ a s *content* „txt“, nastaveným nami pomocou metódy *setContent(string content)*. V tomto prípade už pri prvom ohodnotení v metóde *evaluate(File file)* triedy **ZipHandler**, sa spracuje súbor pomocou metódy *handle(File file)*. Zmenou v tomto prípade je, že sa súbor nepohltí, ale pošle sa nasledovníkovi upravený súbor. Tento súbor vyzerá tak, že vo *format* má obsah z *content*, t.j. „txt“ a *content* má prázdny. Implementácia v jazyku **OAL** je v kóde 3.4. Toto reprezentuje rozbalenie zabaleného súboru a poslanie jeho obsahu na ďalšie spracovanie. Nasleduje trieda **TxtHandler**, ktorá podobne ako v minulom príklade rozpozná, že má obslúžiť tento súbor a zavolá sa jej metóda *handle(File file)*.

```
1 if (file.format == self.extension)
2     self.handle(file);
3 end if;
4 create object instance txthandler_1 of TxtHandler;
5 txthandler_1.extension = "txt";
6 self.successor = txthandler_1;
7 self.passOn(file);
```

Kód 3.4: Implementácia metódy *evaluate()* triedy **ZipHandler** v **OAL** kóde.

Ani tu sa nekončí spracovávanie súboru a aj napriek tomu, že sa zavolala metóda *handle(File file)*, tak aj tak sa zavolá metóda *passOn(File file)* a pošle sa na spracovanie nasledovníkovi, poslednej triede reťaze **PdfHandler**, čo sa vykoná metódou *evaluate(File file)*. Po zistení, že to nie je vhodný súbor na spracovanie, sa zavolá metóda *passOn(File file)*, ktorá vypíše *"NO OTHER VALID FILE EXTENSIONS."* na konzolu a ukončí sa program.

3.4 Vygenerovaný Python kód

Python kód sme vygenerovali rovnakou metódou ako v poslednej podkapitole 2.4 predchádzajúcej kapitoly 2. Teda napísali sme si jednoduchý program, ktorý využíva *OALToPythonVisitor* vytvorený nástrojom **ANTLR** na preklad z kódu v jazyku **OAL** do kódu v jazyku Python.

Použitím rovnakých úprav pre vyriešenie problému prázdneho tela metódy, teda pridaním príkazu na zapísanie príkazu *" pass"* namiesto prázdneho tela metódy a využitím príkazu *Client().start()* na spustenie programu sme vyriešili rovnaké problémy. Znovu sme získali funkčný kód v jazyku Python, ktorý je ekvivalentný našej implementácii v jazyku **OAL**.

Na rozdiel od implementácie *dekorátora*, pri príklade *zreťazenia zodpovedností* sa naskytol trochu iný problém s príliš jednoduchým konštruktorom.

Pri objektoch typu **File** to zväčša neprekážalo, lebo je praktické mať metódy na zmenu atribútov *format*, *data* a *content*. Keby bol príklad viac rozvinutý, tak mohlo by byť praktickejšie už pri vytvorení objektu nastaviť tieto atribúty, ale v našom príklade

toto nebolo nutné.

Pre inštancie tried, ktoré rozvíjali rozhranie **FormatHandler** sme po vytvorení objektu priradili hodnotu *extension* priamo, nie cez metódu. Síce sa týmto porušil princíp zapuzdrenia, ale hodnoty, ktoré sme nastavovali, by mali byť pre tieto triedy predvolené hodnoty pre atribúty *extension*. Taktiež by triedy mohli priamo vytvárať inštanciu nasledovníka v konštruktore, na obrázku 3.4 sme znázornili porovnanie takéhoto prístupu s vygenerovaným kódom. Toto je ďalšia nevýhoda tohto jednoduchého konštruktoru, ale dá sa to obísť.

```
class ZipHandler:
    instances = []

    def __init__(self):
        self.extension = None
        self.file = None
        self.successor = None
        ZipHandler.instances.append(self)
```

```
class ZipHandler:
    instances = []

    def __init__(self):
        self.extension = "zip"
        self.file = None
        self.successor = TxtHandler()
        ZipHandler.instances.append(self)
```

Obr. 3.4: *Vľavo:* vygenerovaný základný konštruktor
Vpravo: praktický viac rozvinutý konštruktor

Ďalšie obmedzenie, na ktoré sme natrafili je, že pri použití **OAL** príkazu *if () - end if;* nie je podpora pre *else*, alebo aspoň nám to pri implementovaní nefungovalo. Tak namiesto konštrukcie *if - else* sme použili dve *if* konštrukcie, jedna ktorá kontrolovala, či je pravdivá podmienka, druhá kontrolovala nepravdivosť podmienky. Na obrázku 3.5 je porovnanie vygenerovaného Python kódu so situáciou, že keby sa v konštruktore vytváral nasledovník a použila by sa konštrukcia *if - else*.

Síce sme natrafili na niektoré obmedzenia v Python kóde vygenerovanom z kódu

```
def evaluate(self, file):
    if (file.format==self.extension):
        self.handle(file)

    if (file.format!=self.extension):
        txthandler_1=TxtHandler()
        txthandler_1.extension = "txt"
        self.successor = txthandler_1
        self.passOn(file)
```

```
def evaluate(self, file):
    if (file.format==self.extension):
        self.handle(file)
    else:
        self.passOn(file)
```

Obr. 3.5: *Vľavo:* vygenerovaný *if* príkaz *Vpravo:* náš *if-else* príkaz

napísaného v jazyku **OAL**, jednalo sa najmä o pripomienky typu, dĺžka kódu, nastavovanie hodnoty na viac miestach, keď sa to dá na jednom a podobne. Ale tieto obmedzenia nepôsobili negatívne na samotnú funkčnosť programu.

Kapitola 4

Implementácia návrhového vzoru Most

Posledný vzor, ktorý implementujeme je *most*. Najprv objasníme teóriu o tomto návrhovom vzore, potom sa zameriame na diagram tried príkladu, následne opíšeme dynamiku príkladu a čo má reprezentovať. Nakoniec, po vygenerovaní Python kódu, ktorý reprezentuje daný príklad, tento kód opíšeme.

4.1 Návrhový vzor most

Posledný návrhový vzor, ktorý sme implementovali, je *most*. Rovnako ako *dekorátor*, tento vzor patrí medzi štrukturálne návrhové vzory a poskytuje riešenie pre zloženie tried alebo objektov. Tento vzor opisuje ako riešiť oddelenie abstrakcie od implementácie [6].

Často sa stáva, že používateľovi majú byť dostupné iba niektoré metódy a implementácia sa rieši osobitne od nich. Takto vzniknú dve rôzne množiny príkazov, jedna verejne známa, druhá súkromná, ktorá sa využíva pri implementácii verejne známych metód. Týmto spôsobom sa zaručí skrytie implementácie a používateľ sa nezaťažuje detailmi.

Tento návrhový vzor sa využíva pri oddelení abstrakcie a implementácie, čo poskytne možnosť zmeny implementácie za behu. Taktiež toto poskytuje možnosť osobitne rozvíjať implementáciu alebo abstrakciu dedením. Nie sú na sebe priamo ničím závislé, tak sa navzájom neovplyvnia. Toto sa prejaví aj tak, že implementácia je skrytá

od klienta a hocijaké zmeny v implementácii klienta nezasiahnu [6].

V abstrakcii sú ponúkané metódy, ktoré sú viditeľné klientom a ten ich môže voľne používať. Zatiaľ abstrakcia má referenciu na objekt, ktorý má na starosti implementáciu a na vykonanie svojich metód využíva metódy tohto objektu. Tieto metódy sa nemusia zhodovať a často je situácia taká, že implementácia ponúka primitívne operácie, kým abstrakcia ponúka operácie zodpovedajúce týmto, len na vyššej úrovni [6].

Z takéhoto zloženia systému vyplývajú výhody, najmä oddelenie rozhrania a implementácie. Navzájom na sebe nezávisia, čím sa vyskytne voľnosť ľubovoľne meniť implementáciu pre konkrétne rozhranie. Toto zostáva v pozadí a nezasiahne to používateľov. Taktiež poskytuje možnosť lepšie rozširovať hierarchie abstrakcie a implementácie osobitne. Pomocou tohto vzoru sa skryje implementácia od klienta. Nie je ani len známe kedy nastane zmena v implementácii, čím sa zaručí úplné oddelenie od nej [6].

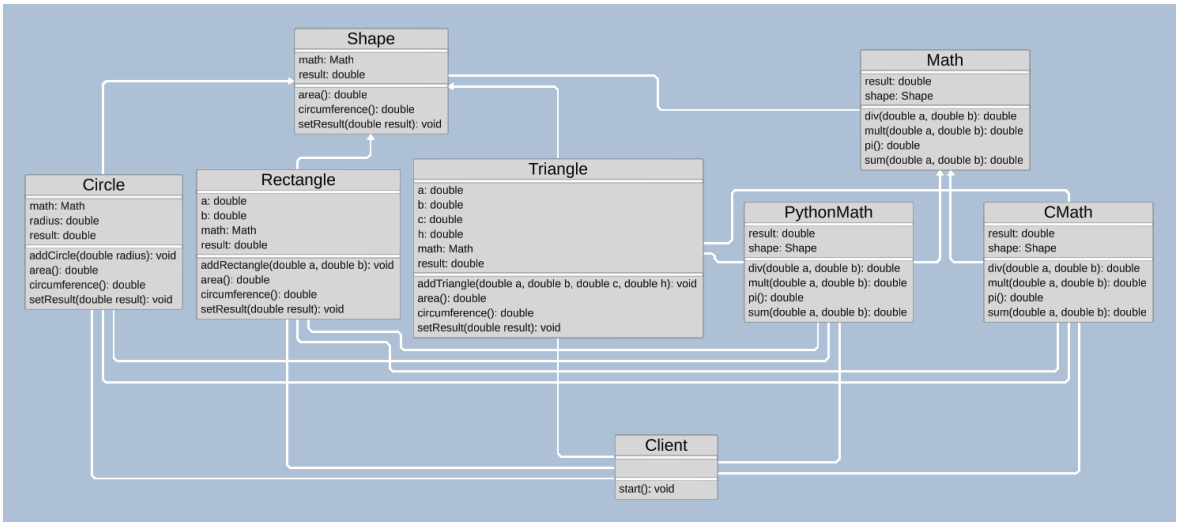
4.2 Diagram tried Shapes and Math

Návrhový vzor *most* pozostáva z dvoch častí. Prvá časť sú triedy, ktoré reprezentujú abstrakciu systému, teda príkazy, ktoré má k dispozícii používateľ alebo ovládací komponent. Druhá časť pozostáva z tried, ktoré majú na starosti samotnú implementáciu týchto príkazov. Samozrejme, tieto časti nemusia byť rovnaké, ani nejako priamoúmerné.

Chceli by sme dobre objasniť hlavne to, že klient využíva metódy ponúkané abstrakciou, tá následne využíva metódy implementácie. Preto je implementácia úplne oddelená od používateľa. Výsledný diagram príkladu možno vidieť na obrázku 4.1.

V našom príklade, abstrakciu reprezentuje rozhranie **Shape**, ktoré znázorňuje nejaký geometrický objekt. Ponúka metódy na výpočet obsahu a obvodu útvaru, konkrétne sa volajú *area()* a *circumference()*. Okrem metód, taktiež obsahuje atribút *math*, ktorý obsahuje inštanciu objektu triedy, ktorá rozširuje rozhranie *Math*, a atribút *result*, ktorý je pomocnou premennou pri výpočtoch.

Rozhranie abstrakcie rozširujú tri triedy, ktoré predstavujú kruh, obdĺžnik a trojuholník. Sú to **Circle**, **Rectangle** a **Triangle**. Každá implementuje obe metódy z rozhrania, aby pre daný tvar vypočítali správne hodnoty obsahu a obvodu. Okrem týchto dvoch metód obsahujú ešte tretiu, pomocou ktorej sa pridelia hodnoty atribú-



Obr. 4.1: Diagram tried v *AnimArchu* pre návrhový vzor most

tov jednotlivých tried. Atribúty sa pošlú v metóde ako parametre a ich hodnoty sa priradia do atribútov triedy.

Pre triedu **Circle** sa pridaná metóda volá *addCircle(double radius)*, pričom *radius* predstavuje polomer daného kruhu. Trieda **Rectangle** obsahuje metódu *addRectangle(double a, double b)*, pričom parametre a a b vystihujú dĺžky hrán obdĺžnika. Napokon trieda **Triangle** ponúka pridanú metódu *addTriangle(double a, double b, double c, double h)*. Parametre a, b a c predstavujú hrany trojuholníka a h vystihuje výšku, ktorú potrebujeme pri výpočte jeho obsahu.

Implementáciu predstavuje rozhranie **Math** s metódami pre výpočet základných matematických operácii, čo sú sčítavanie, *sum(double a, double b)*, násobenie, *mult(double a, double b)*, delenie, *div(double a, double b)*, a vrátenie hodnoty π, *pi()*. V binárnych metódach sa vždy operácia $\circ$, kde $\circ \in \{+, *, /\}$, vykonáva ako $a \circ b$.

Triedy, ktoré rozširujú rozhranie **Math** sú **PythonMath** a **CMath**, ktoré reprezentujú implementáciu týchto operácií v rôznych programovacích jazykoch. Sú to osobitné triedy, pretože v niektorých jazykoch sa iným spôsobom reprezentujú čísla, hlavne desatinné, a ich výpočty. Pre jednoduchosť nášho príkladu, my máme implementovanú len triedu **PythonMath**, ktorú využívame v príklade.

Vzťahy, ktoré sú medzi triedami pozostávajú zo zovšeobecnenia medzi rozhraniami a triedami, ktoré ich rozširujú a asociáciou medzi triedami, ktoré obsahujú inštanciu vytvoreného objektu v atribútoch. Tieto vzťahy asociácie nám slúžia pri animácii, kedy

sa prejavujú vzťahy a spolupráca medzi triedami.

4.3 OAL kód a animácia prípadov výpočtov vzorcov

V našom príklade opisujeme situáciu, v ktorej používateľ pomocou metód v abstrakcii chce vypočítať obvod kruhu a potom obsah trojuholníka. Používateľ interaguje jedine s metódami ponúkanými abstrakciou a tá potom deleguje výpočet implementácii. Nižšie sú vymenované kroky, ktoré sa udejú a pre ne potom podrobnejšie popíšeme ako ich znázorní animácia a ako vyzerá samotný **OAL** kód. Tento kód 4.1 máme rozpísaný pre triedu **Client**.

```
1 create object instance circle_1 of Circle;
2 circle_1.addCircle(2);
3 circle_1.circumference();
4
5 create object instance triangle_1 of Triangle;
6 triangle_1.addTriangle(6, 5, 5, 4);
7 triangle_1.area();
```

Kód 4.1: Implementácia metódy *start()* triedy **Client** v **OAL** kóde v *moste*.

Kroky, ktoré sa vykonávajú v príklade, pričom animácia je na obrázku 4.2.

1. Klient chce vypočítať obvod kruhu s polomerom $radius = 2$
   - Abstrakcia volá metódy v implementácii na vykonanie výpočtu vzorca $O = 2\pi r$
   - Postupne sa s rôznymi hodnotami volá metóda pre násobenie, a metóda na získanie hodnoty π
2. Výsledok výpočtu obvodu kruhu sa vypíše na konzolu
3. Klient zavolá výpočet obsahu trojuholníka s rozmermi $a = 6, b = 5, c = 5, h = 4$
   - Abstrakcia zavolá metódy aby sa vypočítal vzorec $A = \frac{a \cdot h}{2}$
   - Volajú sa metódy na násobenie a delenie

4. Výsledok výpočtu obsahu trojuholníka sa vypíše na konzolu

V prvom kroku sa najprv vytvorí inštancia *circle_1* triedy **Circle**. Toto sa prejaví v animácii tým, že sa rozsvieti trieda v diagrame. Ďalej s touto inštanciou pracujeme. Najprv sme pridali hodnotu polomeru pomocou metódy *addCircle(double radius)*, čo sa znázorní tým, že sa zvýrazní metóda *start()* v triede **Client**, potom sa rozsvieti vzťah medzi triedami **Client** a **Circle**, nakoniec sa rozsvieti aj volaná metóda *addCircle(double radius)*.

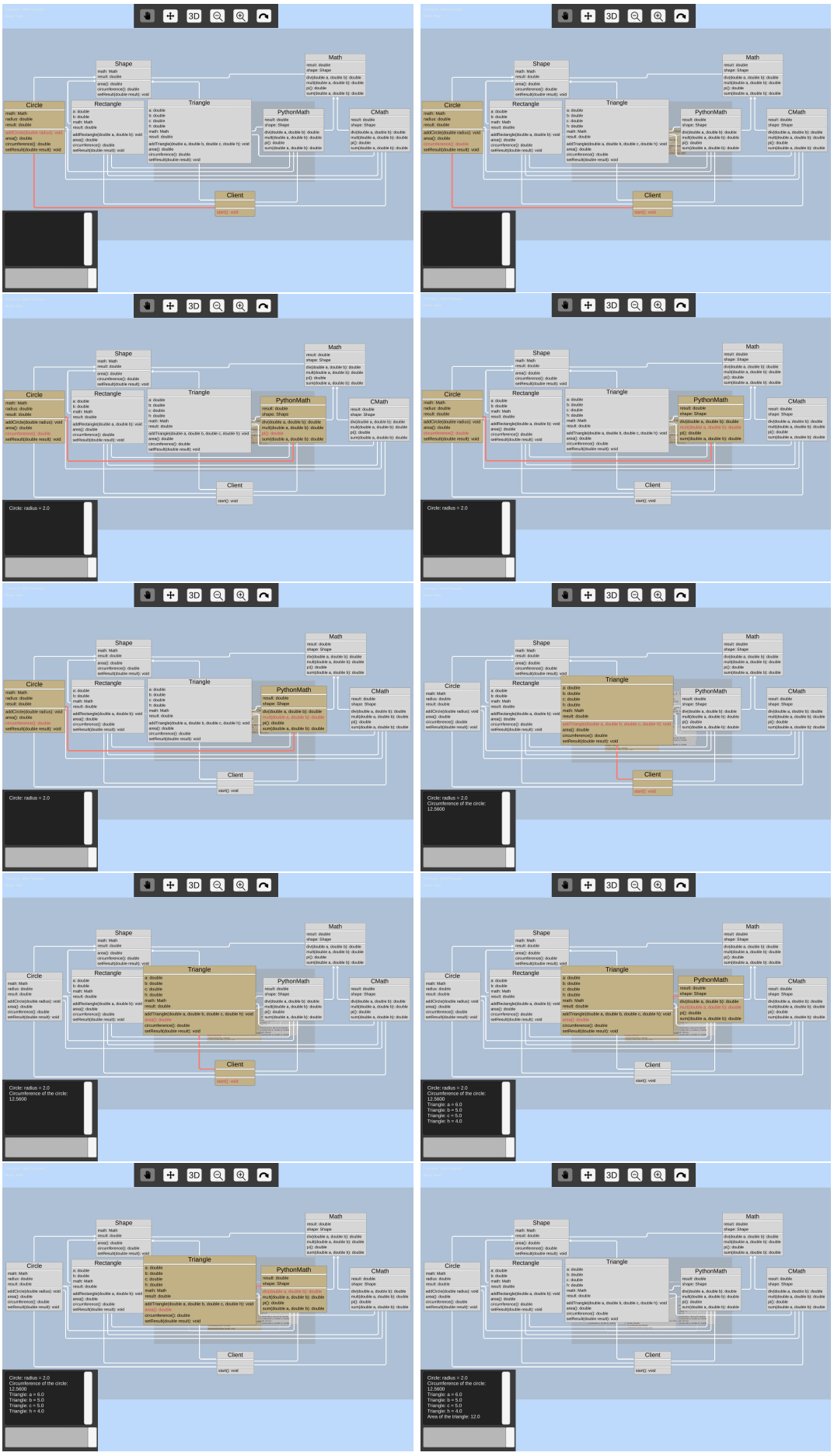
Metóda *addCircle(double radius)*, okrem priradenia hodnoty polomeru do triednej premennej na uloženie tejto hodnoty, taktiež vytvorí inštanciu *math_1* nejakej triedy, ktorá rozširuje rozhranie **Math**. V našom prípade sme zvolili **PythonMath**. Táto vytvorená inštancia sa uloží do premennej *math*, ktorá patrí triede **Circle**.

Pokračuje sa v práci s inštanciou *circle_1*, teraz pre kruh, ktorý reprezentuje chceme vypočítať obvod. To dosiahneme tak, že zavoláme metódu *circumference()*, ktorá správne vypočítanú hodnotu vypíše na konzolu programu. Implementáciu možno vidieť v OAL kóde 4.2.

Funguje tak, že najprv vypíše polomer kruhu, o ktorý sa jedná a následne postupne volá funkcie, ktoré poskytuje *math*, vo forme *self.math.func()*. V tomto kroku aj pomocou animácie vidno, že abstrakcia volá metódy v triede implementácie, teda sa vysvieti vzťah medzi triedami **Circle** a **PyhtonMath** a využívané metódy v triede **PythonMath**. V implementácii **PythonMath** je aritmetika jednoducho napísaná. Pomocou príkazov, ktoré poskytuje **OAL** na aritmetické výrazy sa vypočítajú hodnoty, ktorých výsledok sa zapíše do pomocnej premennej triedy rozširujúcu rozhranie **Shape**. Potom táto trieda využíva tieto výsledky na ďalšie počítanie alebo na ich vypísanie.

Práve pri získavaní výsledkov týchto výpočtov sa vyskytlo obmedzenie. V podmnožine **OAL**, ktorú my používame, sa nenachádza príkaz *return* a nedá sa pracovať s návratovými hodnotami metód. Preto sa hodnota získaná po výpočte aritmeticnej operácie musela nepriamo poslať inštancii objektu abstrakcie. Vyriešili sme to tak, že implementácia zavolala metódu *setResult(double result)* abstrakcie, čím sa nastavila jej pomocná premenná *result*, z ktorej už vie získať hodnotu. Priamejšie sa to nedalo, lebo nástroj *AnimArch* nepodporuje viacúrovňové priraďovania a získavania hodnôt, typu *result = self.math.result*.

Cieľom je vypočítať vzorec $O = 2\pi r$, teda musí z vnútra postupovať a postupne



Obr. 4.2: Animácia v *AnimArchu* pre návrhový vzor most, výpočet obvodu kruhu

volať funkcie na získanie hodnoty π, ktorého výsledok využije pre výpočet 2π. Tento výsledok sa použije pri výpočte už celého vzorca a konečný výsledok sa vypíše na konzolu. Postupný výpočet by sa mohol skrátene znázorniť ako $mult(mult(2, pi()), radius)$.

```
1 write("Circle: radius = ", self.radius);
2 self.math.pi();
3 self.math.mult(2, self.result);
4 self.math.mult(self.result, self.radius);
5 write("Circumference of the circle: ", self.result);
```

Kód 4.2: Implementácia metódy *circumference()* triedy **Circle** v **OAL** kóde.

Po vypočítaní hodnoty pre obvod kruhu, nasleduje počítanie obsahu trojuholníka. V tomto prípade sa jedná o vzorec $A = \frac{a \cdot h}{2}$.

Znovu sa najprv vytvorí inštancia *triangle_1* triedy **Triangle**, pričom sa inicializujú hodnoty strán a, b, c a výšky h na $a = 6$, $b = 5$, $c = 5$, a $h = 4$ pomocou metódy *addTriangle(double a, double b, double c, double h)*. Znovu sa vytvorí nová inštancia triedy **PythonMath**, ktorú používa len *triangle_1*. Potom sa zavolá výpočet obsahu pomocou metódy *area()*, ktorej algoritmus je popísaný v úseku kódu 4.3.

Metóda *area()* triedy **Triangle** taktiež najprv vypíše údaje o trojuholníku, v tomto prípade dĺžky strán a výšku. Potom vynásobí $a \cdot h$, ktorého výsledok sa použije pri vydelení s číslom 2. Teda znovu výpočet možno vystihnúť ako $div(mult(a, h), 2)$. Výsledok sa vypíše na konzolu.

Tak ako pri výpočte obvodu kruhu, tak aj pri výpočte obsahu trojuholníka abstrakcia rozvíjajúca **Shape** volá metódy v implementácii **Math**, pričom vďaka animácii jasne vidno ako spolupracujú.

```
1 write("Triangle: a = ", self.a);
2 write("Triangle: b = ", self.b);
3 write("Triangle: c = ", self.c);
4 write("Triangle: h = ", self.h);
5 self.math.mult(self.a, self.h);
6 self.math.div(self.result, 2);
7 write("Area of the triangle: ", self.result);
```

Kód 4.3: Implementácia metódy *area()* triedy **Triangle** v **OAL** kóde.

4.4 Vygenerovaný Python kód

Python kód pre príklad vzoru sme vygenerovali rovnakým spôsobom ako pre predošlé vzory. Vytvorili sme jednoduchý program v jazyku C#, kde sme tiež rovnako doplnili príkaz *" pass"*, čím sa udrží funkčnosť kódu a riadok kódu *Client().start()*, čím sa zaistí spustenie programu.

Pomocou tohto postupu sme dostali funkčný kód v jazyku Python so správnou syntaxou, ktorý priamoúmerne predstavuje príklad implementovaný pomocou diagramu tried a **OAL** kódu, porovnanie vidno na obrázku 4.3 nižšie. V tomto prípade, kedže jedinú interakciu s užívateľom predstavuje výpis výsledkov na konzolu, tak funkčnosť sa prejaví presne týmto. Mohli by sme ešte upraviť kód tak, že by si od užívateľa vypýtal konkrétne hodnoty pre parametre funkcií, ktoré inicializujú triedy útvarov, teda *addCircle(double radius)* a *addTriangle(double a, double b, double c, double h)*.

```
write("Triangle: a = ", self.a);
write("Triangle: b = ", self.b);
write("Triangle: c = ", self.c);
write("Triangle: h = ", self.h);
self.math.mult(self.a, self.h);
self.math.div(self.result, 2);
write("Area of the triangle: ", self.result);
```

```
def area(self):
    print("Triangle: a = ", self.a)
    print("Triangle: b = ", self.b)
    print("Triangle: c = ", self.c)
    print("Triangle: h = ", self.h)
    self.math.mult(self.a, self.h)
    self.math.div(self.result, 2)
    print("Area of the triangle: ", self.result)
```

Obr. 4.3: *Vľavo:* OAL kód metódy *area()* v triede **Triangle**
Vpravo: ekvivalentný Python kód metódy *area()* v triede **Triangle**

Výsledný Python kód je jednoducho čitateľný, pochopiteľný a upravovateľný. Je to priamočiary preklad **OAL** kódu, teda sú tam niektoré limitácie. Jednou z nich je to, že sa v konštruktore nemôžu vykonávať žiadne ďalšie akcie. Keďže ak sa v OAL kóde chce pracovať s inštanciou objektu, tak sa vytvorí pomocou príkazu *create object instance name_ of_ instance of NameOfClass;*, tento sa pomocou parseru preloží na konštruktor, v ktorom sa inicializujú všetky atribúty triedy na *None*. Preto treba dodatočnú funkciu, ktorá nahradí netriviálny konštruktor, v našom príklade to boli metódy názvov *addShape()*, kde *Shape* bol *Circle*, *Triangle*, *Rectangle* v triedach rovnakých názvov. V obrázku 4.4 sme ukázali kódy, že ako sa to prekladá teraz a ako by sa to možno žiadalo, teda zredukovaním počtu metód.

```python
class Circle:
    instances = []

    def __init__(self):
        self.radius = None
        self.math = None
        self.result = None
        Circle.instances.append(self)

    def addCircle(self, radius):
        self.radius = radius
        pythonMath_1=PythonMath()
        self.math = pythonMath_1
        pythonMath_1.shape = self
```

```python
class Circle:
    instances = []

    def __init__(self, radius):
        self.radius = radius
        pythonMath_1=PythonMath()
        self.math = pythonMath_1
        pythonMath_1.shape = self
        self.result = None
        Circle.instances.append(self)
```

Obr. 4.4: *Vľavo:* kód vygenerovaný prekladom z OAL do Pythonu
Vpravo: kompaktnejší, praktickejší kód napísaný nami

Ako sme spomínali pri implementácii, ďalší problém bol pri získaní výsledku výpočtu. Toto by sa v Pythone jednoducho dalo vyriešiť, čo je ďalšia odlišnosť. Znovu by sa dalo predísť nadbytočným metódam a pomocným premenným. Porovnanie vygenerovaného kódu a nami napísaných metód je na obrázku 4.5.

Okrem týchto limitácii, výsledný Python kód zachováva plnú funkčnosť vytvoreného príkladu. Len pomocou analýzy statického kódu sa nemusia pochopiť vzťahy a spolupráce tried, preto je praktické mať aj **OAL** kód, ktorý sa zobrazuje animáciou, ktorá pomôže práve toto pochopiť. Keďže sú to len iné reprezentácie toho istého, tak to poskytuje viacero pohľadov, pričom každý sa zameriava na niečo iné.

```
def circumference(self):
    print("Rectangle: a = ", self.a)
    print("Rectangle: b = ", self.b)
    self.math.mult(2, self.a)
    i = self.result
    self.math.mult(2, self.b)
    self.math.sum(i, self.result)
    print("Circumference of the rectangle: ", self.result)
```

```
def mult(self, a, b):
    self.result = a*b
    self.shape.setResult(self.result)

def pi(self):
    self.result = 3.14
    self.shape.setResult(self.result)

def sum(self, a, b):
    self.result = a+b
    self.shape.setResult(self.result)
```

```
def circumference(self):
    print("Rectangle: a = ", self.a)
    print("Rectangle: b = ", self.b)
    result = self.math.sum(self.math.mult(2, self.a), self.math.mult(2, self.b))
    print("Circumference of the rectangle: ", result)
```

```
def mult(self, a, b):
    return a*b

def pi(self):
    return 3.14

def sum(self, a, b):
    return a+b
```

Obr. 4.5: *Vľavo:* kód vygenerovaný prekladom z OAL do Pythonu
Vpravo: kompaktnejší, praktickejší kód napísaný nami

Záver

V tejto bakalárskej práci sme implementovali tri návrhové vzory, pričom sme vytvorili ich *UML* diagramy a animácie pomocou *OAL*. Taktiež sme overili funkcionalitu vygenerovaného zdrojového kódu, reprezentujúceho tieto animácie, v jazyku Python.

Najprv sme sa oboznámili s *vývojom riadeným modelom* a vysvetlili súvisiace pojmy a nástroje, ktoré sme v práci použili. Popísali sme syntax *OAL* a opísali postup, ktorým sa generuje kód napísaný v *OAL* do zdrojového kódu v jazyku Python.

Následne sme popísali jednotlivé návrhové vzory. Implementovali sme ich ako konkrétne scenáre prípadov použitia, ktoré sú využiteľné v nástrojoch, ktoré využívajú *xUML* a *OAL*. Najprv sme navrhli a vytvorili ich statický *UML diagram tried*, potom sme ich implementovali v *OAL* a popísali sme výslednú animáciu v nástroji *AnimArch*. Nakoniec sme vyhodnotili funkčnosť vygenerovaného kódu v jazyku Python.

Pri vytváraní implementácie vzorov v *OAL* sme zistili niektoré chyby pri spracovaní *OAL* kódu v nástroji *AnimArch*, ktoré pomohli jeho vývoju.

Pri analýze vygenerovaného Python kódu sme overili, že sa dá automaticky vygenerovať kód v jazyku Python, ktorý je hneď spustiteľný a ekvivalentný kódu v jazyku *OAL*. Teda situácie, ktoré sa dajú reprezentovať v *OAL* sa dajú rovnocenne reprezentovať a spustiť v jazyku Python. Funkčnosť kódu bola zachovaná pri vytváraní inštancií objektov, volaniach metód, kontrole podmienok, načítaní a vypísaní vstupu a výstupu na konzolu a aj aritmetických operáciach. Jedine syntax kódu bola zdĺhavejšia, než keby ju napísal programátor ručne, čo sa preukázalo primitívnymi konštruktormi a zbytočnými viacnásobnými priradeniam hodnôt atribútov. Tieto ale odzrkadľujú syntax a prácu s jazykom *OAL*.

Vylepšenia implementácie vzorov by sa mohli vyskytnúť pri implementácii viacerých *OAL* príkazov. Výrazné obmedzenia predstavoval príkaz *return*, ktorý nie je implementovaný. Keby sa mohli využívať návratové hodnoty funkcií, tak by to poskytlo viac

možností a funkcionality.

Taktiež by sa mohla rozšíriť báza implementovaných vzorov a modelov. Ešte je obmedzená a nie sú implementované ani všetky základné návrhové vzory. Okrem nich, by sa mohli implementovať aj iné typy vzorov a modelov, čím by sa rozšírila možnosť overenia funkcionality nástrojov.

Literatúra

[1] BridgePoint. Object Action Language Reference Manual. Available at `http://www.ooatool.com/docs/OAL08.pdf` [Online, accessed 2022-10-31].

[2] Hakan Burden, Rogardt Heldal, and Toni Siljamaki. Executable and Translatable UML – How Difficult Can it Be? *18th Asia Pacific Software Engineering Conference*, pages 114–121, 2011. doi: 10.1109/APSEC.2011.37.

[3] One Fact. Action Language (OAL) Tutorial. `https://xtuml.org/learn/action-language-tutorial/`. [Online, accessed 2022-12-12].

[4] Kirill Fakhroutdinov. The Unified Modeling Language. `https://www.uml-diagrams.org/`. [Online, accessed 2023-02-26].

[5] Matej Ferenc, Ivan Polasek, and Juraj Vincur. Collaborative Modeling and Visualization of Software Systems Using Multidimensional UML. *IEEE Working Conference on Software Visualization (VISSOFT)*, pages 99–103, 2017. doi: 10.1109/VISSOFT.2017.19.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley, 2009. ISBN: 0-201-63361-2.

[7] Alexander S. Gillis. model-driven development (MDD). `https://www.techtarget.com/searchsoftwarequality/definition/model-driven-development`. [Online, accessed 2022-12-12].

[8] Lukáš Gregorovič, Ivan Polasek, and Branislav Sobota. Software Model Creation with Multidimensional UML. *Information and Communication Technology - EurAsia Conference*, 9357:343–352, 2015. doi: 10.1007/978-3-319-24315-$3_3$5.

[9] Stephen J. Mellor, Marc J. Balcer, and Ivar Jacobson. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, 2002. ISBN 0201748045, 9780201748048.

[10] Filip Novák. *Vizualizácia softvérových architektúr a generovanie zdrojového kódu*. Diplomová práca. FMFI UK, 2022.

[11] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, LLC., 2012. ISBN-13: 978-1-93435-699-9.

[12] Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. Model-Driven Development: Piecing Together the MDA Jigsaw Puzzle. *Informatik Spektrum*, 31:394–407, 2008. doi: 10.1007/s00287-008-0275-8.

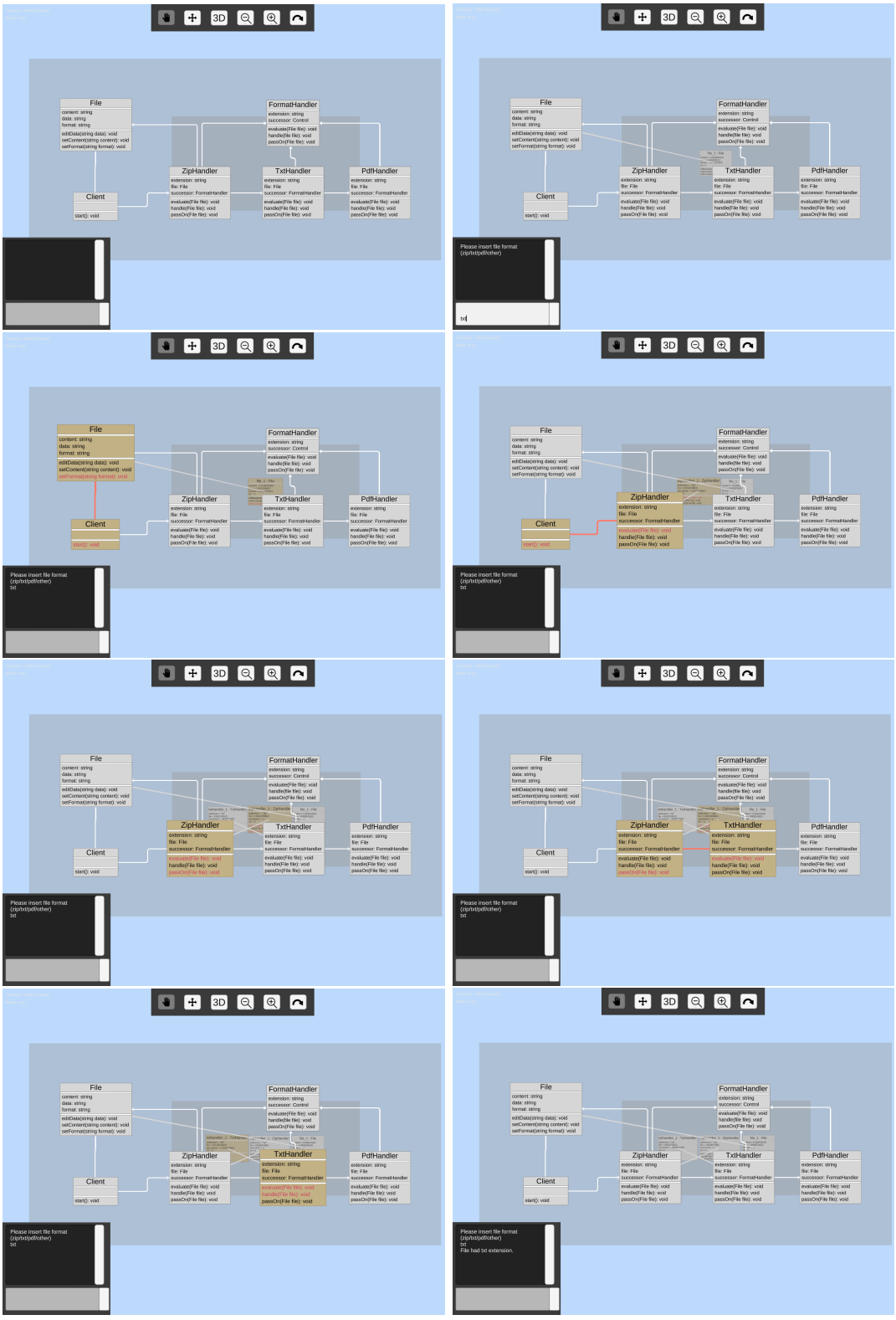
[13] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999. ISBN 0-201-30998-X.

[14] Leon Starr. *Executable UML:How to Build Class Models*. Prentice-Hall, 2002. ISBN: 0-13-067479-6.

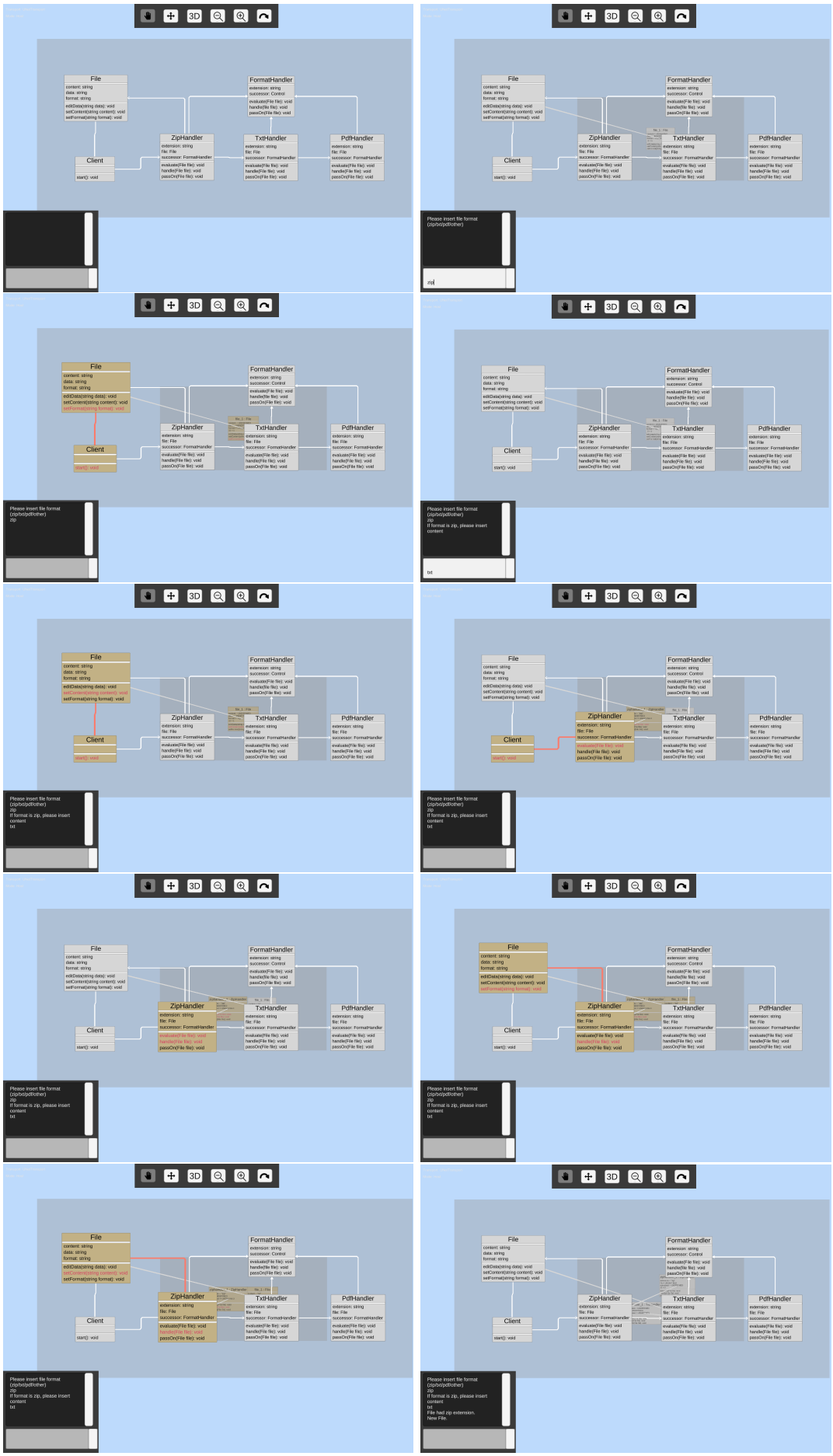
[15] Sparx Systems. Enterprise Architect. `https://sparxsystems.com/`. [Online, accessed 2023-05-20].

Príloha

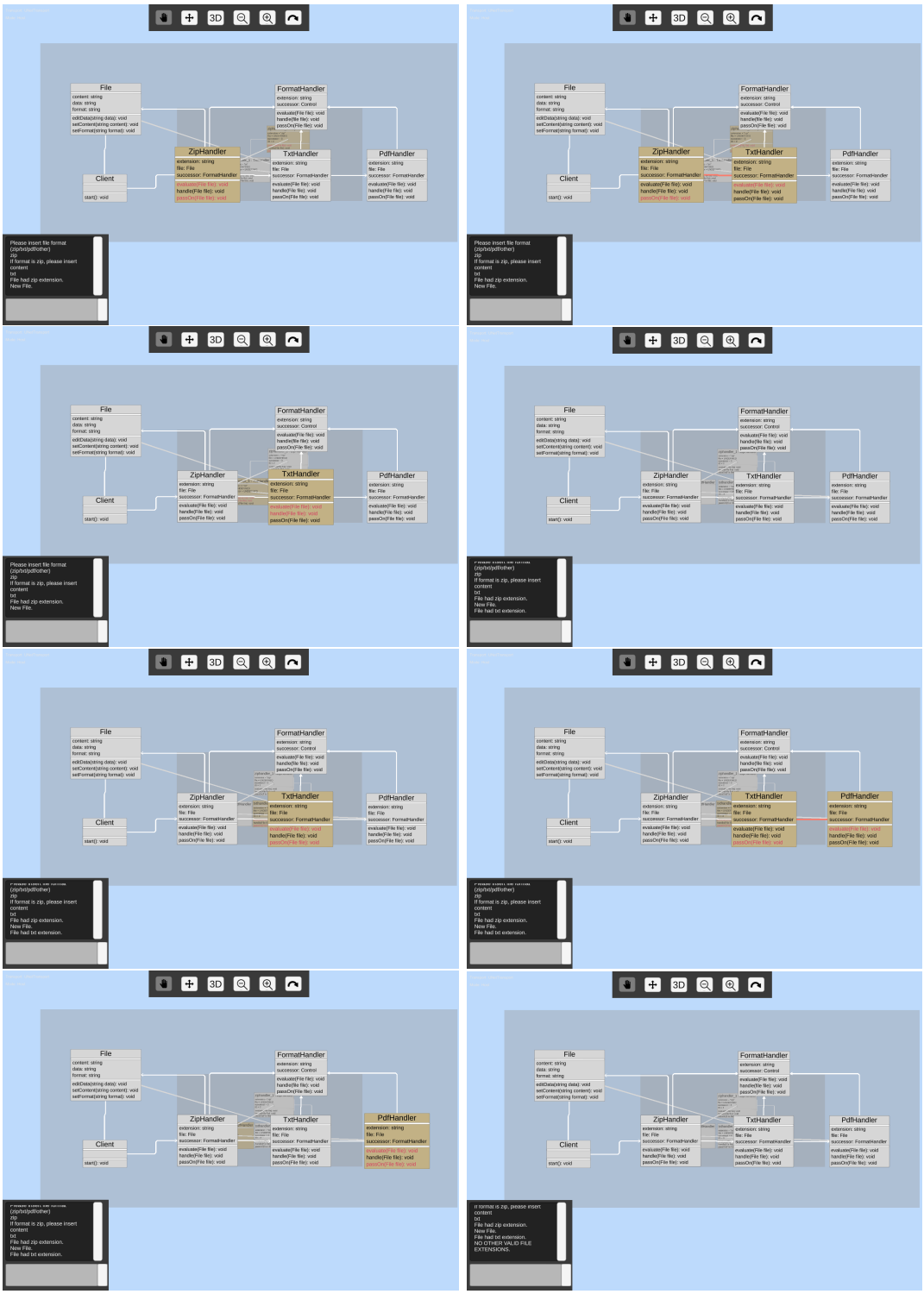
V tejto prílohe uvádzame obrázky animácií.



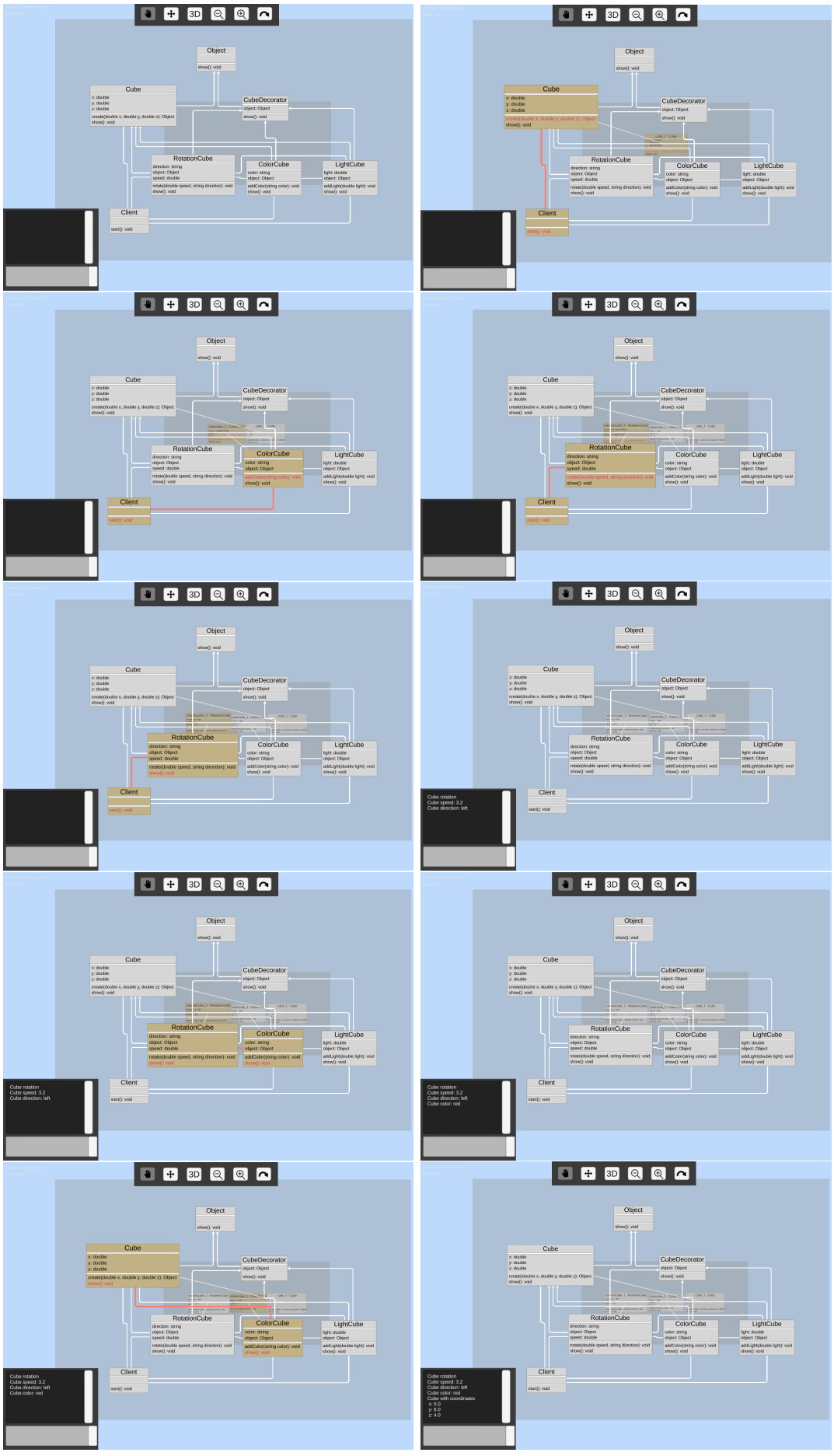
Obr. 4.6: Animácia v *AnimArchu* pre návrhový vzor zreťazenie zodpovedností



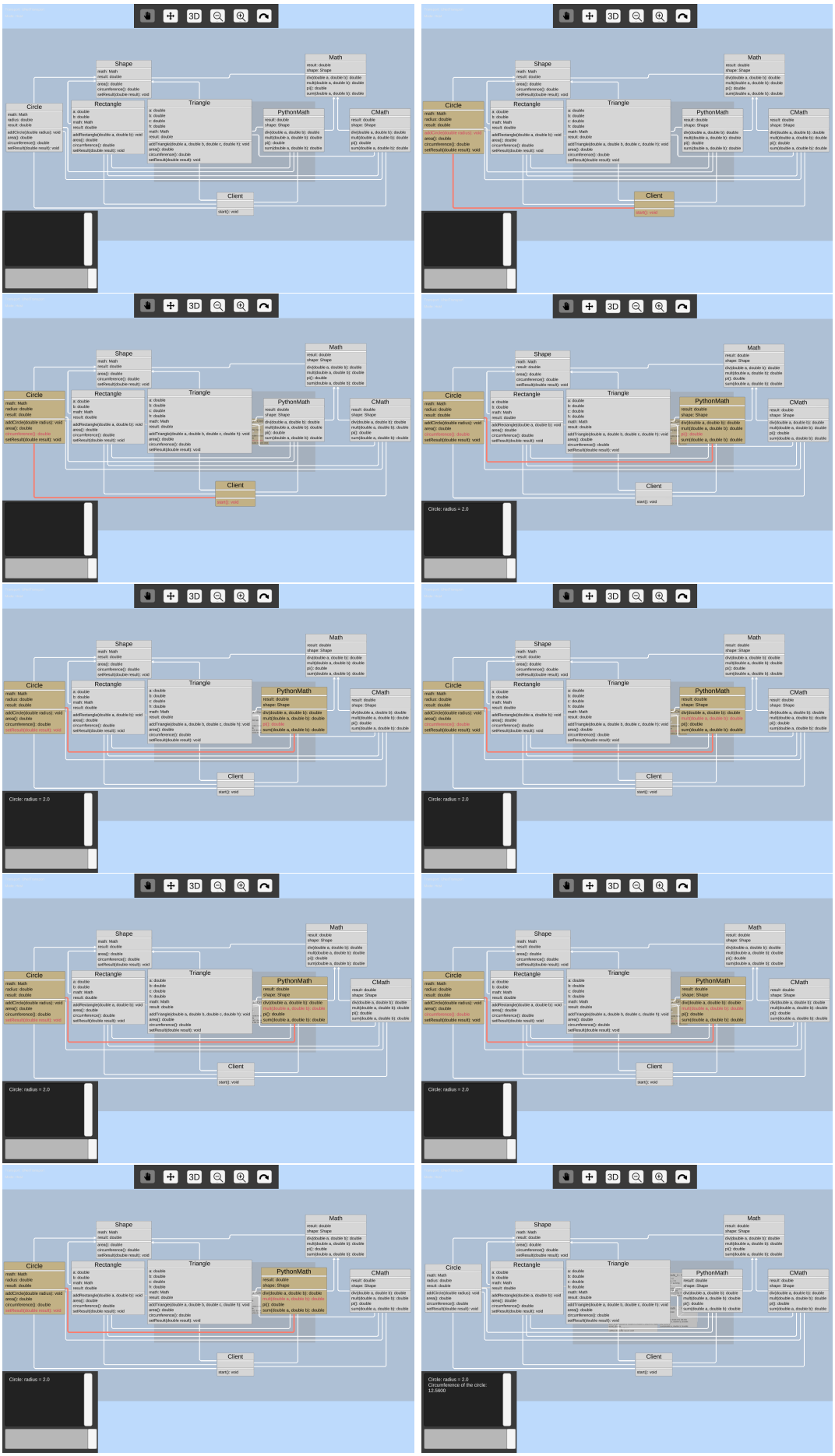
Obr. 4.7: Animácia v *AnimArchu* pre návrhový vzor zreťazenie zodpovedností



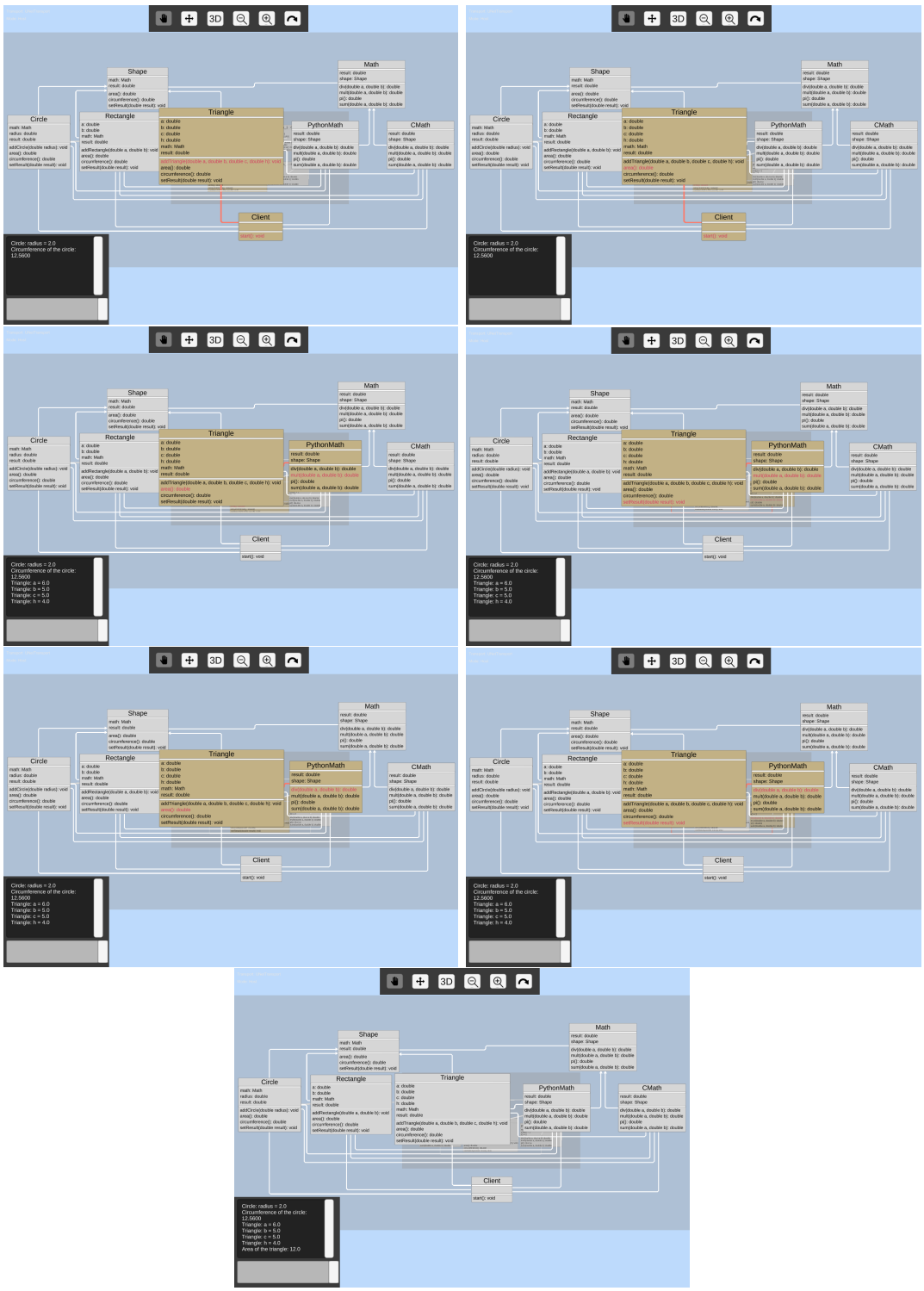
Obr. 4.8: Animácia v *AnimArchu* pre návrhový vzor zreťazenie zodpovedností



Obr. 4.9: Animácia v *AnimArchu* pre návrhový vzor dekorátor



Obr. 4.10: Animácia v *AnimArchu* pre návrhový vzor most, výpočet obvodu kruhu



Obr. 4.11: Animácia v *AnimArchu* pre návrhový vzor most, výpočet obsahu trojholníka