

Nástroj pro generování kódů z třídních diagramů

Tool for generating codes from class diagrams

Bc. Jakub Konvička

Semestrální práce

Vedoucí práce: Ing. Svatopluk Štolfa, Ph.D.

Ostrava, 2023

Zadání semestrálního projektu

Název: Nástroj pro generování kódů z třídních diagramů

Rok zadání: 2022/2023

Vedoucí: Ing. Svatopluk Štolfa, Ph.D.

Student: Jakub Konvička

Zaměření: Softwarové inženýrství

Forma studia: prezenční

Text zadání:

Cílem projektu je vytvoření nástroje pro generování kódu z třídních diagramů. Vytvoření rozhraní pro načtení diagramu a nastavení pravidel. Nástroj bude umožňovat generování do různých jazyků např. JAVA, C#. Bude možno přidat nový jazyk definováním transformačních pravidel. Taktéž pro jeden jazyk možnost definování více možností nebo na např. konkrétní vazbě nebo části modelu. Očekávaným výstupem projektu je:

1. Funkční zdrojový kód v jazyce Java, C#, ...
2. Textová část (10-20 stran)
3. Prezentace na cca 10 minut
4. Data pro experimenty

Obsah textové části

1. Seznámení s problematikou
2. State of the art
3. Podrobný popis vybrané části a její začlenění do nástroje.
4. Experimenty, vyhodnocení (možno použít tabulky a grafy)
5. Závěr - zhodnocení výsledků

Práce bude mimo hlavní část zahrnovat také seznámení se a aktivní použití např.

1. Platformy Java s prostředím Eclipse nebo C# apod.
2. Správa pomocí verzovacího systému např. GIT.
3. Testování - dle V modelu.
4. Měření kvality kódu - např. pomocí Sonar Qube, ...
5. Propojení příspěvků do verzovacího systému („commitů“) s úkoly.
6. Vykazování odpracovaného času na úkolech.

Literatura:

Podle pokynů vedoucího semestrálního projektu.

Abstrakt

Cílem tohoto projektu je návrh a následná implementace nástroje pro generování objektového kódu z třídních diagramů, vytvoření rozhraní a nastavení transformačních pravidel pro převod diagramu do několika jazyků. Nástroj umožní uživateli transformační pravidla přidávat pro potřeby generování do libovolného jazyka.

Klíčová slova

Generování kódu, Třídní diagram, Transformační pravidlo, OOP

Abstract

The purpose of this project is to design and then implement a tool for generating object code from class diagrams, creating an interface and setting up transformation rules for converting the diagram into several languages. The tool will allow the user to add transformation rules to any language for generation purposes.

Keywords

Code generation, Class diagram, Transformation rule, OOP

Poděkování

Rád bych na tomto místě poděkoval vedoucímu semestrální práce Ing. Svatopluku Štolfovi, Ph.D. za rady a čas, který mi věnoval při řešení dané problematiky.

Obsah

Seznam použitých symbolů a zkratek	6
Seznam obrázků	7
Seznam tabulek	8
1 Úvod	9
2 State of the Art v oblasti generování kódu	10
2.1 Generování kódu pomocí AI	11
2.2 IBM Rhapsody	14
2.3 Enterprise Architect	15
3 Vývoj vlastního řešení generátoru kódu	16
3.1 Řízení projektu	16
3.2 Návrh	17
3.3 Implementace serverové části	18
3.4 Implementace klientské části	23
3.5 Nasazení	26
4 Experimenty	27
4.1 Specifikace jazyka a transformačních pravidel	27
4.2 Proces testování	27
4.3 Měření kvality kódu	28
4.4 Výsledky	30
5 Závěr	31
Literatura	32
Přílohy	32

Seznam použitých zkratek a symbolů

OOP	– Objektově orientované programování
UML	– Unified Modeling Language
AI	– Artificial intelligence (Umělá inteligence)
CNN	– Konvoluční neuronová síť
RNN	– Rekurentní neuronová síť
GUI	– Graphic User Interface
SysML	– Systems Modeling Language
REST	– Representational State Transfer

Seznam obrázků

2.1	Princip fungování technologie AlphaCode AI [2]	11
2.2	Vývoj modelu technologie AlphaCode AI [2]	12
2.3	Princip algoritmu pix2code	13
2.4	Snímek obrazovky programu IBM Rhapsody [4]	14
2.5	Snímek obrazovky programu Enterprise Architect [5]	15
3.1	Životní cyklus úlohy	16
3.2	Jira Kanban board	17
3.3	Architektura REST [6]	18
3.4	Open API - server	22
3.5	Sekvenční diagram vyobrazující postup generování kódu z třídních diagramů	24
3.6	Uživatelské rozhraní generátoru kódu z třídního diagramu	25
4.1	SonarQube - počet chyb v kódu v čase	28
4.2	SonarQube - pokrytí kódu testy v čase	28
4.3	SonarQube - přehled výsledků měření	29

Seznam tabulek

3.1 Definice klíčových slov pro transformační pravidla	19
3.2 Definice struktury třídy	19

Kapitola 1

Úvod

Generování kódu z třídních diagramů je automatizovaný proces, na jehož vstupu je popis třídního diagramu. Tento vstup může být definován několika způsoby, nejčastěji se jedná o třídní diagram specifikovaný ve standardu UML, který znázorňuje jednotlivé třídy a vazby mezi nimi. Vizuální podoba třídního diagramu se pak převádí do formátu (JSON/XML), který je určen pro výměnu dat. Takto specifikovaný diagram se pak předává na vstup generátoru kódů, který mapuje specifikaci diagramu na objektový kód, který je výstupem celého procesu.

Jak již bylo naznačeno v předchozím odstavci, generování kódu z třídního diagramu je proces skládající se z několika kroků. Prvním krokem je vytvoření třídního diagramu, z hlediska uživatelské přívětivosti je vhodné tento diagram vytvářet v grafické podobě dle standardu UML, pomocí příslušného nástroje. Dalším krokem je zmiňovaná konverze do textové podoby ve formátu XML nebo json. Poté je třeba definovat, jakým způsobem se diagram namapuje na zdrojový kód, což se obvykle provádí pomocí transformačních pravidel, které využívá samotný generátor kódů.

Proces automatizovaného generování kódu z třídního diagramu může programátorovi ušetřit mnoho času a snížit možnost výskytu chyb, které by mohly být způsobeny ručním přepisováním třídního diagramu do kódů.

Kapitola 2

State of the Art v oblasti generování kódu

Tato kapitola popisuje současný stavu znalostí, technologií, metod a postupů v oblasti obecného generování kódu a v závěru se zaměřuje na specifickou oblast generování kódů z třídních diagramů, která je náplní této práce.

Generování kódu přebírá úkol psaní opakujícího se kódu, takže mají vývojáři mají více času k soustředění se na zajímavější programátorské výzvy, které je baví. Samotná myšlenka principu generování kódů vychází z tvrzení: „čas programátorů je drahý“ a „programátoři nemají rádi repetitivní úlohy“. [1] Generování kódu neznamená jen odstranění náročné práce, ale přináší samotnému softwaru výhody, zejména pak v oblastech produktivity, kvality, konzistence a abstrakce. V následujících odstavcích budou tyto oblasti podrobněji popsány z pohledu přínosu automatizovaného generování kódu.

Z hlediska produktivity je generování kódu jednoznačným přínosem. Vyvíjené aplikace mohou být již v prvotní fázi vývoje složené z vyšších desítek až stovek tříd. Generátory kódu dokáží vytvořit stovky tříd během několika sekund. Pokud je třeba model následovně upravit či jej rozšířit, tak je možné kódy poměrně snadno přegenerovat a distribuovat je do aplikace.

Generátory kódu produkují kód s konzistentním názvoslovím i strukturou, tato konzistence umožňuje snazší budoucí úpravy kódu a pro celý tým se struktura a jmenné konvence stávají pravidlem. S tím je i spojená kvalita kódu, která je totožná napříč všemi moduly nebo třídami.

Propracovanější generátory kódu podporují vysokou míru abstrakce, a to především tím, že na jejich vstupu přijímají abstraktní model. Který může být znovupoužitelný pro několik cílových systémů. Výhodou takto generovaného kódu je vysoká míra přehlednosti a navíc tento kód umožňuje skrýt implementované části aplikační logiky za rozhraní, které jednotlivé třídy implementují. [1]

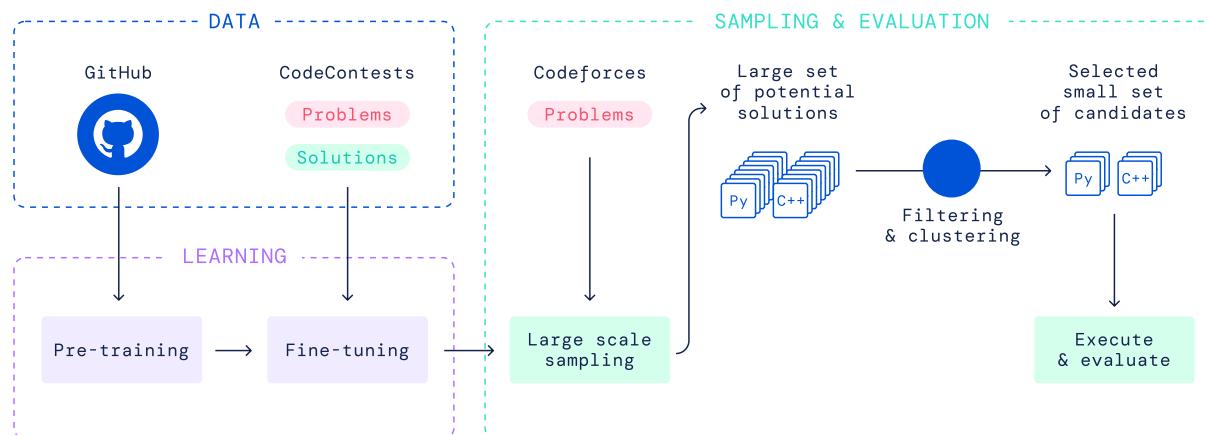
2.1 Generování kódu pomocí AI

V době psaní tohoto textu je na vzestupu užívání stále propracovanějších modelů strojového učení a s nadsázkou by se dalo říci, že se vývojáři snaží o implementaci AI do všech svých systémů. Tyto modely jsou neustále zdokonalovány a využívány jsou také v oblasti automatizovaného generování kódu. Modely se mohou trénovat na různorodých datech, jsou jim předkládána jednotlivá transformační pravidla společně s rozmanitými vstupy třídních diagramů v grafické či textové podobě.

2.1.1 Deepmind AlphaCode AI

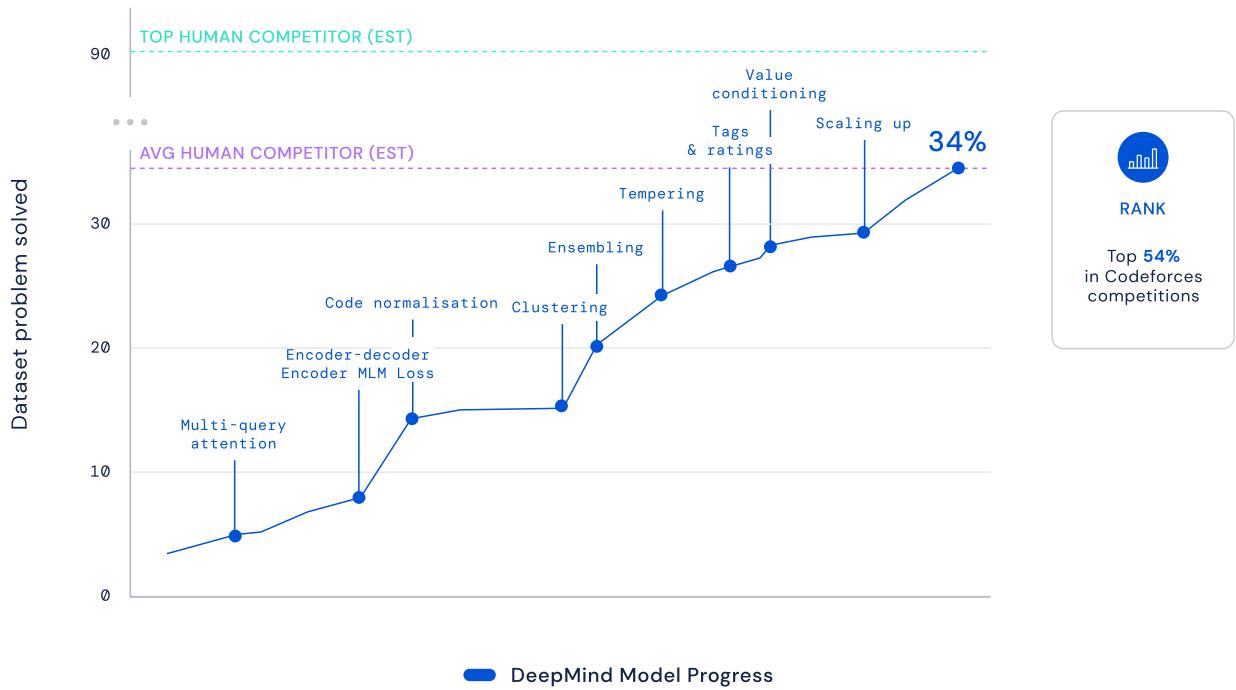
AlphaCode AI je technologie staví na AI a vyvíjí ji společnost Deepmind¹. Tato technologie je založena na současně nejmodernějších technikách strojového učení a umožňuje vytváření systémů, které jsou schopny rozpozнат, zpracovat a interpretovat lidskou řeč a psaný text. A to se v tomto případě využívá pro převod specifikace softwaru do spustitelného kódu.

V současnosti je tato technologie využívána v různých oblastech, jako jsou chatboti, automatizace procesů, analýza dat apod. AlphaCode AI je schopna analyzovat velké množství dat a provádět složité operace s poměrně vysokou přesností. Společnost Deepmind nabízí řešení na míru, která umožňují využívat výhod této technologie v souladu s konkrétními potřebami a cíli. Princip fungování této technologie je vyobrazena na obrázku 2.1. Vývoj modelu AlphaCode je dle společnosti Deepmind v počátku, již nyní se ale dle počtu vyřešených úloh rovná průměrnému lidskému řešiteli. Křivka vývoje je na obrázku 2.2.



Obrázek 2.1: Princip fungování technologie AlphaCode AI [2]

¹<https://www.deepmind.com/>



Obrázek 2.2: Vývoj modelu technologie AlphaCode AI [2]

2.1.2 OpenAI Codex

Tento nástroj byl vyvinut společností OpenAI², která se specializuje na výzkum umělé inteligence a strojového učení. Codex je výsledkem jejich práce na vytvoření pokročilého jazykového modelu, který dokáže interpretovat přirozený jazyk a generovat kód.

Codex je schopen generovat kód v mnoha různých programovacích jazycích, jako je Python, JavaScript, Ruby nebo Go. Nástroj je také schopen provádět další úlohy, jako je např. analýza dat, tvorba webových stránek nebo tvorba mobilních aplikací. Codex je také schopen rozpoznávat programovací paradigma a používat vhodný programovací styl pro daný jazyk.

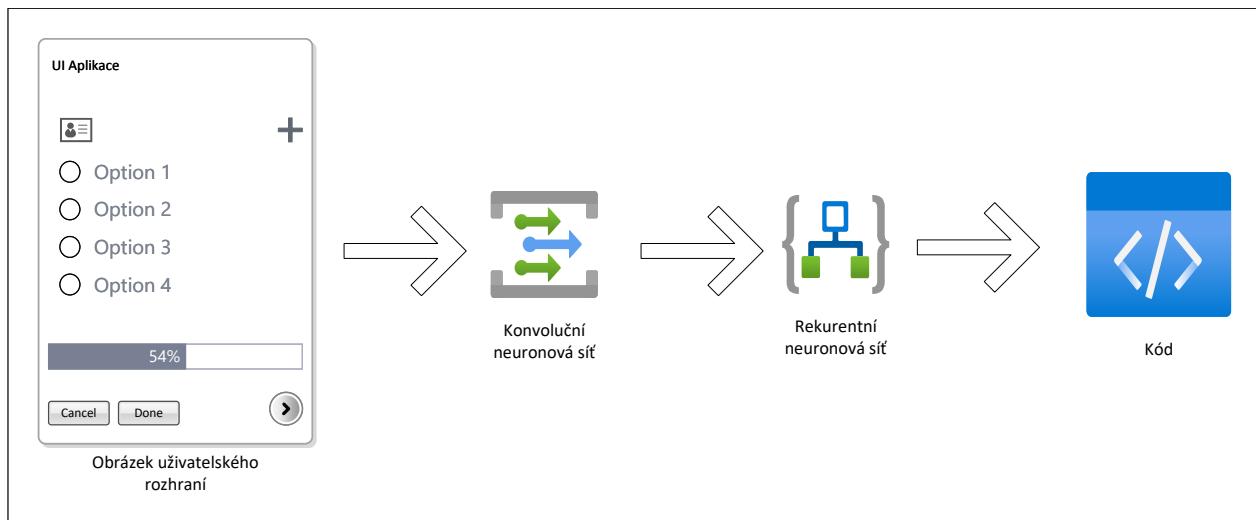
V současnosti je nástroj Codex stále v poměrně rané fázi vývoje, je náchylný k reprodukci chyb, které byly součástí datasetu, na kterém se neuronová síť učila.

²<https://openai.com/>

2.1.3 Pix2Code

Mezi poměrně specifickou oblast je automatizované generování kódu či struktury grafického rozhraní aplikace. Vstupem pro tyto generátory je grafická podoba uživatelského rozhraní vytvořená designérem aplikace. Výstupem by pak měl být kód, který definuje strukturu a styl uživatelského rozhraní. K takovému generování se používají konvoluční³ a rekurentní⁴ neuronové sítě.

V poměrně pokročilé fázi vývoje je projekt s názvem pix2code⁵, jehož algoritmus, slouží k převodu grafických uživatelských rozhraní (GUI) na kód. Jeho princip spočívá v trénování neuronové sítě, která bere vstup v podobě snímku obrazovky s GUI a generuje odpovídající kód v požadovaném jazyce. Vstupem pro učení konvoluční neuronové sítě (model) je datová sada obsahující dvojice snímků obrazovky a odpovídajícího kódu. Takto naučená síť se pak využívá k rozpoznávání jednotlivých prvků v uživatelském rozhraní, jako jsou tlačítka, textová pole apod. Poté se vytvoří rekurentní neuronová síť (RNN), která slouží k převodu rozpoznaných prvků na kód. RNN pracuje se sekvencí prvků rozpoznaných CNN a generuje odpovídající kód v programovacím jazyce. Popsaný princip je vyobrazen diagramem na obrázku 2.3 níže [3].



Obrázek 2.3: Princip algoritmu pix2code

³Konvoluční neuronové sítě jsou často používány pro zpracování obrazových dat, hlavní vlastností konvolučních neuronových sítí je schopnost zachytit hierarchii různých úrovní abstrakce v datech.

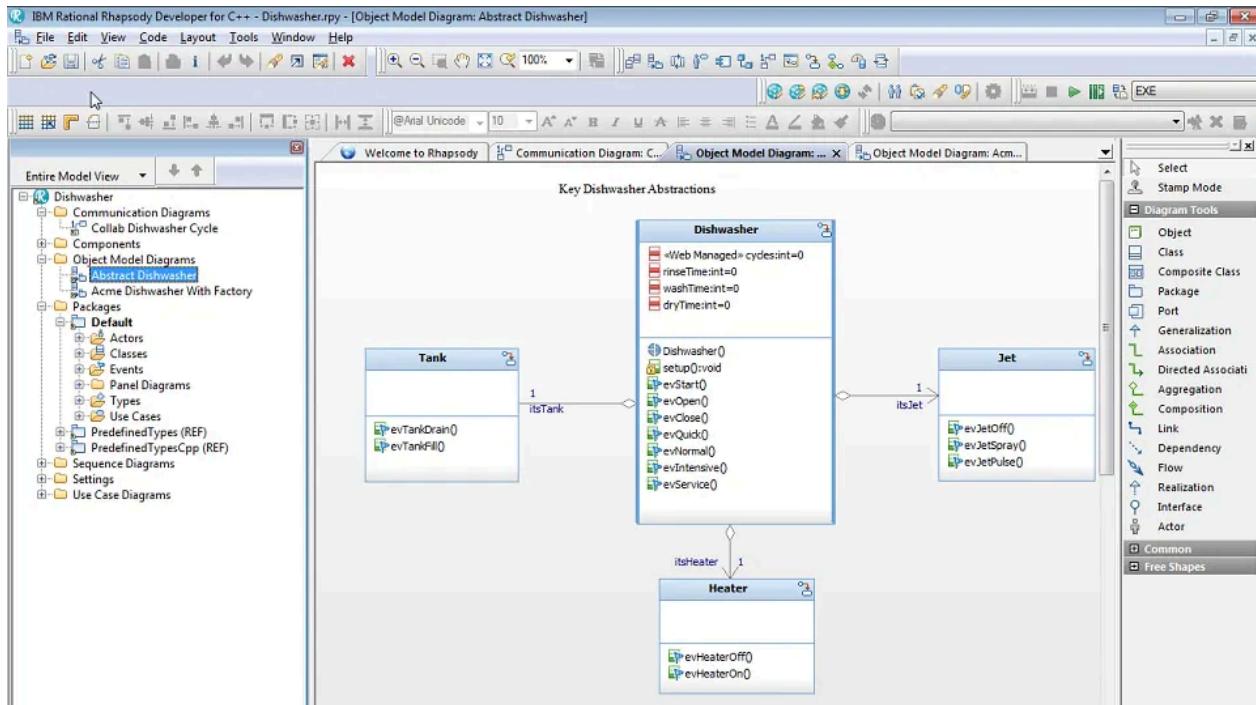
⁴Rekurentní sítě mají schopnost uchovávat vnitřní stav, který se mění s každým novým vstupem a ovlivňuje výstup sítě.

⁵<https://github.com/tonybeltramelli/pix2code>

2.2 IBM Rhapsody

Modely neuronových sítí jsou mocné, ale v kritických sekcích průmyslu si zatím nemůžeme dovolit AI využívát na maximum. V současnosti se hodí spíš jako doplněk stávajících řešení využívajících deterministické postupy a jsou řízeny pravidly, na které je stoprocentní spolehlivé. Právě pro tyto účely se na trhu nacházejí robustní řešení. Společnost IBM nabízí systém s názvem Rhapsody⁶. Jeho hlavní funkcí je umožnit návrh a vývoj komplexních softwarových systémů pomocí vizuálního modelování, které umožňuje lepší přehlednost, přesnost a efektivitu vývoje. Po modelování třídních (a dalších) diagramů v integrovaném prostředí Rhapsody je možné vygenerovat kód do jazyků C, C++, Java, Ada, MISRA-C, MISRA-C++. Model je vytvářen ve standardu SysML⁷ a UML.

Systém Rhapsody integruje podporu pro plnou vysledovatelnost požadavků (traceability links), propojuje požadavky s prvky návrhu a testovacími případy (UNIT testy), aby se zajistilo pokrytí a snadně se spravovaly případné požadavky na změny v systému. Součástí je také modul pro generování dokumentace vyvíjeného softwaru. Snímek obrazovky programu se nachází na obrázku 2.4.



Obrázek 2.4: Snímek obrazovky programu IBM Rhapsody [4]

⁶<https://www.ibm.com/products/architect-for-systems-engineers>

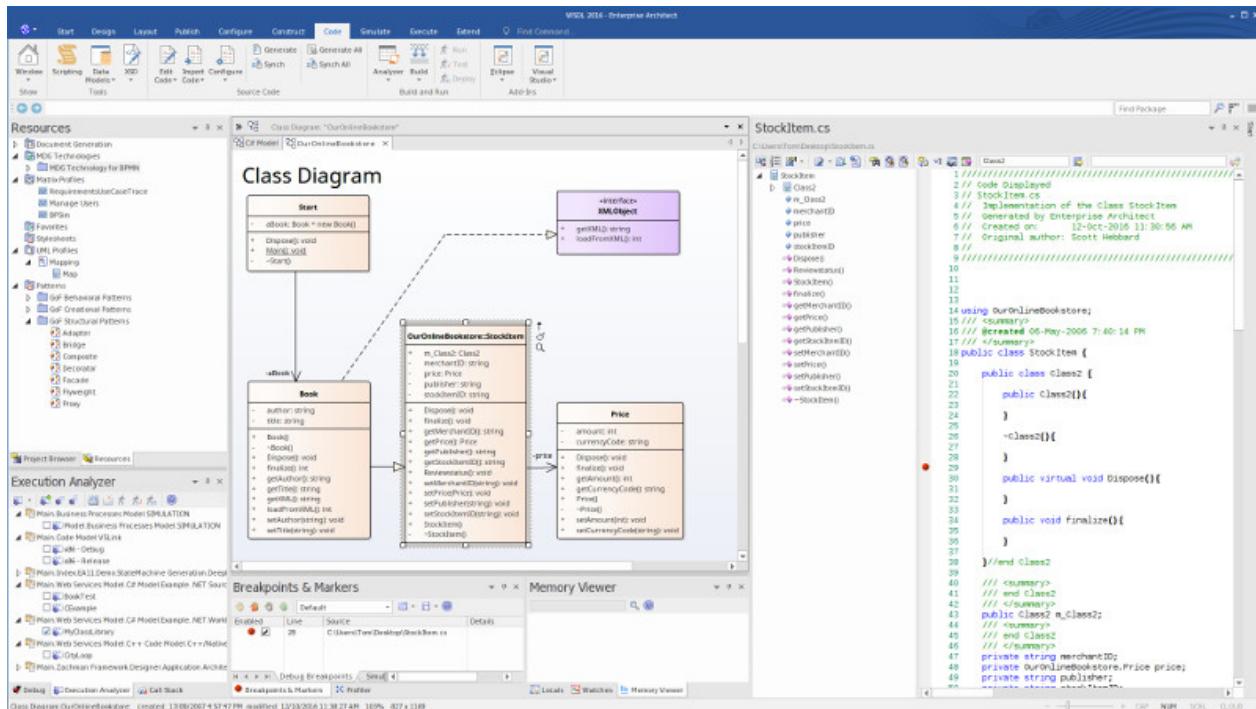
⁷<https://sysml.org/>

2.3 Enterprise Architect

Enterprise Architect, vyvíjený společností Sparx Systems⁸, je v průmyslu velmi využívaný nástroj pro modelování a analýzu softwaru. Dle slov autorů nástroje se jedná o celopodnikové řešení pro vizualizaci, analýzu, modelování, testování a údržbu systémů, softwaru, procesů a architektur. Výhodou Enterprise Architectu je jeho podpora pro generování kódu do různých programovacích či skriptovacích jazyků, jako jsou např. Java, JavaScript, C, C++, C#, PHP a Python. Plně je integrována podpora jazyka UML.

Nástroj Enterprise Architect je možné spouštět pod operačními systémy Windows, Linux a MacOS. Uživatelům je ale také nabízena možnost cloudového řešení se vzdáleným přístupem. Další výhodou tohoto software je jeho flexibilita a rychlosť načítání extrémně velkých modelů a možnost paralelní práce několika členů vývojového týmu v případě práce v cloudu [5].

Na obrázku 2.5 je snímek obrazovky programu Enterprise Architect s náhledem na třídní diagram a vygenerovaným kódem v jazyce C# v pravé části obrazovky.



Obrázek 2.5: Snímek obrazovky programu Enterprise Architect [5]

⁸<https://sparxsystems.com/>

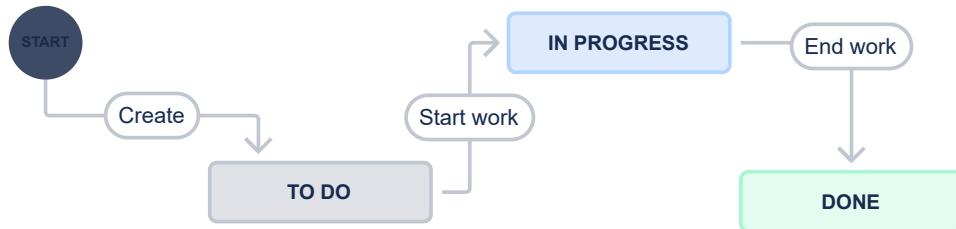
Kapitola 3

Vývoj vlastního řešení generátoru kódu

3.1 Řízení projektu

Pro účely konfiguračního managementu zdrojových kódů, které vznikly v rámci vývoje tohoto semestrálního projektu byl zvolen nástroj GIT. Jedná se o nástroj přímo určený ke správě verzí a sledování změn v kódu. Repozitář projektu je veřejně dostupný na url adrese <https://github.com/jkonvicka/CodeGenerator>, v rámci tohoto webového nástroje je možné zobrazit si jednotlivé vývojové větve a taktéž commity¹.

Správa jednotlivých úkolů při vývoji semestrálního projektu probíhala v prostředí systému Jira², jedná se o nástroj pro kompletní správu projektů a sledování chyb. Systém Jira umožňuje uživatelům vytvářet projekty, přidělovat úkoly, plánovat a řídit pracovní postupy, sledovat stav projektu a vytvářet dokumenty. Jednotlivé úkoly je možné propojit s verzovacím systémem GIT, nástroj taktéž umožňuje nastavit životní cyklus úloh tak, aby byl splněn specifický softwarový proces. Nastavený životní cyklus úloh pro tento projekt je vyobrazen pomocí diagramu na obrázku 3.1. A pomocí tzv. worklogů je možné sledovat čas, který byl vývojářem využit k jeho vyřešení.



Obrázek 3.1: Životní cyklus úlohy

¹Commit je operace ve verzovacím systému, která slouží k uložení změn provedených v souborech v rámci projektu

²<https://www.atlassian.com/software/jira>

Systém Jira umožňuje spravovat projekty pomocí několika metodik. Protože na projektu pracoval pouze jeden vývojář, byla zvolena poměrně jednoduchá a přehledná agilní metodika zvaná Kanban. Základem metodiky Kanban je vizualizace pracovního postupu na jednotlivých úlohách pomocí systému stavu těchto úloh na tzv. Kanban boardu. Kanban board se obvykle skládá ze sloupců, které zobrazují různé fáze procesu, např. „v procesu“ nebo „hotovo“. Podoba užitého Kanban boardu pro tento projekt je vyobrazena na obrázku 3.2.

The screenshot shows a Jira Kanban board with three columns:

- TO DO 2 ISSUES**:
 - Automatic inheritance property and method generation from interfaces and abstract classes (CSP-9)
 - Modify generetor to be able generate "implements Interface" and deriive classes (CSP-16)
- IN PROGRESS 1 ISSUE**:
 - Write "Řízení projektu" section at project documentation (CSP-22)
- DONE 15 ISSUES**:
 - [BUG] Fix wrong generated code (CSP-20)
 - Specify REST API (CSP-8)
 - Create simple unit test project and basic tests for exaple code generation (CSP-15)

A code snippet is also visible in the Done column:

```

import System
import System.Text

class UserClass : DatabaseObject:
    def __init__(self, Id : long, _workstation : long, Name : string = string.Empty):
        self.Id = Id
        self._workstation = _workstation
        self.Name = Name

    def getID() >> DATA_TYPE:
        return self.Id

    def setID(self, _Id : long) >> DATA_TYPE:
        self.Id = _Id
        return

    def getName() >> DATA_TYPE:
        return self.Name
  
```

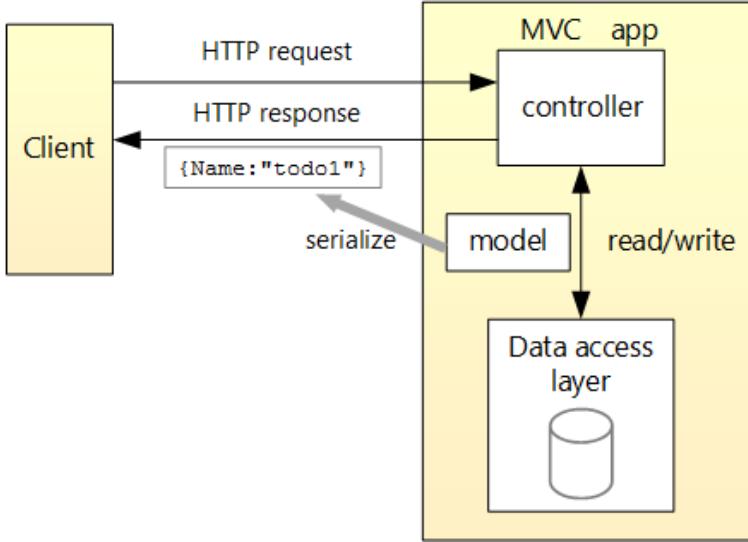
Obrázek 3.2: Jira Kanban board

3.2 Návrh

Pro dosažení lepší udržovatelnosti a efektivní správy projektu v průběhu vývoje byl zvolen architektonický styl Klient-Server. Tento styl umožňuje rozdělit softwarové řešení na dvě části: klienta (nebo klienty) a server. Klientská část představuje aplikaci spuštěnou na straně uživatele, která slouží k interakci s serverovou částí. Klient poskytuje uživatelské rozhraní a zpracovává uživatelské vstupy.

Server je pasivním prvkem systému, je řízen pořadavky klientů. Součástí serverové části je aplikační logika generátoru kódu z třídních diagramů. Pro účely tohoto projektu bylo rozhodnuto, že serverová část bude implementovat rozhraní odpovídající tzv. REST API. REST je dalším architektonickým stylem, který využívá standardních protokolů a HTTP metody jako GET, POST, PUT

a DELETE pro přenos dat a stavu mezi klientem a serverem. Model architektonického stylu REST je vyobrazen na obrázku 3.3. Přenášená data (vstup a výstup) budou ve formátu JSON³.



Obrázek 3.3: Architektura REST [6]

3.3 Implementace serverové části

Pro běhové prostředí serverové části je užit framework .NET Core a ASP .NET Core. Aplikace je psána v jazyce C#. Aplikace byla při vývoji strukturována do několika od sebe logicky oddělených částí, v rámci frameworku .NET se tyto části nazývají projekty.

3.3.1 Generování kódu

Stěžejním projektem, který obsahuje aplikační logiku pro generování kódů je projekt s názvem CodeGenEngine, jedná se o projekt typu class library. Prakticky se jedná o knihovnu, která poskytuje rozhraní pro práci s generátorem kódu. Tato knihovna obsahuje veřejné třídy, rozhraní, které zmíněné třídy implementují a také třídy abstraktní. Vstupním bodem či rozhraním se zbytkem serverové části pro generování kódů je třída CodeGenerator, jejíž instance tvoří jednoduché rozhraní pro práci s generátorem. Instance této třídy se vytváří pro specifický programovací jazyk, do kterého je pak generován kód dle zadaných specifikací tříd z třídního diagramu.

Aby byl výsledný generátor kódu co možná nejlépe upravitelný a natolik obecný, aby bylo možné generovat kód tříd do libovolného jazyka bez nutnosti zásahu do aplikační logiky generátoru kódu tak, byla vytvořena slovníková struktura, obsahující definice a klíčová slova. Při samotném generování kódu je pak procházena specifikace třídy a jednotlivé definice jsou nahrazovány řetězci ze

³<https://www.json.org/json-en.html>

zmíněného slovníku. V těchto řetězcích se mohou vyskytovat klíčová slova, které umožní obecně navrhnout strukturu třídy a nahradit tato klíčová slova definovanými názvy ze specifikace třídy. V tabulce 3.1 se nachází popis jednotlivých klíčových slov, které je možné využít v rámci transformačních pravidel pro převod specifikace třídy na kód v daném jazyce. Samotnou strukturu třídy je nutné popsat dalšími specifickými klíčovými slovy, jejich význam je zapsán v tabulce 3.2.

Tabulka 3.1: Definice klíčových slov pro transformační pravidla

Klíčové slovo	Význam	Příklad nahrazení
INCLUDE	Název modulu	stdio.h
NAMESPACE	Název namespace	Demo
ACCESSOPERATOR	Operátor přístupu	private/public/...
CLASSNAME	Název třídy	User
<BASECLASES>	Seznam bázových tříd/interface	Human
DATATYPE	Datový typ	long, int, string
NAME	Název (Atributu)	
<ARGUMENTS>	Seznam argumentů metody/konstruktoru	
DEFAULTVALUE	Default hodnota	5

Tabulka 3.2: Definice struktury třídy

Klíčové slovo	Význam
INCLUDES_DECLARATION	reference knihoven, modulů...
NAMESPACE_DECLARATION	deklarace namespace
CLASS_DECLARATION	deklarace třídy
PRIVATE_PROPERTIES_DECLARATION	sekce deklarace privátních atributů třídy
PUBLIC_PROPERTIES_DECLARATION	sekce deklarace veřejných atributů třídy
DEFAULT_CONSTRUCTOR_DECLARATION	deklarace default konstruktoru
PARAMETRIZED_CONSTRUCTOR_DECLARATION	deklarace parametrického konstruktoru
GETTERS_AND_SETTERS_DECLARATION	Deklarace s definicí getterů a setterů
PUBLIC_METHODS_DECLARATION	Deklarace veřejných metod
PRIVATE_METHODS_DECLARATION	Deklarace privátních metod

Podoba specifikace slovníku pro jazyk C# pak ve formátu JSON vypadá následovně:

```
{  
    "IncludeTemplate": "using INCLUDE; ",  
    "NamespaceTemplate": "namespace NAMESPACE; ",  
    "ClassDeclarationWithoutBaseClassTemplate": "ACCESSOPERATOR class CLASSNAME",  
    "ClassDeclarationWithBaseClassTemplate": "ACCESSOPERATOR class CLASSNAME : <BASECLASSES  
    >",  
    "PropertyDefinititonTemplate": "ACCESSOPERATOR DATATYPE NAME { get; set; }",  
    "PropertyGetterTemplate": "ACCESSOPERATOR DATATYPE getName() { return this.NAME; }",  
    "PropertySetterTemplate": "ACCESSOPERATOR void setName(DATATYPE _NAME) { this.NAME =  
    _NAME; }",  
    "OpenDefinitonBodyTemplate": "{}",  
    "CloseDefinitonBodyTemplate": "}",  
    "DefaultConstructorDeclarationTemplate": "ACCESSOPERATOR CLASSNAME()",  
    "ParameterizedConstructorDeclarationTemplate": "ACCESSOPERATOR CLASSNAME(<ARGUMENTS>)",  
    "PublicMethodDeclarationTemplate": "ACCESSOPERATOR DATATYPE NAME(<ARGUMENTS>)",  
    "PrivateMethodDeclarationTemplate": "ACCESSOPERATOR DATATYPE NAME(<ARGUMENTS>)",  
    "ArgumentWithoutDefaultValueTemplate": "DATATYPE NAME",  
    "ArgumentWithDefaultValueTemplate": "DATATYPE NAME = DEFAULTVALUE",  
    "ClassTemplate": "INCLUDES_DECLARATION\r\nNAMESPACE_DECLARATION\r\nCLASS_DECLARATION\r\n  
n{\r\nPRIVATE_PROPERTIES_DECLARATION\r\nPUBLIC_PROPERTIES_DECLARATION\r\n  
nDEFAULT_CONSTRUCTOR_DECLARATION\r\nPARAMETRIZED_CONSTRUCTOR_DECLARATION\r\n  
nGETTERS_AND_SETTERS_DECLARATION\r\nPUBLIC_METHODS_DECLARATION\r\n  
nPRIVATE_METHODS_DECLARATION\r\n}",  
    "PropertyInitializationTemplate": "this.NAME = NAME;",  
    "FileExtensionType": "cs"  
}
```

Specifikace třídy je procházena způsobem, který je popsán prostřednictvím návrhového vzoru Visitor. Jedná se o návrhový vzor patřící do skupiny vzorů chování. Tento návrhový vzor umožňuje oddělit aplikační logiku od předpisů tříd a přidat či upravit funkcionality navázané na jednotlivé třídy bez nutnosti zásahu do těchto tříd. Uplatněním vzoru Visitor pak docílíme značné přehlednosti a schopnosti řídit průchod na sebe navazujícím třídám. Tento návrhový vzor se hodí i v případě tohoto projektu, potřebujeme efektivně projít specifikaci zadané třídy, která je popsána kompozicí objektů, a seznamu obsahující další objekty. Průchod popisovanou kompozicí tříd je řízen z třídy Language implementující rozhraní IVisitor. Po vyvolání metody Accept na kořenovém objektu je započata transformace specifikace třídy dle zvolených transformačních pravidel.

Třídy využívané pro popis tříd zpravidla implementují rozhraní IElement a IMapped. IElement je rozhraní, které obsahuje metodu Accept přijímající parametr instance třídy implementující rozhraní IVisitor a prostřednictvím této třídy je pak delegován průchod návrhovým vzorem Visitor. Jednotlivá klíčová slova jsou pak při průchodu nahrazovány dle specifikace definic ve slovníku jazyka.

Vazba klíčových slov na vlastnosti (property) objektu je pak interně mapivána ve třídě implementující rozhraní IMapped. Třída pak musí implementovat metodu s názvem GetMapped vracející objekt typu Dictionary (slovník, obsahující podporované klíčové slova a hodnoty pro nahrazení). Příklad implementace zmíněné metody se nachází níže.

```
public Dictionary<string, string> GetMapping()
{
    return new Dictionary<string, string>()
    {
        { "NAME", Name },
        { "DATATYPE", ReturnType.Key.ToString() },
        { "ACCESSOPERATOR", AccessOperator.ToString().ToLower() },
    };
}
```

Nahrazování klíčových slov pro jednotlivé objekty se odehrává při průchodu kompozice tříd definující podobu třídy. Průchod řídí instance třídy Language (vzor Visitor). Při generování kódu je také dbáno na dobrou čitelnost generovaného kódu. Podoba samotné třídy a rozmístění jednotlivých prvků a deklarací je specifikována ve slovníku jazyka pod klíčovým slovem ClassTemplate. Kromě toho je výsledný kód organizován podle aplikační logiky implementované v třídě Language, řízeno je také odsazování řádků, aby byl vygenerovaný kód s formátováním podobným tomu, které by použil člověk.

Třídní diagram tohoto projektu je součástí příloh této práce.

3.3.2 REST API

Rozhraním pro komunikaci se serverovou částí vyvíjeného systému pro generování kódu je projekt psán pod frameworkm ASP .NET. Pro účely tohoto projektu byla část aplikace poskytující REST API nakonfigurována tak, aby splňovala požadavky pro tzv. Open API. Open API⁴ (dríve pouze Swagger) popisuje, jaké operace jsou dostupné v API, jaké parametry přijímají a jaké odpovědi může uživatel očekávat. Tato specifikace se generuje automaticky (nebo na vyžádání) při sestavování aplikace, prostřednictvím tohoto API jsou pak dostupné jednotlivé metody API. Snímek obrazovky popisovaného Open API (Swagger UI) se nachází níže na obrázku 3.4.

Aplikace v jednotlivých REST metodách přijímá JSON specifikace, které jsou pak z textové podoby validovány a parsovány do jejich objektové reprezentace. Specifikace tříd, ze kterých vznikají tyto instance v aplikaci nesou v jejich názvu postfix „Ext“ (ClassExt, MethodExt apod.). V projektu „Converts“ se nacházejí statické metody ve statické třídě ExternalConverts tzv. extension metody (rozšiřující metody) s názvem „ConvertToInternal“, které rozšiřují samotnou funkcionalitu jmenovaných „Ext“ objektů. Tyto metody vracejí interní objektovou reprezentaci externích objektů,

⁴<https://swagger.io/>

Obrázek 3.4: Open API - server

které jsou užívány v samotném projektu generující kód ze specifikací tříd. Příkladem takové rozšiřující metody, která přidává instanci třídy „ClassExt“ neparametrickou metodu „ConvertToInternal“ vracející instanci třídy „ClassExt“. Následuje ukázka této metody.

```
public static Class ConvertToInternal(this ClassExt ext)
{
    Class convert = new Class()
    {
        Name = ext.Name,
        NameSpace = ext.NameSpace,
        AccessOperator = ext.AccessOperator.ConvertToInternal(),
        BaseClasses = ext.BaseClasses.Select(b => b.ConvertToInternal()).ToList(),
        Includes = ext.Includes.Select(i => i.ConvertToInternal()).ToList(),
        Methods = ext.Methods.Select(m => m.ConvertToInternal()).ToList(),
        Properties = ext.Properties.Select(p => p.ConvertToInternal()).ToList()
    };
    return convert;
}
```

Využití těchto externích objektů a následné mapování na interní reprezentace těchto objektů se při prvním pohledu zdá poměrně zbytečná. Jde však o klíčové rozhodnutí z hlediska návrhu softwaru, díky němuž jsme schopni oddělit aplikační logiku a rozhraní aplikace na straně serveru. Vrstva REST API slouží pouze k vypořádání požadavků se strany klienta a z hlediska bezpečnosti v této vrstvě nemá být přístup k aplikační logice. Navíc tímto rozhodutím docílíme logickému oddělení a členění aplikace, snižujeme provázanost (coupling) a naopak zvyšujeme míru koheze - tedy soudržnosti jednotlivých softwarových jednotek.

3.4 Implementace klientské části

Běhovým prostředím pro klientskou část aplikace pro generování kódu z třídního diagramu je Apache HTTP Server⁵. Aplikace slouží především k vytvoření UML třídního diagramu v grafickém uživatelském rozhraní. Po vytvoření třídního diagramu v aplikaci je model převeden na objektovou reprezentaci, následně převeden na JSON dokument a zaslán požadavek na běžící serverovou část, kde je vykonáno samotné vygenerování kódu dle specifikace z třídního diagramu ve zvoleném programovacím jazyce. V klientské aplikaci je zobrazen výčet podporovaných programovacích jazyků pro transformaci z třídního diagramu na kód. Tyto podporované jazyky jsou načteny za běhu aplikace, kdy je z klienta zaslán požadavek na získání výčtu identifikátorů podporovaných jazyků. Uživatel si pak vybírá z tohoto výčtu před odeláním požadavku ke generování kódů na server. Následně je ze serveru přijat vygenerovaný kód a dle aplikační logiky na straně klienta je kód zpracován, jsou vytvořeny zdrojové soubory obsahující zdrojové kódy. Dále je vytvořena adresářová struktura se soubory obsahující zdrojové kódy a ty jsou následně komprimovány a uživateli je umožněno stažení. Tento proces je popsán sekvenčním diagramem na obrázku 3.5.

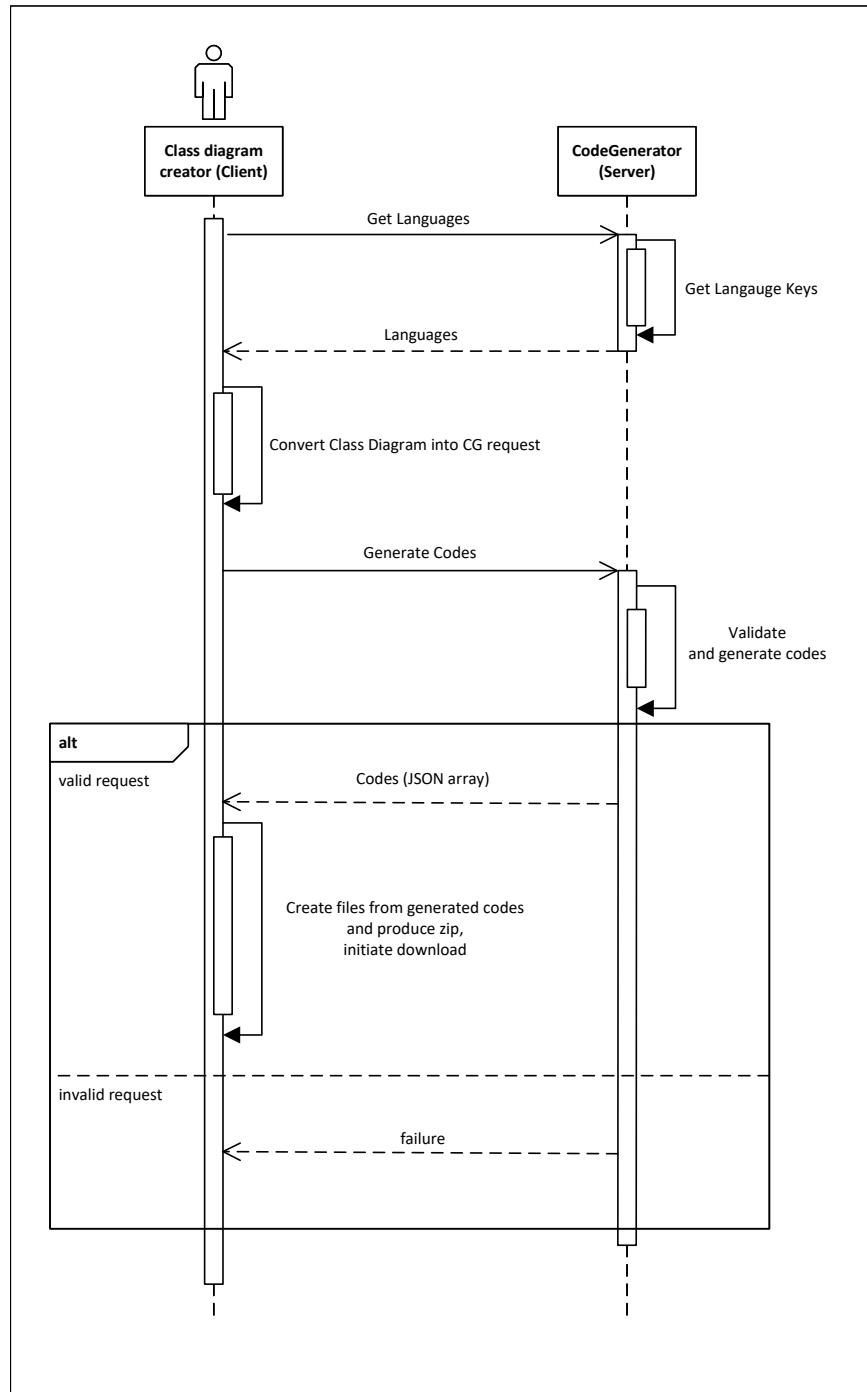
Uživatelské rozhraní klienstské aplikace je napsáno v jazyce HTML a grafická podoba je dána CSS skripty. Aplikační logika je napsána v jazyce JavaScript. Třídní diagram je na webové stránce zobrazován za pomocí frameworku GoJS⁶, který umožňuje nastavení podoby a vlastností diagramů. Třídní diagram je v paměti prohlížeče reprezentován strukturou objektů, tato struktura je pak konvertována do bodoboy, kterou vyžaduje serverová část generátoru kódů. Komunikace se serverovou částí je realizována prostřednictvím REST aplikačního rozhraní serveru.

Snímek obrazovky uživatelského rozhraní klientské části se nachází na obrázku 3.6.

Jak již bylo zmíněno výše, k zobrazování podoby třídního diagramu bylo využito frameworku GoJS. Uživatelské rozhraní klientské aplikace tvoří modul frameworku GoJS, který zobrazuje aktuální podobu třídního diagramu. Modifikace diagramu (přidávání, odebrání, editace) každého prvku je řízena aplikační logikou, která se vyvolává prostřednictvím rozhraní aplikace (formuláře, tlačítka). V levé části aplikace je menu s možnostmi úpravy jednotlivých částí diagramu a funkční prvky pro interakci/řízení se serverovou částí - generátorem kódů. Uživatel dynamicky interaguje s dia-

⁵<https://httpd.apache.org/>

⁶<https://gojs.net>



Obrázek 3.5: Sekvenční diagram vyobrazující postup generování kódu z třídních diagramů

mem. Kliknutí na libovolný objekt v diagramu vyvolá aktualizaci formulářových prvků, do kterých je načten aktuální obsah zvolené třídy (atributy, metody) apod. Uživatel pak může do jednotlivých tříd přidávat, či mazat jednotlivé atributy i metody. Mezi jednotlivými objekty v diagramu (mezi

The screenshot shows a user interface for generating code from UML class diagrams. On the left, there's a sidebar with buttons for management (Global, Node, Linking, Property, Method) and configuration (Generator URL: http://localhost:5000/, Load Languages: CSharp). Below these are buttons for Import configuration, Export configuration, Export SGV diagram, and GenerateCode.

The main area displays a UML class diagram with the following classes and associations:

```

classDiagram
    class IdentifiableEntity {
        -Id: int
    }
    class User {
        -FirstName: string
        -LastName: string
        -_accountArray: Account[]
        +Pay(amount: int): bool
        -GetAccountBallance(): int
    }
    class Account {
        -Ballance: int
    }
    User "1" -- "*" Account : _accountArray

```

On the right, there's a panel for adding methods to the User class. It includes fields for Method Name (Enter method name), Visibility (Public selected), Return Type (string), Parameters (Add Parameter), and a RemoveMethod button. A dropdown menu shows the current method definition: `private GetAccountBallance() : int`.

Obrázek 3.6: Uživatelské rozhraní generátoru kódu z třídního diagramu

třídami) je možné tvorit relace prostřednictvím tzv. linků (propojů). Uživatel může zvolit mezi relací asociace, agregace, kompozice nebo generalizace. Potvrzením této akce je pak změna vyobrazena na diagramu a je změněn vnitřní stav objektové reprezentace třídního diagramu. Relace asociace, agregace a kompozice pak ještě upravuje i samotné atributy třídy, kde je dodán atribut dle zvolené vazby na zvolený objekt (třídu). Např. relace asociace vyvolá akci, která přidá atribut datového typu, se kterým je zvolený objekt v relaci. Aplikace také umožňuje import a export kompletního nastavení a podoby třídního diagramu. Tyto funkcionality byly implementovány na straně klienta, byly vytvořeny parsery a funkce, které umožňují export a import z JSON dokumentu. Struktura

JSON dokumentu pro tyto účely (na straně klienta) je obdobná struktura, která je využívána v rámci endpointů na straně serveru k definici podoby tříd. Zmíněné struktury však nejsou totožné či zaměnitelné, protože diagram na straně klienta je složen ze samotných tříd, atributů, metod a linků, struktura taktéž obsahuje tzv. globální nastavení pro diagram, kde je např. namespace nebo požadované knihovny. Byla taktéž implementována funkcionality, která uloží aktuální podobu diagramu ve formátu svg.

3.5 Nasazení

Pro účely snadného nasazení systému generátoru kódů a klinta, který slouží k obsluze generátoru, byla vytvořena Docker⁷ konfigurace. Konfigurační soubory jsou součástí repozitáře obsahující zdrojové kódy (<https://github.com/jkonvicka/CodeGenerator>). Velmi snadné spuštění kontejnerů s klienskou a serverovou aplikací zajišťuje konfigurace Docker Compose, prostřednictvím této konfigurace se spustí kontejnery s oběma aplikacemi, nastaví se síťové spojení mezi kontejnery tak, aby bylo možné z kliencké aplikace obsluhovat serverovou část (generátor kódů).

Klientská aplikace je po spuštění dostupná adrese `http://localhost/Client` na portu 80 a generátor kódů na adrese `http://localhost:5000` (port 5000). Swagger UI se nachází na adrese `http://localhost:5000/swagger/index.html`. Prostřednictvím příkazu `docker-compose up -build` v kořenovém adresáři repozitáře s kódem je možné nasadit celý systém do Docker kontejnerů.

Následuje ukázka kódu pro službu docker-compose, která sdružuje spuštění jednotlivých služeb:

```
version: '3.9'
services:
  server:
    build:
      context: ./Server
      dockerfile: ./CodeGenRestAPI/Dockerfile
    ports:
      - "5000:5000"

  client:
    build:
      context: ./Client
      dockerfile: ./Dockerfile
    ports:
      - "80:80"
```

⁷<https://www.docker.com/>

Kapitola 4

Experimenty

V rámci testování nástroje pro generování kódu z třídních diagramů, jehož vznik je popisován touto prací, byla provedena sada testů s experimentálními konfiguracemi a specifikacemi třídních diagramů. Následující text pojednává o průběhu provádění těchto testů včetně zhodnocení výsledů testů.

4.1 Specifikace jazyka a transformačních pravidel

K ověření korektnosti návrhu a finální implementace serverové části nástroje byla vytvořena čtveřice konfiguračních souborů obsahující konfiguraci jazyka a transformačních pravidel s užitím klíčových slov pro převod specifikace třídy (třídní diagram) na kód (ve formátu JSON). Tyto konfigurační soubory jsou uloženy v adresáři na straně serveru¹ (výkonná část s logikou pro průchod struktury třídy a generování kódu). Při inicializaci serverové části jsou tyto konfigurace jazyků načteny do paměti. Načtené jazyky pak mohou být využity při žádosti o vygenerování kódu.

4.2 Proces testování

Pro účely generování kódu je na straně serveru v REST API specifikován endpoint, na jehož vstupu je zadávána specifikace třídy a požadovaný jazyk pro generování kódu. Pokud je identifikátor požadovaného jazyka dostupný ve službě poskytující transformační pravidla (na straně serveru), pak je zahájena transformace specifikace třídy na kód v daném jazyce dle definovaných transformačních pravidel. Identifikátor jazyka je shodný s názvem konfiguračního souboru, který je načítán při inicializaci serverové části nástroje pro generování kódu.

¹<https://github.com/jkonvicka/CodeGenerator/tree/main/Server/CodeGenRestAPI/Configuration/Languages>

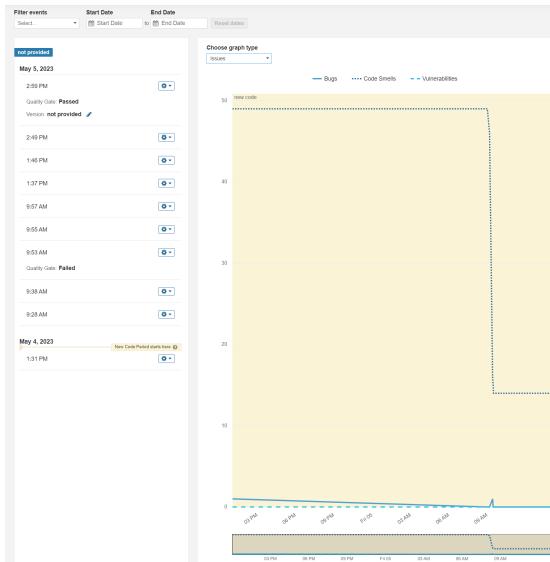
Kód generátoru kódu byl testován dle V-modelu. Pro účely testování jednotlivých modulů byly vytvořeny tzv. unit testy, výsledky testů jsou součástí příloh této práce. K jednotkovému testování byl využit framework NUnit².

V rámci integračního testování byl generátor kódů otestován tzv. smoke testy, nejdříve zasláním HTTP requestů na endpointy prostřednictvím aplikace Postman³, pak v rámci klientské části.

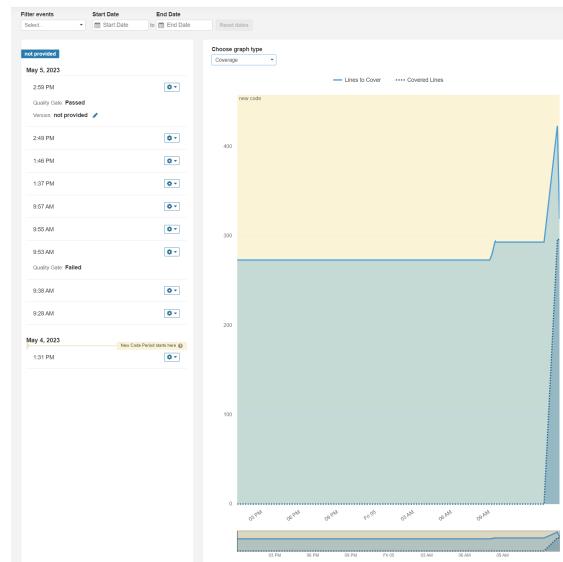
4.3 Měření kvality kódu

Pro ověření kvality kódu serverové části nástroje pro generování kódu byl použit nástroj SonarQube⁴ spuštěný v Docker kontejneru na lokální pracovní stanici. Tento nástroj slouží k automatizovanému statickému hodnocení kódu z hlediska kvality, včetně detekce chyb, závislostí, duplicit kódu, nedostatků v návrhu apod.

Pro spuštění analýzy kódu byla spuštěna instance nástroje SonarQube nad zdrojovým kódem serverové části nástroje pro generování kódu a manuálně byl vyvoláván proces měření. Na základě výstupů byly identifikovány problémy a nedostatky v kódu, které byly následně odstraněny. Výstupy v podobě grafů, které byly zpracovány nástrojem SonarQube se nacházejí na obrázcích 4.1 a 4.2.



Obrázek 4.1: SonarQube - počet chyb v kódu v čase



Obrázek 4.2: SonarQube - pokrytí kódu testy v čase

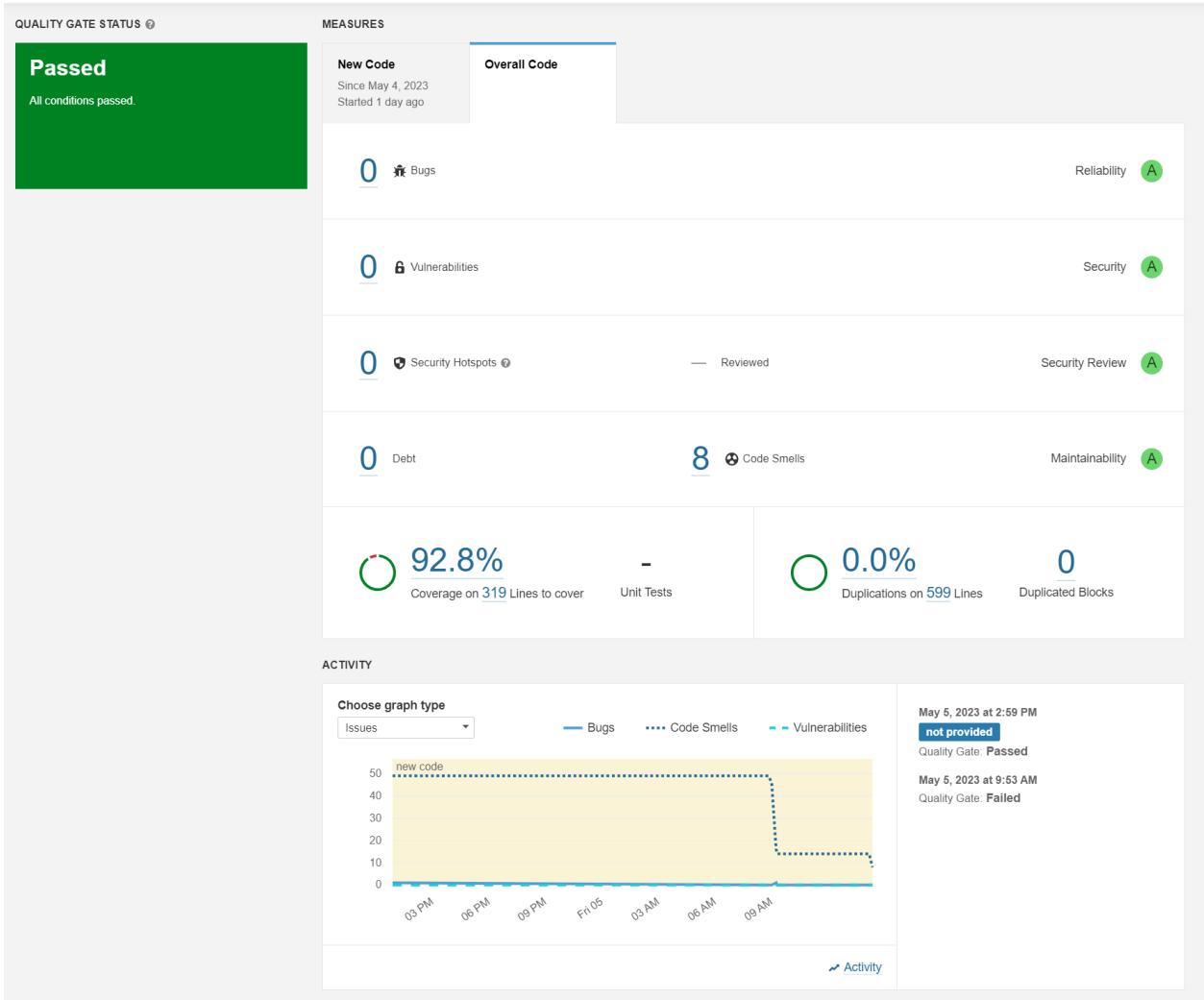
Součástí výstupu z nástroje SonarQube byla také informace o pokrytí kódu testy. Pro ověření tohoto pokrytí byly manuálně spuštěny UNIT testy serverové části a jejich výsledky byly analyzovány. Na základě výstupů z testů bylo zjištěno, že výsledné pokrytí kódu testy bylo 92,8 %, což bylo

²<https://nunit.org/>

³<https://www.postman.com/>

⁴<https://www.sonarqube.org/>

v rámci tohoto projektu považováno za dostatečné. Po odstranění nalezených chyb a nedostatků byl kód serverové části nástroje pro generování kódu optimalizován s ohledem na výstupy z nástroje SonarQube a výsledné pokrytí kódu testy. Výstup SonarQube summarizující kvalitu kódu se nachází na obrázku 4.3.



Obrázek 4.3: SonarQube - přehled výsledků měření

Celkově lze konstatovat, že použití nástroje SonarQube přispělo k zlepšení kvality kódu serverové části nástroje pro generování kódu z třídních diagramů, byly opraveny chyby, které nástroj označil jako „Major a Critical“, chyby typu „Blocker“ se ve fázi použití nástroje SonarQube nenacházely. Exportované výsledky, které je možné importovat do nástroje SonarQube jsou součástí příloh této práce.

4.4 Výsledky

Nástroj na generování kódu byl testován samostatně prostřednictvím webové aplikace Swagger UI, která byla vygenerována v závislosti na specifikaci REST API prostřednictvím ASP .NET Core. Testování také probíhalo na straně klienta a prostřednictvím aplikace Postman, která umožňuje přesnou konfiguraci zaslaných žádostí na server (úpravy hlaviček, těla requestu apod.).

Byla testována funkčnost uživatelského rozhraní a komunikace se serverovou části generátoru. V rámci tohoto testování byly vytvořeny třídní diagramy a z těchto diagramů byly vygenerovány kódy tříd v jazyce Java, C#, C++ a Python. Tímto byla ověřena korektnost samotného návrhu řešení tohoto projektu. Dále tím bylo ověřeno to, že konfigurace jazyka a míra obecnosti pro specifikaci transformačních pravidel umožňuje generování kódu do takto specificky jiných a od sebe lišících se jazyků jako je např. C#, Python a C++. A to bez nutnosti zásahu do aplikační logiky, veškeré úpravy byly prováděny jen v rámci specifikace transformačních pravidel. K úpravě jednotlivých transformačních pravidel na straně serveru zajišťující generování kódu není nutné zasahovat do kódu, pro změnu pravidel aplikaci generující kód nemusíme znova kompilovat. Stačí upravit konfiguraci transformačního pravidla v JSON dokumentu daného jazyka a restartovat serverovou část.

Byla také testována kvalita kódu serverové části plnící funkci generování kódu z třídního diagramu. Výsledky a popis přínosu tohoto testování byly popsány výše.

Data užitá k testování a výše popsaným experimentům, včetně výsledků jsou součástí příloh této práce.

Kapitola 5

Závěr

V rámci této semestrální práce byla úspěšně navržen a implementován nástroj pro generování kódu z třídního diagramu. Uživatel může vytvořit třídní diagram v grafickém uživatelském rozhraní, který je následně převeden na objektovou reprezentaci a odeslán požadavek na serverovou část. Serverová část poté generuje kód dle specifikace z třídního diagramu ve zvoleném programovacím jazyce a výsledný kód je přijat a zpracován klientskou částí aplikace. Výsledkem je adresářová struktura se soubory obsahujícími zdrojové kódy, které jsou uživateli k dispozici ke stažení.

Klientská část aplikace je napsána v jazyce HTML, CSS a JavaScript, a využívá frameworku GoJS pro zobrazování třídního diagramu v grafickém uživatelském rozhraní. Komunikace se serverovou částí je realizována pomocí REST aplikačního rozhraní serveru.

Výsledná aplikace umožňuje uživatelům snadno vytvářet třídní diagramy a generovat kód ve zvoleném programovacím jazyce. Tento projekt může být využit jako základ pro další rozšíření o další funkce nebo podporu pro více programovacích jazyků. Nabízí se také rozšíření o autentizaci na straně klienta i serveru, možnost načtení transformačních pravidel (konfigurací) z různých typů úložišť (databáze, endpoint) apod.

Součástí této práce jsou přílohy, které obsahují report odpracovaného času na projektu, dokumenty popisující návrhové rozhodnutí, konfigurace transformačních pravidel a výsledky experimentů a také zdrojové kódy.

Literatura

1. HERRINGTON, Jack. *Code Generation in Action*. 1. vyd. 209 Bruce Park Avenue, Greenwich: Manning Publications Co., 2003. ISBN 1930110979.
2. *DeepMind AlphaCode* [online]. [cit. 2023-03-15]. Dostupné z: <https://www.deepmind.com/blog/competitive-programming-with-alphacode>.
3. BELTRAMELLI, Tony. Pix2code: Generating Code from a Graphical User Interface Screenshot [online]. 2017 [cit. 2023-03-14]. Dostupné z DOI: 10.48550/ARXIV.1705.07962.
4. *IBM Rhapsody* [online]. [cit. 2023-03-14]. Dostupné z: <https://www.ibm.com/products/uml-tools>.
5. *Enterprise Architect* [online]. [cit. 2023-04-01]. Dostupné z: <https://sparxsystems.com/>.
6. *Create a web API with ASP.NET Core* [online]. [cit. 2023-04-02]. Dostupné z: <https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio>.