VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA KÓDU A MINIMALIZACE GRAFU ZÁVISLOSTI PROJEKTŮ

CODE ANALYSIS AND PROJECT DEPENDENCY GRAPH MINIMIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **OLIVER GOLEC**
AUTHOR

VEDOUCÍ PRÁCE **doc. Dr. Ing. DUŠAN KOLÁŘ,**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

148275

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Golec Oliver**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Analýza kódu a minimalizace grafu závislosti projektů**
Kategorie:	Překladače
Akademický rok:	2022/23

Zadání:

1. Prostudujte různé možnosti redukce grafu závislostí projektů. Popište způsoby řešení problému uzlů grafu, které nejdou dále zredukovat. Prostudujte možnosti transformace syntaxe mezi použitými jazyky pro případnou další minimalizaci.
2. Na základě bodu 1 navrhněte aplikaci, která bude schopna zanalyzovat předloženou sadu *.NET solution* sestávající se z projektů ve VB.net a C#, provést modelaci minimalizace projektových referencí s doporučením optimální strategie a provést kódové změny.
3. Navrženou aplikaci implementujte v C# dle požadavků vedoucího.
4. Aplikaci otestujte na sadě s *legacy* kódem s velkým množstvím projektů - sadu konzultujte s vedoucím.
5. Zhodnoťte svůj přínos a diskutujte případné nedostatky či možná vylepšení.

Literatura:

- Aho, Alfred Vaino; Lam, Monica Sin-Ling; Sethi, Ravi; Ullman, Jeffrey David (2006). Compilers: Principles, Techniques, and Tools (2 ed.). Boston, Massachusetts, USA: Addison-Wesley. ISBN 0-321-48681-1
- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:
První 2 body zadání a rozpracovaný bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Kolář Dušan, doc. Dr. Ing.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	24.10.2022

Abstrakt

Rozsiahle projekty často obsahujú *legacy* komponenty so zložitými grafmi závislostí. V prípade, že tieto komponenty prekročia rozumnú veľkosť a závislosti medzi nimi je nemožné upravovať ručne, prichádza na rad automatizované riešenie. Táto práca sa zaoberá procesom vytvárania aplikácie, ktorá minimalizuje graf závislostí na platforme .NET za použitia efektívnych grafových algoritmov a nástrojov pre manipuláciu s .NET projektami. Práca popisuje konfiguračné zložky platformy .NET, predstavuje algoritmus na redukciu projektových závislostí a pomocou nástroja Roslyn a knižníc platformy MSBuild implementuje aplikáciu na redukciu závislostí. Výsledná aplikácia redukuje počet *legacy* projektov o približne 80 percent.

Abstract

Extensive projects often contain *legacy* components with complicated dependency graphs. In case they exceed reasonable size and their references are impossible to untangle manually, automated solution is inevitable. This work studies the proccess of creating an application that minimizes dependency graph on .NET platform using efficient graph algorithms and tools for .NET project manipulation. Thesis describes .NET platform configuration, introduces algorithm for project dependency reduction and using Roslyn and MSBuild libraries implements application that reduces dependencies. As a result, application reduces project dependency graph by approximately 80 percent.

Kľúčové slová

Acyklický orientovaný graf, .NET, C#, Visual Basic, Roslyn, NuGet, Syntax, Strom, Projekt, Solution

Keywords

Acyclic directed graph, .NET, C#, Visual Basic, Roslyn, NuGet, Syntax, Tree, Project, Solution

Citácia

GOLEC, Oliver. *Analýza kódu a minimalizace grafu závislosti projektů*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Dr. Ing. Dušan Kolář,

Analýza kódu a minimalizace grafu závislosti projektů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána doc. Dr. Ing. Dušana Koláře. Ďalšie informácie mi poskytli pán Ing. Peter Solár a pán Ing. Jan Vala. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Oliver Golec
9. mája 2023

Poďakovanie

Za pomoc a venovaný čas počas vypracovávania tejto práce by som rád poďakoval pánovi doc. Dr. Ing. Dušanovi Kolářovi. Za možnosť realizácie tejto práce spoločnosti FNZ a za cenné rady a konzultácie pánovi Ing. Petrovi Solárovi a pánovi Ing. Janovi Valovi.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Vo veľkých projektoch je možné stretnúť sa s takzvanými *legacy* komponentami, s pomerne zložitými grafmi závislostí. Tieto komponenty patria medzi najstaršie súčasti projektov a spolupracujú s novými, alebo sa postupne nahrádzajú. Ak neprebieha pravidelná údržba a refaktorizácia *legacy* kódu, veľmi rýchlo sa stane, že komponenty prerastú rozumnú veľkosť aj pre funkčne pomerne jednoduché projekty. V okamihu, kedy je závislosti medzi projektami príliš zložité upravovať a redukovať ručne, prichádza na rad automatizované riešenie.

Je prakticky nemožné prechádzať, či ručne upravovať závislosti medzi stovkami .NET projektov a previesť zmeny v konfiguračných a zdrojových súboroch. Rozsiahly *legacy* kód s množstvom závislostí negatívne ovplyvňuje a spomaľuje prácu softvérového vývojára, kedy vývojové prostredie, podporné nástroje, ale aj preklad a zostavenie programu pracujú kvôli komplexnému vyhodnocovaniu pomaly.

Bakalárska práca sa zaoberá návrhom aplikácie, ktorá umožní analýzu a následnú redukciu .NET závislostí medzi projektami programovacích jazykov C# a VB.NET. Celý proces je prevedený s využitím grafových algoritmov na redukciu orientovaných acyklických grafov, nástrojov pre analýzu a transformáciu kódu od spoločnosti Microsoft. Práca vznikla v spolupráci so spoločnosťou FNZ, ktorá potrebuje refaktorizačný nástroj pre .NET projekty, aby bola práca s ich kódovou štruktúrou efektívnejšia, rýchlejšia a prehľadnejšia.

Nasledujúce kapitoly popisujú, ako sú kód a konfiguračné zložky na platforme .NET usporiadané, predstavujú algoritmy pre redukciu grafu závislostí a implementáciu zlučovania .NET projektov s popisom používaných nástrojov a postupov. Na konci sú zhrnuté výsledky testov a záverečné zhodnotenie práce.

Kapitola 2

Štruktúra problému

Táto kapitola rozoberá súčasný stav *legacy* kódu spoločnosti FNZ, definuje základné členenie na platforme .NET a rozoberá existujúce nástroje, ktoré sa používajú na každodennú prácu a umožňujú manipulovať s členením .NET projektov. Znalosti z tejto kapitoly pomôžu uviesť do kontextu úkony potrebné pri zlučovaní projektov v rámci minimalizácie závislostí.

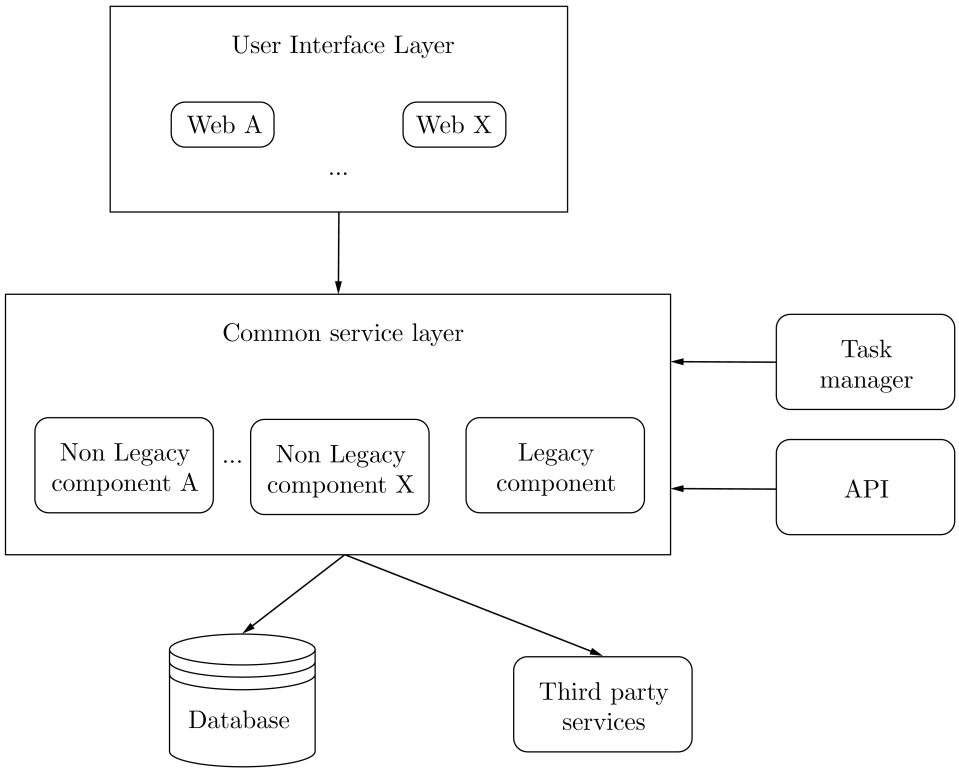
2.1 Spoločnosť FNZ

Spoločnosť FNZ[1] sa zaoberá vývojom finančných platforiem za pomoci .NET technológií od spoločnosti Microsoft[2]. Vývoj rokmi prešiel od jazyka VB.NET k jazyku C#, čo za sebou zanechalo rozsiahly *legacy* kód, ktorý je obtiažne zredukovať. Pri pohľade na kódovú štruktúru sa naskytne hneď niekoľko problémov:

- Vyše 200 projektov vo vyše 200 štruktúrach *solution*.
- Projekty pozostávajú zo zmesi jazykov C# a VB.NET, čo ešte pridáva na obtiažnosti riešenia.
- Kód jednej *legacy solution* štruktúry obsahuje vyše 2 milióny riadkov kódu.
- Na jednu *solution* štruktúru pripadá približne 30 000 zdrojových súborov.
- Nástroje používané pri vývoji často nezvládajú takéto kvantum kódu a projektov, napríklad Resharper alebo Visual Studio.
- Samotný preklad a zostavenie programu trpia rovnako, pretože sa musia vyhodnotiť všetky závislosti pre jednotlivé projekty, preložiť ich a prepojiť.
- Niektoré z projektov sa ani nevyužívajú, pretože veľké časti z nich boli nahradené novými projektami, nástrojmi, alebo už nie sú potrebné.
- Veľa projektov vychádza z takzvaného kódového jadra a vývojovými tímami sa ďalej samostatne upravujú pre konkrétne produkčné riešenia.

Refaktorizačný nástroj popisovaný v tejto práci môže výrazne zredukovať počet projektov, čím prispeje k urýchleniu práce vývojárov a sprehľadneniu kódu. Nasledujúci diagram reprezentuje typickú kódovú štruktúru:

[1] `https://fnz.com`
[2] `https://www.microsoft.com/cs-cz`



Obr. 2.1: Diagram štruktúry kódu FNZ.

Diagram napomáha si uvedomiť, že v prípade zlučovania viacerých projektov v rámci jednej štruktúry *solution* je nutné upraviť zmenené referencie aj mimo meneného *legacy* komponentu. Výsledné riešenie preto bude musieť rozanalyzovať celý kód, vrátane ďalších štruktúr *solution* a nie len samotný komponent. Rovnako je možné si všimnúť, že závislosti sa nevyskytujú vo vnútri vrstiev, ale naprieč nimi. Napríklad vo vrstve používateľského rozhrania medzi sebou nemajú referencie jednotlivé weby.

2.2 Používané programovacie jazyky

Pretože sada *solution* pozostáva z mixu programovacích jazykov C# a VB.NET je nutné poznať ich podobnosti a špecifiká.

2.2.1 Programovací jazyk C#

Programovací jazyk C# [9] je mnohoúčelový, objektovo orientovaný jazyk. Vznikol v roku 2000 a jeho prvá verzia bola sprístupnená v roku 2002. Bol vytvorený za účelom všeobecného využitia, pričom dbá na jednoduchosť a efektívnosť. Jedná sa o staticky typovaný, typovo bezpečný jazyk, čo znamená, že je zaručaná typová bezpečnosť už v čase prekladu. Narozdiel od jazyka *VB.NET* nedovoľuje upraviť možnosti prekladu v súvislosti s typovaním, preto sú pri preklade projekty jazyka C# medzi sebou konzistentné.

Bol navrhnutý primárne pre technológie spoločnosti Microsoft, konkrétne platformu *.NET Framework* (V dnešných dňoch nahradená novšou *.NET Core.*), no používa sa aj pre iné platformy. Napríklad *Mac OS X*, *iOS* alebo *Android.*

2.2.2 Programovací jazyk Visual Basic .NET

Programovací jazyk *VB.NET* (Visual Basic .NET) [17] je jazyk vyvinutý spoločnosťou Microsoft pre platformu *.NET Framework*. Prvýkrát bol predstavený v roku 2002 a nadviazal na svojho predchodcu *VB6*. Možno konštatovať, že k dnešnému dňu jeho použitie pre platformu *.NET* výrazne nahradil spomínaný jazyk C# , napriek tomu, že obidva jazyky vznikli v približne rovnakom čase. Jedná sa o objektovo orientovaný jazyk, a na rozdiel od svojich predchodcov podporuje neobmedzenú dedičnosť.

Jazyk *VB.NET* bol navrhnutý, aby mohol byť použitý na mnoho účelov a druhov aplikácií v operačnom systéme *Windows*. Napríklad webové aplikácie, webové služby alebo desktopové aplikácie.

Z hľadiska zostavenia výsledného programu a minimalizácie závislostí projektov sú dôležité nastavenia možností prekladu. Konkrétne to sú „*Option Strict*“ a „*Option Explicit*“, ktoré môžu byť nastavené na úrovni súborov, rovnako tak na úrovni projektov v konfiguračných súboroch. Prirodzene, nastavenie v zdrojovom súbore má pre prekladač vyššiu prioritu.

- **Option Strict** – Slúži na špecifikáciu podpory implicitných konverzií dátových typov.
- **Option Explicit** – Slúži na špecifikáciu podpory vopred nedeklarovaných premenných.

```
Option Strict [On | Off]
Dim intNumber = 100
Dim longNumber As Long = 1000000
Dim intNumber2 As integer = longNumber
...
Option Explicit [On | Off]
Dim intNumber3 As Integer = 100
intNumber4 = intNumber3
```

Výpis 2.1: Ukážka možností prekladu programovacieho jazyka VB.NET [17].

V ukážke vyššie sú uvedené príklady, ako možnosti ovplyvňujú preklad programu. V prípade, že by bola nastavená možnosť *Strict*, program nie je preložiteľný, pretože pri deklarácii premennej *intNumber* absentuje špecifikácia dátového typu a pri priradení hodnoty do premennej *intNumber2* nie je typová zhoda. Následne, pri možnosti *Explicit*, nie je možné priradiť hodnotu do vopred nedeklarovanej premennej.

2.3 Hierarchické usporiadanie platformy .NET

V tejto časti bude predstavená logická a súborová štruktúru .NET projektov. Celá podkapitola sa opiera o informácie z knihy *Mastering Visual Studio .NET* [12].

2.3.1 Solution

Štruktúra .NET *solution* daná konfiguračným súborom s príponou „.sln“, ktorá združuje .NET projekty sa nachádza na najvyššej úrovni. Jedná sa o pomyselný kontajner, ktorý obsahuje navzájom prepojené projekty spolu s informáciami potrebnými pre preklad, zostavenie programu, či nastavenia pre vývojové prostredie Visual Studio. Hierarchicky pod

štruktúrou *solution* sa nachádzajú spomínané projekty. Projektami možno rozumieť kolekciu súborov, logický celok preložiteľný na knižnicu, spustiteľný súbor, alebo webovú stránku. Pod jednou štruktúrou *solution* sa preto typicky nachádzajú rôzne druhy projektov v závislosti na ich účele, napríklad spomínaná webová stránka ako prvok používateľského rozhrania, alebo konzolová aplikácia pre databázovú vrstvu. V kódovom *legacy* komponente *solution* určenom na minimalizáciu sa vyskytujú výhradne projekty typu knižnice, ktoré sa ďalej využívajú v iných projektoch.

Pre každý projekt je definovaný samostatný konfiguračný súbor vo formáte XML s informáciami pre platformu MSBuild, či vývojové prostredie Visual Studio. Práve obsah konfiguračných súborov z hľadiska analýzy závislostí dôležitý.

Konfiguračný súbor pre *solution*

Súbor s príponou „.sln“ je textový konfiguračný súbor, v ktorom sa nachádzajú informácie o konfigurácii štruktúry *solution*, všetkých projektoch a doplňujúce informácie pre prostredie Visual Studio. Všetky projekty majú v rámci súboru jedinečný identifikátor *GUID*[3] a cestu k jeho konfiguračnému súboru, ktorý o sebe nesie aj informáciu o jazyku a type. Vývojové prostredie Visual Studio si podľa týchto informácií načíta dané projekty a stiahne potrebné balíčky. V niektorých prípadoch môže obsahovať konfiguračný súbor pre *solution* aj informácie o projektových závislostiach, ktoré sú dnes už však redundantné, pretože všetky informácie o závislostiach medzi projektami v sebe nesú ich konfiguračné súbory.

Konfiguračný súbor pre projekt

Projektový konfiguračný súbor s príponou „.csproj“, respektíve „.vbproj“ nesie informácie o svojom type, štruktúre, závislostiach na iných projektoch, ale aj závislostiach na externých balíčkoch, alebo lokálnych knižničných súboroch. Z hľadiska výslednej aplikácie sú dôležité práve projektové a externé závislosti, pretože pri zlučovaní projektov je potrebné ich presúvať a môžu medzi nimi nastávať konflikty. V nasledujúcej ukážke je možné vidieť, čo môže byť v projekte pri preklade zahrnuté a nakonfigurované. Postupne zhora to sú:

- **System** – všetky súčasti štandardnej knižnice *System*.
- **Automapper** – Externý balíček, knižnica s názvom *Automapper*. Stiahne sa automaticky pri preklade.
- **Announcer.cs** – Zdrojový súbor, ktorý má byť preložený a zobrazený v prostredí Visual Studio.
- **VProject6.vbproj** – Závislosť v podobe projektu programovacieho jazyka VB.NET. Projekt s názvom „VProject6“ musí byť preložený skôr ako projekt s daným konfiguračným súborom.

[3]GUID (Globally unique identifier) – Unikátna 128 bitová hodnota.

```
...
<ItemGroup>
        <Reference Include="System" />
        ...
</ItemGroup>
<ItemGroup>
    <PackageReference Include="AutoMapper" Version="12.0.1" />
</ItemGroup>
<ItemGroup>
        <Compile Include="Announcer.cs" />
        ...
</ItemGroup>
<ItemGroup>
        <ProjectReference Include="..\VProject6\VProject6.vbproj">
                <Project>{4b38dcd1-3a1f-401f-af5d-6d9791064c6f}</Project>
                <Name>VProject6</Name>
        </ProjectReference>
        ...
</ItemGroup>
...
```

Výpis 2.2: Ukážka konfiguračného súboru pre projekt jazyka C#.

2.3.2 Menný priestor

V kóde lokalizuje jednotlivé objekty menný priestor, *Namespace*. Jedným z hlavných dôvodov logického členenia kódu do menných priestorov je predchádzanie konfliktom. Je vysoká šanca, že v tisíckach zdrojových súborov majú isté triedy rovnaký názov. Pre identifikáciu odkiaľ pochádzajú jednotlivé triedy, rozhrania, vymenované typy, delegáty, štruktúry a v prípade jazyka *VB.NET* aj moduly, slúžia direktívy použitia označené kľúčovým slovom *using*, respektíve *import*. Za direktívou použitia spravidla nasleduje identifikátor menného priestoru. Direktívy zabezpečujú, že v kóde sa môže nachádzať inštancia jedného zo spomínaných objektov bez toho, aby bolo pri deklarácii nutné uviesť identifikátor menného priestoru, v ktorom sa daný objekt nachádza. Podľa konvencií sa identifikátory označujú hierarchicky, na základy súborovej štruktúry. Princíp vnorenia zložiek v súborovom systéme funguje aj v menných priestoroch. Nejedná sa však o pravidlo, ale o konvenciu, preto neplatí pri všetkých *legacy* projektoch, ktorými sa práca zaoberá.

```
using System;

namespace Level1
{
        namespace Level2
        {
                namespace Level3
                {
                        class Customer
                        {
                        ...
namespace Level1.Level2.Level3
{
...
```

Výpis 2.3: Ukážka syntaxe menného priestoru Namespace v jazyku C#.

Ukážka demonštruje, akým spôsobom sa v jazyku C# usporadúva menný priestor. Obdobne to funguje aj v jazyku VB.NET. Na ukážke je možné vidieť štyri menné priestory, pričom *Level1.Level2.Level3* je ekvivalentom pre vnorený priestor *Level3*.

Projekty môžu mať nastavené direktívy použitia aj implicitne, kedy sú globálne uložené v konfiguračných súboroch na úrovni projektov. Objekt je možné v kóde používať aj bez nutnosti špecifikácie direktívy použitia. V tom prípade je nutné použiť *kvalifikovaný názov objektu* [9], čo pozostáva z názvu menného priestoru a názvu typu, napríklad „*Web.Utils.ProductEnum*“. Bodkou sú zľava oddelené jednotlivé časti názvu menného priestoru a najviac vpravo je názov vymenovaného typu.

2.3.3 Zostavenie

Zostavenie (Assembly) [6] je v rámci platformy .NET označenie pre spustiteľný súbor, alebo dynamicky prepojenú knižnicu vo formáte „.exe“, respektíve „.dll“. Ako napovedá názov, jedná sa o zostavenie výsledného súboru pre každý z projektov. V určitých prípadoch môže jedno zostavenie obsahovať výsledok prekladu viacerých projektov zlúčených do jedného. To je možné napríklad pomocou nástroja *ILMerge*[4]. Zostavenie poskytuje CLR[5] (Common Language Runtime), čo je označenie pre virtuálne prostredie, ktoré za behu spúšťa kód a uľahčuje vývoj. Do pamäte programu sa súbor zostavenia načíta len v prípade potreby, ak je v programe volaný, a tak napomáha efektívnej práci so zdrojmi.

Z pohľadu zostavenia projektov je výsledný súbor dôležitý, pretože ak majú projekty medzi sebou závislosti, tak musí závislý projekt počas prekladu čakať na zostavenie všetkých svojich závislostí. Súbor zostavenia je tiež možné pridať k projektu ako samostatnú referenciu. Nakoniec, na tom je postavená aj kódová *legacy* štruktúra, kedy sú štruktúry *solution* závislé na knižničných súboroch z externých projektov. Preto sa na hromadný preklad používa skript, ktorý každú štruktúru *solution* zostaví vo vopred definovanom poradí.

Okrem definícií typov, verzií, obsahujú súbory zostavenia aj takzvané *Metadáta* [5]. Jedná sa o dáta, ktoré nesú informáciu o iných, primárnych dátach. Metadáta poskytujú interoperabilitu medzi programovacími jazykmi, pretože informácie o všetkých typoch definovaných v kóde nesú jazykovo neutrálnym spôsobom. Aj vďaka metadátam môžu projekty

[4] `https://github.com/dotnet/ILMerge`

[5] `https://learn.microsoft.com/en-us/dotnet/standard/clr`

v jazyku C# a projekty v jazyku VB.NET spolu tvoriť jednu štruktúru *solution* a odkazovať navzájom na seba v kóde.

2.3.4 Nástroj MSBuild

Ďalším pojmom potrebným pre pochopenie štruktúry a organizácie prostredia .NET je *MSBuild* [7]. Jedná sa o nástroj slúžiaci na riadenie štruktúry kódu nad platformou .NET, správu závislostí, zostavenie programu, či testovanie. *MSBuild* je teda sada open-source nástrojov, ktorá uľahčuje, automatizuje vývoj aplikácií v prostredi .NET. Poskytuje schému vo formáte XML pre projekty a súbory štruktúry *solution*, ktoré určujú, ako sa má program preložiť a zostaviť. Za zmienku stojí aj jeho prepojenie s vývojovým prostredím *Visual Studio*, kde projektové súbory s príponami „.csproj“, „.vbproj“, či ďalšími, obsahujú kód, ktorý sa spustí pri zostavovaní projektu za použitia prostredia *Visual Studio*. Tiež z ich obsahu načíta všetky nastavenia potrebné k fungovaniu projektu.

2.4 Používané nástroje

Napriek tomu, že v súčasnosti neexistuje nástroj na refaktorizáciu .NET kódu za účelom zlúčenia projektov, existujú také, ktoré dokážu meniť ich štruktúru a vykonávať niektoré dielčie úpravy. Nasledujúce nástroje patria k výbave takmer každého vývojára.

2.4.1 Visual Studio

Vývojové prostredie *Visual Studio* [12] nepatrí medzi refaktorizačné nástroje, no jedná sa o vývojové prostredie s najväčou integráciou .NET technológií so širokým záberom možností. Z hľadiska zmeny projektovej štruktúry obsahuje používateľské rozhranie na pridávanie, odstraňovanie projektov, ich referencií, či automatickú inštaláciu knižničných balíčkov. Rovnako automaticky vytvára a upravuje všetky konfiguračné súbory potrebné k chodu aplikácie nad platformou *MSBuild*. V dnešných dňoch je práve *Visual Studio* s určitosťou najpoužívanejšie vývojové prostredie pre .NET technológie.

2.4.2 Resharper

Jedným z najpopulárnejších refaktorizačných nástrojov na trhu pre platformu .NET je nástroj od spoločnosti *JetBrains*, *Resharper* [14]. Funguje ako nástroj integrovaný do prostredia *Visual Studio* a obsahuje prostriedky pre automatické doplňovanie kódu, či formátovanie existujúceho kódu. Z hľadiska minimalizácie grafu závislostí stojí za pozornosť možnosť automatickej úpravy menného priestoru na úrovni štruktúry *solution*, či projektu do formátu, v ktorom by podľa súborového umiestnenia mal byť. Pretože kódový komponent *legacy* obsahuje viacero štruktúr *solution*, ktoré na seba vzájomne odkazujú a ďalšie externé entity, je *Resharper* pre takéto plošné úpravy použiteľný len obmedzene. Nemá informácie o týchto externých entitách a jednoducho sa môže stať, že vykoná neželané úpravy.

Kapitola 3

Redukcia grafu závislostí

V tejto kapitole budú vysvetlené možné stratégie zlučovania grafu závislostí projektov. Na samotné zlúčenie dvoch projektov bude zatiaľ nazerané ako na takzvanú „čiernu skrinku“[1]. Pre získanie efektívnej stratégie je nutné si najprv zadefinovať pojem orientovaný graf. Pretože prekladač jazykov C# a VB.NET nepodporuje cyklické závislosti medzi projektami, je možné sa obmedziť na acyklický orientovaný graf.

3.1 Acyklický orientovaný graf

Orientovaný graf [11] je definovaný ako trojica $G = (V, E, \epsilon)$, kde

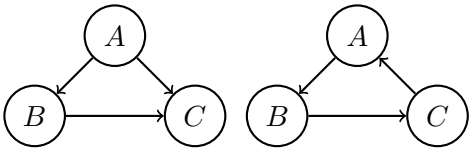
V je neprázdna, konečná množina vrcholov

E je konečná množina orientovaných hrán medzi vrcholmi

ϵ je zobrazenie, takzvaný vzťah incidencie definovaný ako: $E \to V^2$, teda každej hrane z množiny E prislúcha dvojica (X, Y), kde X je počiatočný a Y koncový vrchol

Graf sa označuje ako acyklický, keď neobsahuje žiadne silne súvislé komponenty.

Pod pojmom silne súvislý komponent je možné rozumieť konfiguráciu (V, E, ϵ), kde existujú hrany $(X, Y), (Y, Z), ..(J, K)$ tak, že sa dokážeme dostať orientovanou cestou z X do K a naopak, z K do X [11].



Obr. 3.1: Acyklický verzus cyklický orientovaný graf.

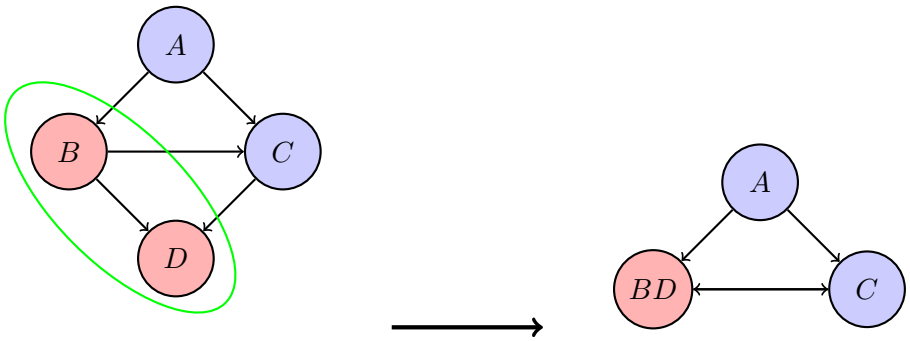
3.2 Problém zlučovania projektov

Vo výslednej aplikácii je žiadané efektívnym spôsobom redukovať počet vrcholov grafu. V prípade, že projekt B v sebe obsahuje referencie projektu A, znamená to, že projekt

[1] Čierna skrinka – Element, ktorého vonkajšie prejavy sú zrejmé, no vnútorná štruktúra je skrytá.

B je závislý na projekte A. To sa v acyklickom orientovanom grafe prejaví ako orientovaná hrana idúca z vrcholu A do vrcholu B. Platí tiež, že je možné zlúčiť projekt A do projektu B a vytvoriť tak nový projekt <AB> len v prípade, že sú obidva projekty rovnakého jazyka. Teda nie je medzi sebou možné zlučovať projekty jazykov VB.NET a C#. Pri ich zlučovaní je takisto potrebné dbať na to, aby sa zlúčením nevytvoril silne súvislý komponent. Takúto situáciu ilustruje obrázok 3.2.

Pre budúce ukážky, nech modrá farba znázorňuje vrcholy projektov jazyka C# a červená farba vrcholy jazyka VB.NET.



Obr. 3.2: Možnosť vytvorenia silne súvislého komponentu.

3.2.1 Formalizácia problému

Predošlé definície umožňujú zapísať problém redukcie grafu formálne:

Nech

$G = (V, E, \epsilon)$ je graf závislostí s dvoma druhmi vrcholov

A je množina všetkých projektov jazyka C#, pričom $A \subseteq V$

B je množina všetkých projektov jazyka VB.NET, pričom $B \subseteq V$

λ je binárna relácia definovaná nasledovne: $a\lambda b \iff \exists\epsilon(e) = (a, b)$, teda existuje orientovaná cesta z bodu a do bodu b, kde $e \in E, a, b \in V$

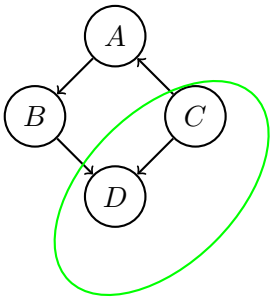
λ^+ je tranzitívny uzáver λ, teda ak platí, že $\exists e_1, e_2 \in E : \epsilon(e_1) = (a, b) \wedge \epsilon(e_2) = (b, c) \implies \exists e_3 \in E : \epsilon(e_3) = (a, c)$ pre $a, b, c \in V$, teda ak existuje orientovaná cesta z bodu a do bodu b, tak potom sa dokážeme z bodu a dostať do ľubovoľného následníka bodu b

Definujeme binárnu operáciu kontrakcie vrcholov $\bigoplus$: $a \bigoplus b \iff$ nahradíme V za $V - \{b\} \wedge \forall e \in E$: nahradíme $\epsilon(e) = (b, x)$ za (a, x) a (x, b) za (x, a), pričom musí platiť, že $((a \in A \wedge b \in A) \vee (a \in B \wedge b \in B)) \wedge \neg\exists c \in V : a\lambda^+ c \wedge c\lambda b$

Nasledujúci diagram s maticou predchodcov grafu ilustruje, akým spôsobom je operácia kontrakcie vrcholov $\bigoplus$ obmedzená na tranzitívny uzáver λ^+

	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	0	0	0
D	0	1	1	0

Obr. 3.3: Matica predchodcov.



Obr. 3.4: Graf závislostí.

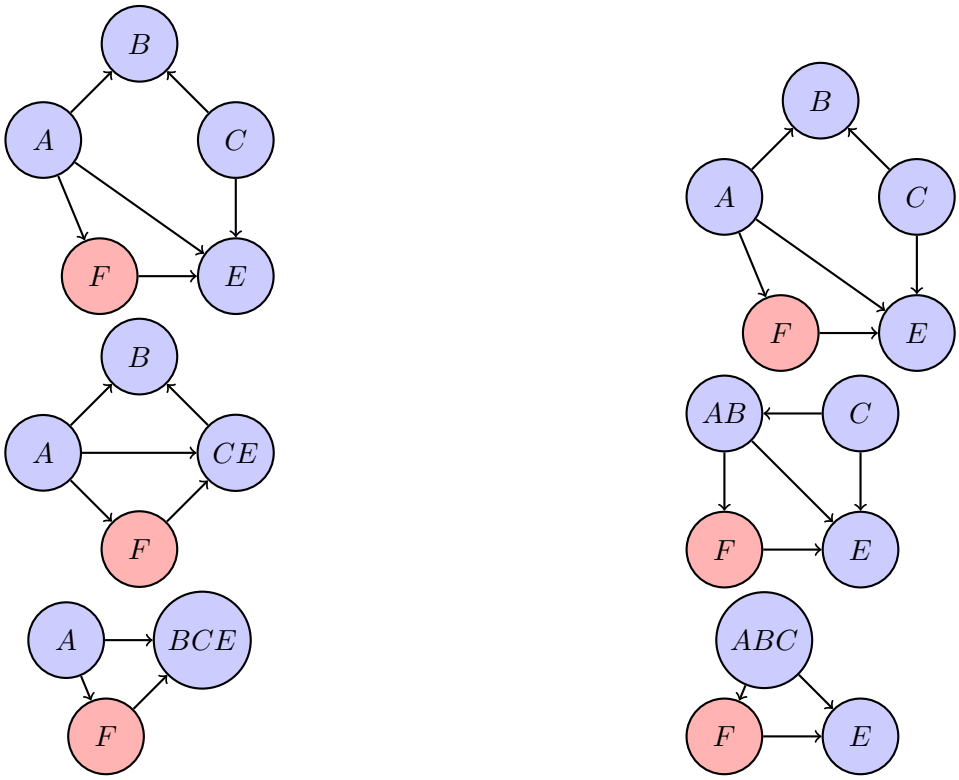
	A	B	C	D
A	0	0	1	0
B	1	0	1	0
C	0	0	0	0
D	1	1	1	0

Obr. 3.5: Tranzitívna matica predchodcov.

V tranzitívnej matici predchodcov je možné si všimnúť, že jeden predchodca vrcholu D má ako tranzitívneho predchodcu vrchol C, čo znamená, že C by sa po kontrakcii stal svojim vlastným predchodcom, preto kontrakcia $C \bigoplus D$ nie je uskutočniteľná.

3.2.2 Problém precedensu kontrakcie vrcholov

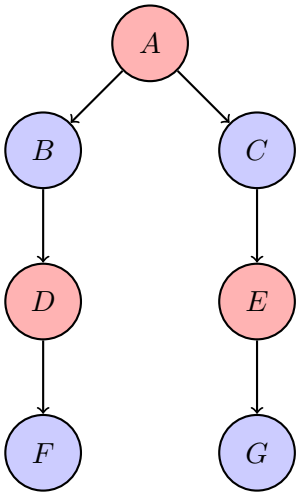
Nasledujúci graf sa môže kontrakciami redukovať na tri vrcholy rôznymi spôsobmi. Môžeme vidieť, že vrchol F je iného typu ako ostatné vrcholy, preto o ňom pre kontrakciu nemá zmysel uvažovať. Je potrebné však kontrolovať, či cez neho môžu vzniknúť silne súvislé komponenty.



Obr. 3.6: Redukcia grafu dvoma rôznymi spôsobmi.

Tento problém by nemal žiadnym spôsobom ovplyvniť počty redukovaných vrcholov pri kontrakcii susedných vrcholov, môže však ovplyvniť množstvo prepisovaných referencií a z hľadiska projektov aj množstvo presunutých súborov, či prepísaného kódu. Napríklad,

ak by do projektu A viedlo ďalších, povedzme 100 hrán od závislých projektov, boli by potrebné oveľa väčšie zásahy do projektovej štruktúry v prípade, kde sa nachádza kontrakcia $A \bigoplus B$. Redukcia grafu s vrcholmi jedného typu je pomerne triviálna záležitosť. Takýto graf by bolo možné redukovať od vrcholu s najnižším, alebo najvyšším poradím v topologicky usporiadanom grafe. Pridanie ďalšieho typu vrcholov spôsobí, že takáto redukcia nie je možná a vrcholy sa dajú redukovať v rôznych kombináciách. Nasledujúci obrázok ilustruje prípad, kedy je možné zlúčiť aj vrcholy, ktoré spolu nesusedia a záleží na poradí zlúčenia:



Obr. 3.7: Kontrakcia nesusediacich vrcholov grafu.

Obrázok vyššie demonštruje, ako pri nesusediacich vrcholoch ovplyvňuje poradie kontrakcie počet možných kontrakcií. V prípade, že by sa previedla kontrakcia vrcholu B a vrcholu G, neboli by možné žiadne ďalšie kontrakcie. Ak by sa však prevádzali kontrakcie postupne, od vrcholov bez následníkov, respektíve vrcholov bez prichádzajúcich hrán, k takejto situácii by nedošlo. Mohli by sa postupne previesť kontrakcie $B \bigoplus C$, $D \bigoplus E$ a $F \bigoplus G$. Je preto nutný usporiadaný prechod grafom.

3.3 Stratégia pre efektívnu redukciu grafu

Pri stratégii redukcie grafu závislostí je možné si tento problém rozdeliť na dva podproblémy nasledujúcim spôsobom:

- Hľadanie vrcholov, o ktorých vzájomnej kontrakcii má zmysel uvažovať.
- Zamedzenie prípadným silne súvislým komponentom po kontrakcii vrcholov, respektíve zlúčení projektov.

3.3.1 Naivný algoritmus

Pre lepšie pochopenie bude najprv tento problém prezentovaný pomocou jednoducho uchopiteľného postupu, bez dôrazu na optimalizáciu. Naivný algoritmus by pozostával z nasledujúcich krokov:

- Postupne vyskúšať zlučovanie všetkých vrcholov grafu medzi sebou.

- Overiť pomocou algoritmu na detekciu silne súvislých komponentov, či by po kontrakcii nevznikla cyklická závislosť.

Pre overenie možnej kontrakcie sa dá použiť na detekciu potenciálne silne súvislého komponentu napríklad Tarjanov algoritmus [19].

Toto riešenie však vôbec nie je optimálne, má veľa redundantných krokov a vysokú časovú zložitosť. $\mathcal{O}(|V|^2 * (|V| + |E|))$ pre kombináciu postupného testovania dvojíc na kontrakciu a Tarjanovho algoritmu pre detekciu silne súvislých komponentov. Tarjanov algoritmus sa spustí pri každej kontrole možnosti kontrakcie dvoch vrcholov. Rovnako je dôležité si uvedomiť, že ak v danom momente nie je možné previesť kontrakciu dvoch vrcholov, kontrakcia tých istých vrcholov môže byť uskutočniteľná neskôr. Preto sa celý proces môže opakovať ešte $|V|$ krát. Ak sa otestovali všetky možné dvojice a našla sa aspoň jedna možná kontrakcia, musí sa začať od znova, aby sa zistilo, či nepribudli nové možnosti pre kontrakciu.

V nasledujúcich podkapitolách budú demonštrované spôsoby a algoritmy, akými je možné proces redukcie grafu zefektívniť.

3.3.2 Topologické usporiadanie

Pre usporiadaný prechod acyklickým orientovaným grafom je potrebné nájsť topologické usporiadanie vrcholov. Prirodzene platí, že vstupný orientovaný graf pre topologické usporiadanie musí byť acyklický. Topologickým usporiadaním grafu $G = (V, E, \epsilon)$ sa rozumie:

- Taká postupnosť všetkých vrcholov grafu G $v_1, v_2, ..., v_n$, že pre akúkoľvek orientovanú hranu z v_i do v_j platí, že $i < j$ [11].

Pre nájdenie topologického usporiadania grafu môže byť použitý Kahnov algoritmus [15]. Intuíciou Kahnovho algoritmu je, že z grafu sa vyberie vrchol, ktorý nemá žiadne prichádzajúce hrany, teda počet jeho prichádzajúcich hrán, *InDegree*, je 0. Po tom, čo tento vrchol vyberieme, počet *InDegree* všetkých jeho následníkov sa dekrementuje o 1. Ak sa takýto vrchol odoberie, zakaždým nastane situácia, že vznikne minimálne jeden nový koreňový vrchol grafu. V opačnom prípade by to znamenalo, že daný graf nie je acyklický.

Algoritmus 1: Kahnov algoritmus topologického usporiadania

```
Input: G - Topologicky usporiadaný graf
Output: L - Usporiadaný zoznam vrcholov
/* Topologicky usporiadaný zoznam vrcholov                    */
L[G.Size];
/* Fronta pre vrcholy s InDegree 0                            */
S = new Queue();
S.Queue(G.GetZeroInDegree());
while L.Size != G.Size do
    zero = S.Dequeue();
    L.Add(zero);
    for child = zero.Children.First ; child != null ; child = child.Next do
        child.Indegree -= 1;
        if child.InDegree == 0 then
            S.Queue(child);
return L;
```

3.3.3 Tranzitívny uzáver

Vzhľadom na fakt, že prekladač spomínaných programovacích jazykov nepovoľuje cyklické závislosti medzi projektami, vstupný graf bude zakaždým acyklický. Táto skutočnosť umožní veľmi efektívne zostaviť maticu susednosti, respektíve maticu dosiahnuteľnosti grafu. Nasledujúci algoritmus ilustruje, ako získať z grafu maticu susednosti, ktorá je reprezentovaná bitovými poliami. Matica susednosti o veľkosti $|V|^2$, kde $|V|$ je kardinalita množiny vrcholov grafu bude mať na každom riadku $i \in \mathbb{N}$ hodnotu 1, ak sa z i-tého vrcholu podľa topologického usporiadania dá dostať v jednom kroku do vrcholu, ktorý je reprezentovaný stĺpcom. V opačnom prípade tam bude hodnota 0.

Algoritmus 2: Algoritmus pre získanie matice susednosti

```
   Input: G - Graf
   Output: M - Matica susednosti veľkosti |V|^2
1  M = 0;
2  for node = G.First ; node != null ; node = node.Next do
3  |  for child = node.Children.First ; child != null ; child = child.Next do
   |  |  /* V matici susednosti budú všetky synovské vrcholy                */
4  |  |  MR[node.index] |= (1 « child.index);
5  return M;
```

Získanie matice susednosti má časovú zložitosť $\mathcal{O}(|V|+|E|)$, kde V je množina vrcholov a E je množina hrán. V prezentovanom algoritme sa pristupuje na každý vrchol v jednom cykle *for*, nazvime ho pracovný vrchol a následne sa vykoná bitová operácia so všetkými jeho potomkami, teda vrcholmi do ktorých sa z aktuálneho pracovného vrcholu je možné dostať. Aktualizácia matice susednosti pozostáva z pridania potomkov odstraňovaného vrcholu a výmenou potomka u jeho rodičov o rozširovaný vrchol. Zo získanej matice sa veľmi jednoducho získa maticu dosiahnuteľnosti.

Matica dosiahnuteľnosti o veľkosti $|V|^2$ bude mať na každom riadku $i \in \mathbb{N}$ hodnotu 1, ak sa z i–tého vrcholu podľa topologického usporiadania je možné dostať v ľubovoľnom počte krokov do vrcholu, ktorý je reprezentovaný stĺpcom. V opačnom prípade bude na danom riadku hodnota 0.

Matica dosiahnuteľnosti je tranzitívnym uzáverom grafu, ktorý bol definovaný pri formalizácii problému redukcie grafu 3.2.1.

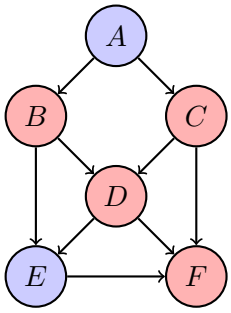
Algoritmus 3: Algoritmus pre získanie matice dosiahnuteľnosti

```
   Input: G - Topologicky usporiadaný graf, M - Matica susednosti
   Output: MR - Matica dosiahnuteľnosti veľkosti |V|^2
1  MR = M;
   /* Ulož do premennej node posledný vrchol podľa topologického usporiadania */
2  for node = G.GetLast ; node != null ; node = node.Previous do
3  |  for child = node.Children.First ; child != null ; child = child.Next do
4  |  |  index = child.index;
   |  |  /* V matici dosiahnuteľnosti budú všetky synovské vrcholy a vrcholy z nich
   |  |     dosiahnuteľné                                                    */
5  |  |  MR[node.index] |= (1 « index) | MR[index];
6  return MR;
```

Z predošlého algoritmu vyplýva, že časová zložitosť získania tejto matice je rovnaká ako v prípade matice susednosti, teda $\mathcal{O}(|V| + |E)$ s jednou bitovou operáciou *or* naviac. Pre úpravu matice po kontrakcii dvoch vrcholov je potrebné upraviť dosiahnuteľnosť všetkým predchodcom odstraňovaného a rozširovaného vrcholu. To je možné vykonať prechodom tou istou maticou, pomocou jednej bitovej operácie zistiť, či bol z vrcholu dosiahnuteľný odstraňovaný vrchol a logickou operáciou *or* daný riadok upraviť.

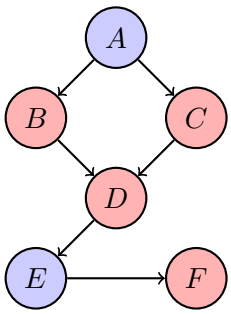
3.3.4 Tranzitívna redukcia

Z grafu 3.8 je na prvý pohľad zrejmé, že niektoré vrcholy nie je možné zlúčiť. Napríklad C a F síce sú rovnakého typu (programovacieho jazyka), no ich kontrakciou by vznikol silne súvislý komponent.



Obr. 3.8: Graf pred tranzitívnou redukciou.

Jedným z nástrojov, ktorým je možné znížiť množstvo testov na vznik silne súvislého komponentu je tranzitívna redukcia. Tranzitívna redukcia grafu G je minimálny podgraf grafu G, označíme ako H, ktorý má rovnaký tranzitívny uzáver ako graf G. Podgraf H grafu G vznikne z grafu G vynechaním niektorých vrcholov a hrán [11].



Obr. 3.9: Graf po tranzitívnej redukcii.

Na výpočet tranzitívnej redukcie sa použije už získaná matica dosiahnuteľnosti.

Algoritmus 4: Algoritmus pre získanie matice tranzitívnej redukcie.

Input: G - Topologicky usporiadaný graf, MR - Matica dosiahnuteľnosti
Output: MTR - Matica tranzitívnej redukcie o veľkosti $|V|^2$

```
   /* Ulož do premennej Node posledný vrchol podľa top. usporiadania                */
1  for node = G.Getlast ; Node != First ; Node = Node.Previous do
2      MTemp[V] = 0;
3      for child = node.Children.First ; child != null ; child = child.Next do
           /* V MTemp majú byť len potomkovia vrcholu child, nie on samotný          */
4          MTemp |= MR[child.index];
5      for child = node.Children.First ; child != null ; child = child.Next do
           /* Ak nie je možné dostať sa do vrcholu child z nejakého iného synovského
              vrcholu, potom patrí child do matice tranzitívnej redukcie             */
6          if !(MTemp & (1 « child.index)) then
7              MTR[node.index] |= (1 « child.index)
8      return MTR;
```

Je zrejmé, že výpočet matice tranzitívnej redukcie bude mať rovnakú časovú zložitosť ako výpočet matice dosiahnuteľnosti. Rozdielom je len, že sa na každú hranu pristúpi dvakrát. Výsledná časová zložitosť je teda $\mathcal{O}(|V|+|E|)$. Tranzitívna redukcia dokáže identifikovať všetky kontrakcie susedných vrcholov. Ako bolo zadefinované pri formalizácii problému 3.2.1, jediný spôsob, akým môže pri kontrakcii dvoch vrcholov vniknúť silne súvislý komponent je, že pri zlúčení vrcholu A s vrcholom B existuje synovský vrchol vrcholu A, z ktorého sa pomocou dopredných hrán je možné dostať do vrcholu B. Ak je možné dostať sa z vrcholu do synovského vrcholu B z nejakého iného synovského vrcholu C, v matici tranzitívnej redukcie sa vrchol B nebude vyskytovať. Tranzitívna redukcia preto dokáže identifikovať všetky susedné vrcholy vhodné na kontrakciu.

Aktualizácia matice tranzitívnej redukcie po kontrakcii vrcholov spočíva v úprave riadkov matice všetkých predchodcov odstraňovaného vrcholu a úprave rozširovaného vrcholu. Úprava prebieha analogicky s prezentovaným algoritmom pre výpočet tranzitívnej redukcie s rozdielom, že sa v úvodnom cykle *For* nepristupuje na každý vrchol, len na predchodcov odstraňovaného vrcholu a rozširovaný vrchol.

3.3.5 Možnosti kontrakcie nesusediacich vrcholov grafu

Po tom, čo je prevedená maximálna možná redukcia susedných vrcholov sa stále môže stať, že na kontrakciu budú dostupné aj vrcholy, ktoré medzi sebou nemajú hranu. V prípade týchto vrcholov je možné použiť maticu dosiahnuteľnosti. Ak vrcholy medzi sebou nemajú orientovanú cestu, znamená to, že ich kontrakciou určite nevznikne silne súvislý komponent. Postupom pre záverečnú fázu je vzostupne podľa topologického usporiadania pre každý vrchol pomocou matice dosiahnuteľnosti nájsť vrchol, do ktorého nevedie orientovaná cesta a tieto dva vrcholy zlúčiť. Je teda nutné prejsť každý riadok matice dosiahnuteľnosti až kým nebudú vyčerpané všetky kontrakcie. Ak je na riadku matice viacero nedosiahnuteľných vrcholov rovnakého druhu, je nutné vybrať vrchol s najnižším ohodnotením podľa topologického usporiadania a po zlúčení nájsť ďalší pár vrcholov. V opačnom prípade by to mohlo negatívne ovplyvniť počet redukovaných vrcholov, ako bolo demonštrované na obrázku 3.7. Pred týmto krokom je opätovne nutné topologicky usporiadať graf. Po záverečnej fáze sa graf dostane do stavu, kedy už nie je možné previesť kontrakciu ďalších vrcholov.

3.3.6 Porovnanie algoritmov

Pri porovnaní naivného algoritmu s optimalizovaným algoritmom kombinujúcim vyššie uvedené, sa dá pozorovať zlepšenie oproti $\mathcal{O}(|V|^2 * (|V| + |E|) * |V|)$. V časti pre redukciu susedných vrcholov bude mať algoritmus časovú zložitosť $\mathcal{O}(|V| + |E|)$ pre výpočet všetkých matíc a pre výpočet topologického usporiadania grafu. Aktualizácia jednotlivých matíc bude prebiehať len počas kontrakcie, ktorých môže byť najviac $|V|$, preto časová zložitosť kontrakcie vrcholov bude $\mathcal{O}(|V|^3)$). Aktualizácia matice dosiahnuteľnosti, susednosti má lineárnu časovú zložitosť a matice tranzitívnej redukcie kvadratickú časovú zložitosť. Je nutné poznamenať, že časová zložitosť bitových operácií je v rámci tejto bakalárskej práce považovaná za konštantnú. Posledná časť redukcie grafu, zlučovanie nesusediacich vrcholov má časovú zložitosť $\mathcal{O}(|V| + |E|)$ pre topologické usporiadanie grafu a $\mathcal{O}(|V|^2)$ pre prechod maticou dosiahnuteľnosti. Pretože graf závislostí projektov je vzhľadom na počet hrán tzv. *riedky* graf [11], v skutočnosti bude algoritmus pomerne rýchly. Napríklad priemerne bude aktualizácia matice tranzitívnej redukcie trvať približne $|V|$ krokov.

3.3.7 Výber poradia kontrakcií vrcholov grafu

Je zrejmé, že v záverečnej fáze pri kontrakcii nesusediacich vrcholov je nutné prechádzať grafom usporiadane, podľa topologického usporiadania. Pri susedných vrcholoch sa však naskytá výber z mnohých možností. Zlučovať postupne od listových, či koreňových vrcholov, alebo napríklad podľa vrcholov s najnižším počtom závislostí. Voľba možnosti pre výslednú aplikáciu bude prezentovaná pri popise testovania časti aplikácie pre analýzu grafu 7.0.2.

Kapitola 4

Zlučovanie .NET projektov

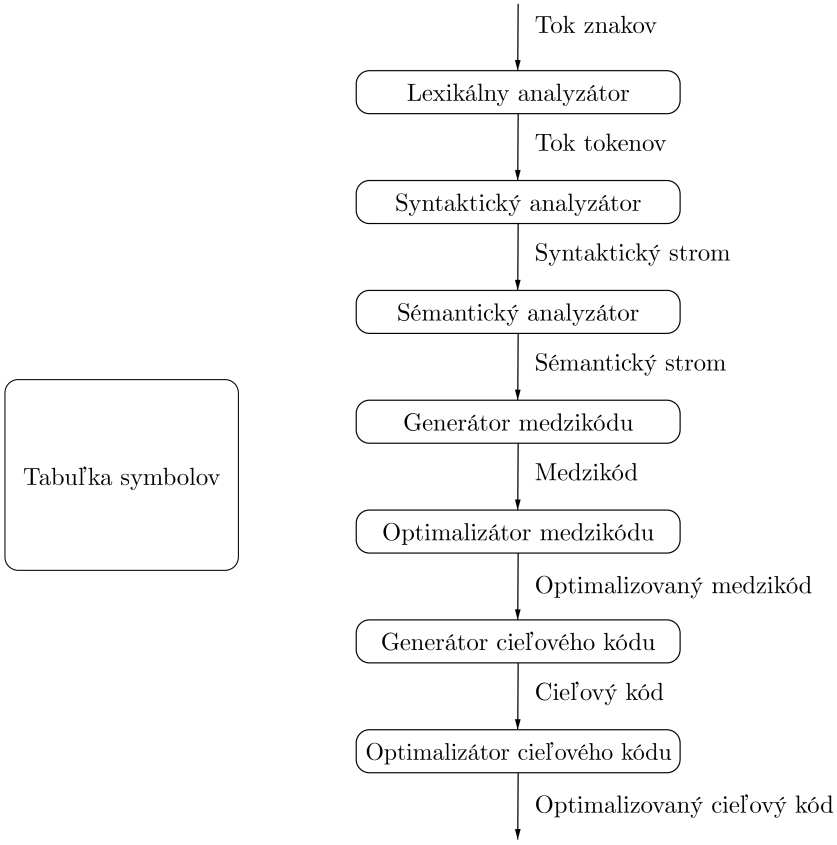
V tejto kapitole sa na operáciu zlučovania projektov prestane nazerať ako na pomyselnú čiernu skrinku. Budú tu vysvetlené princípy a nástroje, ktorými sa redukcia grafu závislostí docieli z hľadiska manipulácie kódu. Pretože v súčasnosti neexistuje nástroj, ktorý by dokázal previesť potrebnú redukciu počtu projektov, musia byť identifikované pomocné prostriedky na automatizáciu procesu.

4.1 Princíp činnosti prekladača

Na pochopenie, ako je potrebné pri minimalizácii grafu závislostí meniť štruktúru konfiguračných a kódových .NET súborov je nutné vedieť, ako prekladače programovacích jazykov fungujú. Činnosť prekladača od načítavania zdrojového textu až po vytvorenie cieľového kódu sa dá zadefinovať do dvoch logických kategórií [8]:

- **Analytická časť** pozostáva z lexikálnej, syntaktickej a sémantickej analýzy, ktorej výsledkom je štruktúra *abstraktný syntaktický strom.* Ten je následne použitý na generovanie takzvaného medzikódu, ktorý je v poslednom kroku analytickej časti optimalizovaný. Hlavnou úlohou analytickej časti je teda štrukturálne usporiadať vstupný kód a odhaliť prípadné syntaktické, či sémantické chyby. Spoločne s medzikódom vstupuje do syntetickej časti aj takzvaná tabuľka symbolov, v ktorej sú uložené informácie o jednotlivých častiach programu.
- **Syntetická časť** zjednocuje výstup analytickej časti a konštruuje výsledný program. V prípade programovacích jazykov C# a VB.NET sa jedná konkrétne o spustiteľný súbor s príponou „.exe“ alebo knižničný modul s príponou „.dll“.

Celá táto podkapitola sa opiera o informácie z knihy *Compilers, Principles, Techniques, and Tools* [8].



Obr. 4.1: Fázy prekladu programu [8].

Z hľadiska vyvíjaného nástroja na minimalizáciu závislostí sú najdôležitejšími časťami syntaktická a sémantická analýza.

4.1.1 Syntaktická analýza

Po lexikálnej analýze, načítaní vstupného textového toku do *lexémov*, s ich následným pretransformovaním do takzvaných *tokenov* začína syntaktická analýza [8]. *Token* reprezentuje dvojica, a to názov *tokenu* a voliteľná hodnotu jeho atribútu. V prípade, že sa *lexém* zhoduje s názvom *tokenu*, je tento *lexém* identifikovaný ako inštancia daného *tokenu*. Výsledkom syntaktickej analýzy je štruktúra s názvom abstraktný syntaktický strom, zjednodušene syntaktický strom. Syntaktický strom takisto definuje poradie jednotlivých operácií programu.

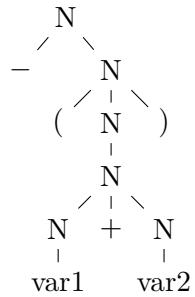
Cieľom syntaktickej analýzy je overiť správnosť poradia *lexémov*, teda či ich sekvencia je generovaná *gramatikou daného jazyka*. Z toho vyplývajú aj syntaktické chyby, ktoré táto časť prekladu dokáže zachytiť, ako napríklad chýbajúce svorkové zátvorky, alebo chýbajúce operandy. Na definovanie, ako spolu súvisí stavba syntaktického stromu a gramatika používaného jazyka pomôže formálna definícia *bezkontextovej gramatiky* (ďalej BKG). *BKG* [21] G tvorí štvorica $G = (N, \Sigma, P, S)$, kde:

- N je konečná množina neterminálnych symbolov, neterminálov.
- Σ je konečná množina terminálnych symbolov, terminálov.

- P je konečná množina prepisovacích pravidiel v tvare $A \to \alpha, A \in N$ a $\alpha \in (N \cup \Sigma)^*$.
- $S \in N$ je počiatočný symbol gramatiky.

Derivačný strom

Derivačný strom (ďalej DS) je stromovou reprezentáciou derivácie gramatiky jazyka a zobrazuje poradie, v ktorom sa neterminály postupne nahrádzajú inými terminálmi a neterminálmi. V listových uzloch DS sa nachádzajú iba terminálne symboly gramatiky. Inými slovami, DS zobrazuje **deriváciu** gramatiky jazyka. Deriváciou sa rozumie postupnosť aplikácie prepisovacích pravidiel, ktorých výsledkom je postupnosť terminálov, listových uzlov DS.



Obr. 4.2: Ukážka derivačného stromu pre výraz −(var1 + var2) [8].

Dá sa povedať, že syntaktický strom je zjednodušená verzia derivačného stromu a neobsahuje informácie napríklad o zátvorkách, alebo medzerách. Reprezentuje len hierarchickú štruktúru vstupného programu, bez ďalších informácií.

4.1.2 Sémantická analýza

Syntaktická analýza produkuje syntaktický strom bez informácií, či pri vytvorení inštancie triedy bola daná trieda deklarovaná, alebo či sú napríklad operandy operácie konkatenácie reťazcov správneho typu. Na získanie kontextu o spomenutých, sémantických informáciách, slúži sémantická analýza ako krok nadväzujúci na syntaktickú analýzu. Používa syntaktický strom, respektíve tabuľku symbolov na zbieranie informácii o zdrojovom programe, typoch a overuje sémantickú správnosť kódu. Tieto získané informácie potom zapíše do tabuľky symbolov, alebo syntaktického stromu [8].

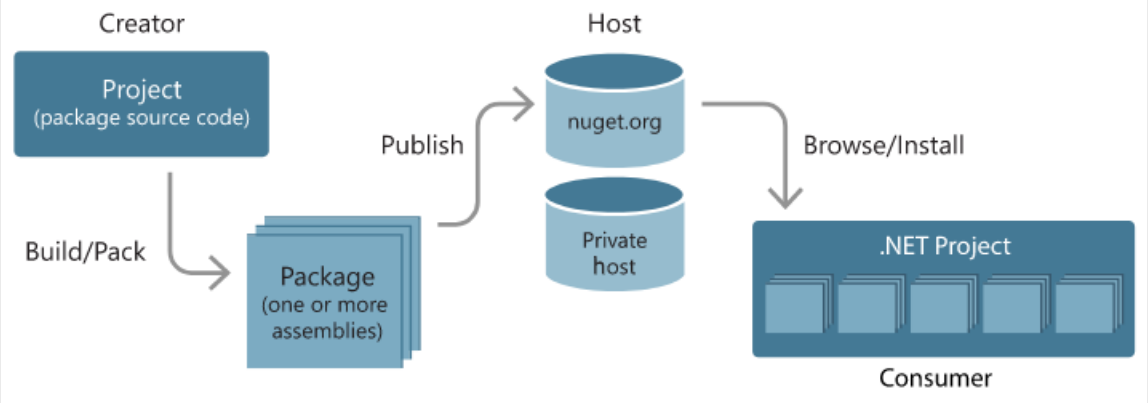
4.2 Nástroje

V tejto kapitole budú prezentované nástroje a knižnice, potrebné pre automatizovanú manipuláciu projektov, ich súčastí, alebo prepis syntaxe jazykov.

4.2.1 NuGet

Nuget je nástroj, ktorý umožňuje vytváranie, zdieľanie a používanie užitočného kódu pre platformu .NET. Takýto kód zapuzdruje do knižníc, v prostredí Visual Studio takzvaných balíčkov. Nuget balíček je v podstate zapuzdrená knižnica pre programovací jazyk s príponou „.nupkg“. Balíčky je možné publikovať na verejný server, základný je nuget.org, alebo

na privátny, spravovaný súkromnou entitou. Prostredie Visual Studio ponúka používateľské rozhranie pre správu Nuget balíčkov, ktoré sa pridávajú k jednotlivým projektom ako závislosti. Tieto závislosti je pri zlučovaní projektov takisto potrebné vyriešiť. V prípade pridania balíčka k projektu sa balíček automaticky stiahne zo servera Nuget [4].



Obr. 4.3: Fungovanie Nuget balíčkov [4].

4.2.2 Roslyn

Platforma Roslyn je sada prekladačov a API rozhraní pre programovacie jazyky VB.NET a C#, ktorá zjednodušuje a automatizuje statickú analýzu kódu, refaktorizáciu, či prepis syntaxe jazykov. Celá platforma je napísaná v programovacom jazyku C#. Roslyn rozdeľuje tri fázy prekladu programu a poskytuje pre ne rozhranie [9]:

- **Syntaktická analýza** – Hierarchické ukladanie kódu do syntaktického stromu, ktorého výstupom je syntaktický strom vo formáte DOM (Document Object Model).
- **Sémantická analýza** – Mapovanie identifikátorov na symboly, s nazbieranými informáciami o jednotlivých symboloch.
- **Zostavenie** – Proces zostavovania výsledného programu, ktorého produktom je už spomínaný súbor zostavenia.

Je dôležité dodať, že všetky dátové štruktúry na platforme Roslyn sú založené na princípe nemennosti. Nemennosť (Immutability) v programovacích jazykoch znamená, že objekt nie je možné modifikovať. Jedinou možnosťou, ako daný objekt zmeniť je vytvoriť jeho novú inštanciu.

Syntaktický model

Keďže Roslyn obsahuje API prekladača programovacích jazykov VB.NET a C#, jednou z jeho súčastí je aj *Syntax API*. Toto rozhranie definuje a poskytuje prístup k dátovým štruktúram, z ktorých pozostáva kód jazyka C#, respektíve VB.NET. [1].

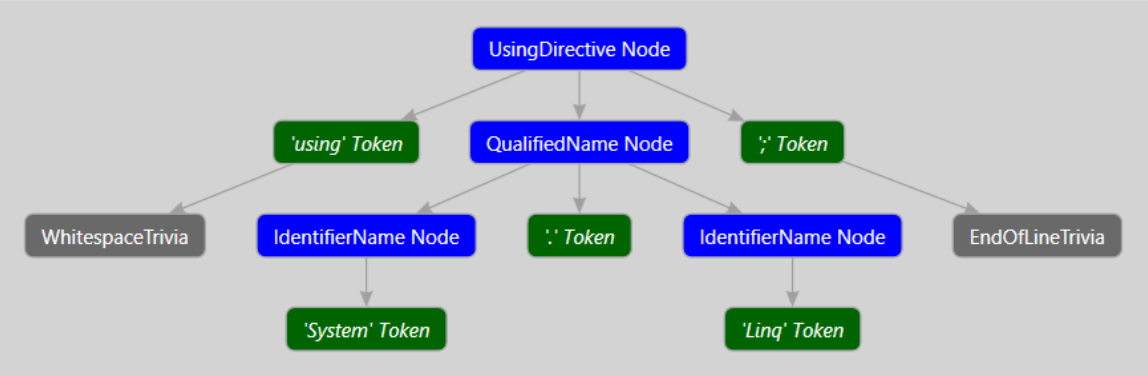
Syntaktický strom

Ako bolo spomenuté v podkapitole o syntaktickej analýze 4.1.1, syntaktický strom je typicky výsledkom syntaktickej analýzy a usporadúva vstupné tokeny do hierarchickej štruktúry.

Z hľadiska Syntax API sa jedná sa o základnú štruktúru pre preklad, analýzu, refaktorizáciu, či generovanie kódu. Syntaktický strom pre jednotlivé programovacie jazyky definujú na platforme Roslyn triedy *CSharpSyntaxTree* a *VisualBasicSyntaxTree*. Samotná štruktúra triedy *SyntaxTree* však skôr pripomína derivačný strom, pretože obsahuje o vstupnom kóde všetky informácie tak, aby sa dal zo štruktúry *SyntaxTree* generovať identický kód aj späť. Nad triedou *SyntaxTree* sú ďalej definované triedy a metódy, ktoré výrazne uľahčujú prácu s ňou. Jedná sa napríklad o *SyntaxFactory* na vytváranie nových uzlov syntaktického stromu, alebo *SyntaxWalker*, respektíve *SyntaxRewriter*, ktoré zasa zjednodušujú prechod, vyhľadávanie v syntaktickom strome, či jeho prepisovanie. Trieda SyntaxTree na platforme Roslyn pozostáva z nasledujúcich častí [3]:

- **Syntax node** – Reprezentuje uzol stromu, ktorý obsahuje neterminálne symboly gramatiky jazyka. Jedná sa napríklad o deklarácie, klauzuly alebo výrazy. Obsahuje informácie o svojich predchodcoch, potomkoch, či voliteľných potomkoch, napríklad klauzula *Else* pri podmienke *If*.
- **Syntax token** – Reprezentuje uzol stromu, ktorý obsahuje terminálne symboly gramatiky jazyka, akými sú premenné, či kľúčové slová. Znamená to, že tieto uzly nemajú žiadne synovské uzly. Informácia o terminálnych uzloch je definovaná v triede *SyntaxToken*.
- **Syntax trivia** – Obsahuje časti zdrojových súborov, ktoré pre fungovanie programu nie sú dôležité, napríklad komentáre, alebo biele znaky. Jej reprezentácia je v triede *SyntaxTrivia* a viaže sa na konkrétny uzol stromu, podľa pozície v zdrojovom kóde.
- **Syntax span** – V rámci štruktúry *TextSpan*, majú všetky spomenuté súčasti syntaktického stromu uložené údaje o počte znakov, pozíciách, či bielych znakoch.
- **Syntax kind** – Jednoznačne identifikuje každý element v syntaktickom strome. Je definovaný vymenovaným typom *SyntaxKind*.
- **Errors** – Obsahuje syntaktické chyby odhalené pri statickej analýze prekladačom. Nie sú obsiahnuté vo vlastnej triede, no definujú atribúty jednotlivých uzlov stromu, ako napríklad *IsMissing* v prípade chýbajúcej časti, alebo *SkippedTokensTrivia*, kam sa ukladajú nerozpoznané lexémy.

Nasledujúci obrázok zobrazuje reprezentuje skladbu syntaktického stromu pre riadok kódu „*using System.Linq;*“, teda direktívy použitia:



Obr. 4.4: Vizualizácia uzlu triedy *SyntaxTree* v prostredí Visual Studio 2022.

Syntax Walker

Syntax Walker je abstraktná trieda z knižnice *CodeAnalysis*, ktorá slúži na navigáciu syntaktickým stromom pomocou návrhového vzoru *Visitor*[1]. Umožňuje vytvoriť vlastnú triedu na ľubovoľnú analýzu jednotlivých častí stromu. Pri implementácii triedy *Syntax Walker* je potrebné vytvoriť triedu, ktorá dedí od jeho abstraktnej triedy a implementovať niektoré zo stoviek jej virtuálnych metód. Každá z týchto metód slúži na navštívenie konkrétneho druhu uzlu v syntaktickom strome. V prípade, že sa zavolá metóda triedy *Syntax Walker*, *Visit*, so syntaktickým stromom, alebo podstromom kódového súboru, do všetkých prepísaných metód bázovej triedy sa dostane len jeden konkrétny uzol stromu, na ktorý je daná metóda určená. Prechod stromom je preto výrazne uľahčený. Bázová trieda vytvorí inštanciu potrebného uzlu aj s odkazom na všetkých predchodcov a potomkov. Jedná sa o automatický prechod stromom pomocou prehľadávania do hĺbky so zaznamenávaním potrebných údajov [9].

```
public class NamespaceWalker : VisualBasicSyntaxWalker
{
        public int Count;

        public NamespaceWalker()
        {
                Count = 0;
        }

        public override void VisitNamespaceStatement(SyntaxNode node)
        {
                Count++;
                // Navštív aj synovské uzly
                base.VisitNamespaceStatement(node);
        }
}
...
var walker = new NamespaceWalker();
walker.Visit(rootNode);
if (walker.Count > 0) ... Do something
```

Výpis 4.1: Ukážka použitia triedy *SyntaxWalker*.

Syntax Rewriter

Ďalšou dôležitou súčasťou na prácu so syntaktickým stromom na platforme *Roslyn* je *Syntax Rewriter*. Rovnako, ako pri triede *Syntax Walker* aj tu sa jedná o abstraktnú triedu, tentokrát však určenú na manipuláciu so syntaktickým stromom. Funguje obdobne, používa návrhový vzor *Visitor*, dovoľuje implementovať virtuálne metódy a pomocou nich selektívne prehľadávať jednotlivé časti syntaxe. *SyntaxRewriter* umožňuje pridávať, meniť, odoberať uzly, alebo vytvárať úplne nové stromy. Je dôležité si pripomenúť, že celá platforma *Roslyn* je založená na princípe nemennosti, preto akúkoľvek zmenu, ktorú *Syntax Rewriter* vykoná

[1]Návrhový vzor Visitor dovoľuje rozšíriť, upraviť funkcionalitu objektu bez nutnosti zmien triedy.

je nutné aplikovať vytvorením nového dokumentu s novým syntaktickým stromom. Na vytváranie úplne nových uzlov tu slúži statická trieda *SyntaxFactory*, ktorá obsahuje metódy na vytvorenie konkrétnych uzlov [9]. Použitie triedy *SyntaxRewriter* je analogické s ukážkou triedy *SyntaxWalker* 4.1.

Sémantický model

Doposiaľ bola reč stále o syntaktickej časti platformy Roslyn, kde bolo povedané, ako pracovať s triedou *SyntaxTree*. Na získanie informácií o jednotlivých metódach, triedach, či premenných je však potrebné presunúť sa do sémantickej vrstvy, kde je hneď niekoľko spôsobov, ako tieto informácie získať. Jedným z hlavných pojmov v sémantickom modeli platformy Roslyn je symbol.

Ako udáva kniha *C# 6.0 in a Nutshell* [9], sémantický model dokáže transformovať identifikátory na symboly a tie už obsahujú informácie o type, či sa jedná o názov metódy, alebo napríklad atribút triedy. Identifikátor *Count* môže znamenať v určitom kontexte názov metódy, aj názov premennej. Pri preklade sa táto informácia získa pri procese, ktorý sa nazýva *binding*[2]. Okrem toho nesie informácie napríklad o tom, v akom mennom priestore sa nachádza, aký má modifikátor prístupu[3], alebo za pomoci triedy *SymbolFinder* aj o tom, kde na iných miestach v kóde sa v rámci celej štruktúry *solution* používa.

Jedným zo spôsobov, ako sa k symbolom za pomoci API prekladačov dostať, je zavolať na konkrétny projekt, alebo dokument metódu *Compilation*, ktorá invokuje prekladač a z jej výsledku sa následné dá získať inštancia triedy *SemanticModel*. *SemanticModel* si uloží dáta o symboloch do vyrovnávacej pamäte, preto je efektívne pracovať s jednou inštanciou po čo najdlhšiu dobu.

Nasledujúca ukážka ilustruje použitie triedy *SemanticModel*. V ukážke je použitá jej inštancia s metódou *GetDiagnostics*, ktorá získa informácie o syntaktických a sémantických chybách v poskytnutom syntaktickom strome z konkrétneho dokumentu. Umožňuje danú chybu lokalizovať a zistiť konkrétny typ chyby:

```
var project = _solution.GetProject(projectId);
var compilation = project.GetCompilationAsync().Result;

foreach (var syntaxTree in compilation.SyntaxTrees)
{
       var semanticModel = compilation.GetSemanticModel(syntaxTree);
       var modelDiagnostics = semanticModel.GetDiagnostics()
}
```

Výpis 4.2: Ukážka práce s triedou *SemanticModel*

Workspace

Keďže všetko na platforme Roslyn je založené na princípe nemennosti, všetky zmeny znamenajú vytvorenie úplne nového objektu v pamäti, vďaka čomu môžu omnoho jednoduchšie prebiehať paralelne. Štartovacím bodom akejkoľvek väčšej zmeny je vrstva *Workspace*, ktorá

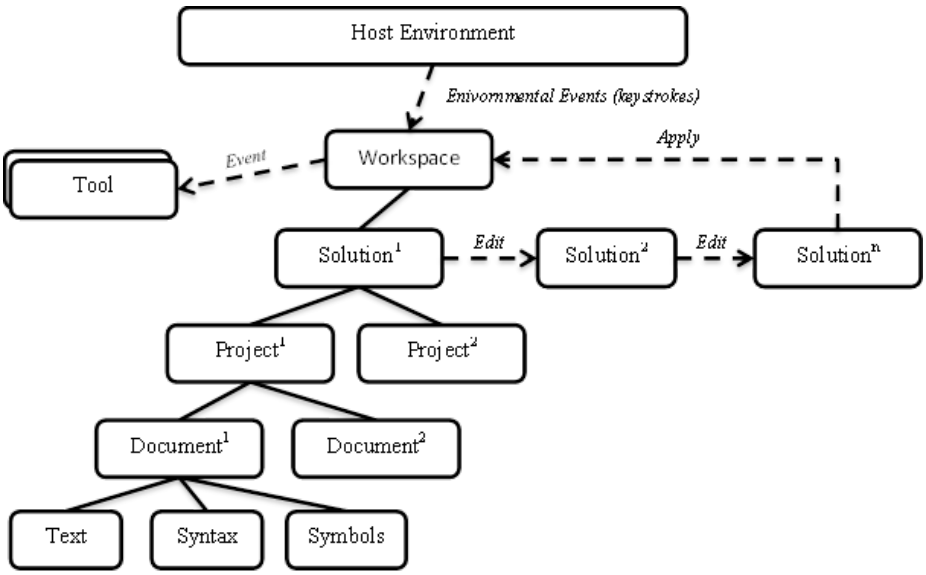
[2]Binding je proces, kedy sa pri preklade identifikátor naviaže na konkrétny objekt.

[3]Modifikátor prístupu je kľúčové slovo, ktoré sa používa pri deklarácii objektov a určuje ich dostupnosť voči ostatným objektom.

poskytuje abstrakciu nad štruktúrou *solution*. *Workspace* API zabezpečuje prístup k štruktúre *solution*, projektom, zdrojovým súborom, syntaktickej, sémantickej analýze a celému procesu prekladu programu. V rámci API je *Workspace* len abstraktná bázová trieda a pre použitie v určitom kontexte je potrebná jej implementácia. Odvodená trieda dedí od bázovej triedy a pre požadovanú funkcionalitu prepisuje niektoré prázdne metódy podľa potreby. V prípade zásahov do implementácie vrstvy *Workspace*, či už na úrovni zmien v zdrojových súboroch, alebo v úprave projektových závislostí je nutné všetky používané objekty v kóde prepísať novými. Napríklad, keď sa modifikuje zdrojový súbor, dokument, jeho nová inštancia obsahuje odkaz na nový projekt a ten na novú štruktúru *solution* [2].

Existujú tri základné, najčastejšie používané implementácie bázy *Workspace* [10]:

- **MSBuildWorkspace** jedná sa o Workspace *implementáciu* využívanú vo výslednej aplikácii. Pracuje v kontexte pre platformu MSBuild, dokáže manipulovať s projektami a jednotlivými dokumentami, nie však so štruktúrou *solution*, pretože nedokáže zapisovať do súborov s príponou ".sln".
- **AdHocWorkspace** Táto implementácia sa používa takmer výhradne na testovacie účely, pretože nedokáže perzistovať zmeny. Všetky operácie prebiehajú vo virtuálnej pamäti, dokáže však manipulovať projekty a dokumenty rovnako ako *MSBuildWorkspace*.
- **VisualStudioWorkspace** Ako napovedá názov, táto implementácia je vytvorená pre vývojové prostredie Visual Studio. Používa sa pri vytváraní nástrojov, ktoré analyzujú kód priamo vo vývojovom prostredí.



Obr. 4.5: Diagram štruktúry Workspace [2].

4.3 Nezlučiteľné projekty

V prípade, že je dosiahnutý najvyšší možný počet redukcií projektov programovacích jazykov C# a VB.NET a je žiadúce pokračovať v redukcii grafu závislostí, zostávajú 3 možnosti.

- Ručne prepísať do iného programovacieho jazyka časti kódu, kde jeden projekt používa komponenty druhého projektu. To je prevediteľné len v prípadoch, kedy medzi projektami nie je veľa závislých častí kódu.
- V prípade, že by kontrakciou vrcholov vznikla cyklická závislosť, je možné rozdeliť problémový projekt na minimálne dva ďalšie a zlúčiť tú časť, ktorá nespôsobí silne súvislý komponent. Tento variant by si však vyžadoval veľmi rozsiahlu syntaktickú a sémantickú analýzu, navyše by neriešil problém ďalšej redukcie grafu. Počet vrcholov grafu by sa nezmenil.
- Poslednou možnosťou je transformácia syntaxe medzi jazykmi. Na transformáciu syntaxe medzi jazykmi existuje niekoľko takzvaných open-source, či komerčných nástrojov, ktoré by bolo na tento účel možné využiť.

4.3.1 Konverzie syntaxe medzi jazykmi

V tejto sekcii je prehľad existujúcich nástrojov pre konverziu syntaxe medzi programovacími jazykmi a nakoniec sú ilustrované základné princípy na akých by mohol vzniknúť vlastný nástroj. Na základe získaných informácií o predošlých snahách transformovať syntax *legacy* kódu bolo zistené, že v minulosti sa objavili pokusy o transformáciu častí kódu pomocou nástroja **Code Converter** 4.3.2. Konverzia kódu však nebola úplne spoľahlivá a objavovali sa syntaktické a sémantické chyby, ktoré si vyžadovali manuálne úpravy, ale aj horšie zachytávané chyby za behu programu. V súčasnosti tak neexistuje nástroj, ktorý by dokázal spoľahlivo pretransformovať státisíce riadkov *legacy* kódu z jazyka C# do jazyka VB.NET. To je hlavný dôvod, prečo bude prezentovaný aj spôsob, akým by mohol vzniknúť vlastný nástroj. Mohol by byť navrhnutý na mieru a obmedziť sa len na konštrukty, ktoré sa používajú v *legacy* kóde.

Na mieste je otázka, či by konverzia syntaxe medzi jazykmi skutočne mala slúžiť na pokračovanie v redukcii projektov. V prípade, že graf závislostí sa upraví do stavu, kedy bez konverzie nie je možné redukovať ďalšie uzly, znamená to, že zostávajúce projekty sú omnoho robustnejšie. Vhodnejšie je postupne transformovať syntax jazykov pred redukciou závislostí, aby sa prípadné chyby spôsobené transformáciou jednoduchšie lokalizovali a odstránili.

4.3.2 Komerčné a open-source nástroje

Ako bolo spomenuté, na trhu dnes existuje niekoľko nástrojov pre konverziu medzi programovacími jazykmi C# a VB.NET.

SharpDevelop Code Converter

Code Converter [13] je open-source nástroj ktorý umožňuje ako konverziu jazyka C# na VB.NET, tak aj opačne. Avšak opačne je táto konverzia oveľa menej spoľahlivá a odporúča sa používať skôr na malé útržky kódu a na edukatívne účely, ako uvádza dokumentácia nástroja. Obsahuje rozšírenia, ktoré pomáhajú upraviť kód pre efektívnejšiu konverziu, alebo zlepšenie jeho kvality po konverzii. Vopred sa počíta s tým, že niektoré konštrukcie, ktoré nástroj vytvorí budú prekomplikované a budú vytvárať nadbytočný kód. Napríklad kontrola typu *null* tam, kde to nie je potrebné. *Code Converter* umožňuje za pomoci nástroja Resharper upraviť formátovanie konvertovaného kódu podľa vlastných požiadaviek. Podporuje konverziu celých projektov. Na nástroji stále prebiehajú úpravy, ktorých je ale postupom

času stále menej, keďže zostávajú len ťažko riešiteľné problémy. Verejne dostupný repozitár obsahuje napríklad aj zoznam nedostatkov, ktorých riešenie je príliš obtiažne[4], alebo zoznam defektov, ktoré si vyžadujú rýchle riešenie[5]. Tento nástroj má zo všetkých ďalej spomenutých najvyššiu šancu na budúce využitie pre rozsiahly *legacy* kód. Musel by ale najprv o niečo zvýšiť svoju efektivitu a odstrániť aspoň z nedostatkov.

VBConversions

Tento nástroj ponúka oproti vyššie spomenutému napríklad vlastné používateľské rozhranie, nie je však integrovaný do prostredia Visual Studio. Okrem toho podporuje všetky verzie projektov jazyka VB.NET s verziami medzi rokmi 2003 a 2019 a niektoré vybrané projekty jazyka C#. Konverzia by mala bez problémov zvládať aj zložitejšie konštrukty ako LINQ, či viac riadkové lambda výrazy. Rovnako je možné vybrať si medzi konverziou malej časti kódu, projektu, alebo viacerých projektov paralelne. Ako v predošlom prípade, tak aj tu má používateľ možnosť upraviť si formátovanie finálneho kódu podľa svojich predstáv. Cena nástroja VBConversions je na úrovni 49,95 dolárov mesačne.

Všetky údaje o produkte boli získané z webovej stránky VBConversions [20].

Instant C#

Instant C# obsahuje oproti predošlým príkladom aj možnosť konverzie zo staršieho jazyka VB6 na novší VB.NET a ten následne prekonvertovať na C#, čo v predošlých prípadoch nefungovalo (Roslyn funguje len na platforme .NET). Ani tu výrobca negarantuje 100% úspešnosť konverzie, no správnosť nástroja bola dovedna otestovaná na miliónoch riadkov kódu. Pravdepodobne sa nehodí pre komerčné využitie, keďže je odporúčaný ako edukačný nástroj pre programátorov v jazyku VB.NET, ktorí sa chcú naučiť C#. Cena nástroja Converter od Tangible Software Solutions je na úrovni 149 dolárov ročne.

Informácie o produkte boli získané z webovej stránky spoločnosti [18].

4.3.3 Vlastný nástroj

Ideou pre vytvorenie vlastného nástroja na konverziu celých projektov jazyka VB.NET na C# je vytvoriť program, ktorý by pomocou syntaktického analyzátora načítaval kód jazyka VB.NET, následne vytvoril rozsiahle mapovanie medzi syntaktickým stromom oboch jazykov. Ďalšou, veľmi dôležitou súčasťou takého nástroja by bolo ošetrenie samotného generovania kódu, aby bol po extrahovaní zo syntaktického stromu jazyka C# správne naformátovaný. To by do veľkej miery mohol vyriešiť refaktorizačný nástroj Resharper, rovnako ako pri nástroji **CodeConverter** 4.3.2.

Pri aktuálnom stave na trhu s nástrojmi pre konverziu jazykov je vlastný nástroj vhodnou alternatívou. Prípadne by mohla vzniknúť abstrakcia nad jedným z nástrojov, ktorá by ešte prepisovala najviac problematické konštrukty. Všetky možnosti s prípadnou konverziou medzi programovacími jazykmi by si však žiadali obrovské množstvo práce a času.

Použitie nástroja Roslyn

Interoperabilita medzi programovacími jazykmi VB.NET a C#, rovnako existencia nástroja Roslyn je základom pre tvorbu vlastného nástroja. Práca s rozhraním prekladača

[4] `https://github.com/icsharpcode/CodeConverter/issues/16`
[5] `https://github.com/icsharpcode/CodeConverter/issues`

na platforme Roslyn už bola vysvetlená v kapitole 2. V prípade vlastného nástroja by drvivá väčšina manipulácie so syntaktickým stromom prebiehala pomocou spomínaných tried **SyntaxRewriter** a **SyntaxFactory**. Obidve triedy majú svoje osobitné implementácie pre jazyky C# a VB.NET.

Princípom transformácie syntaxe by mohlo byť vytvorenie rozsiahleho mapovania medzi jednotlivými elementami syntaktického stromu. Väčšina jazykových konštruktov by bola týmto spôsobom pomerne jednoducho riešiteľná. Napríklad deklarácia tried, premenných, volanie funkcií a mnohé ďalšie. Nasledujúca ukážka demonštruje princípy konverzie na základe typov uzlov v strome:

```
// Kód v jazyku VB.NET
Import System
Namespace AppNamespace
        Public Class AppClass
                ...
        End Class
// Ekvivalentný prepis v jazyku C#
using System;
namespace AppNamespace
{
        public class AppClass
        {
                ...
```

Výpis 4.3: Ukážka transformácie syntaxe medzi jazykmi.

V úvode kódu z ukážky vyššie je zadefinovaná direktíva použitia. Pri mapovaní medzi jazykmi by to znamenalo namiesto uzla **SimpleImportsClauseSyntax** použiť uzol **UsingDirectiveSyntax**. Analogicky pre definície tried. Je zrejmé, že nástroj Roslyn by vlastnú implementáciu transformácie syntaxe výrazne zjednodušoval.

Samozrejme, napriek rozhraniu pre syntaktický strom oboch jazykov, existujú konštrukty, ktorých transformácia je omnoho zložitejšia. Napríklad dotazy na dátové štruktúry typu **LINQ**[6], čo je konštrukt pripomínajúci databázové selekcie a používa sa nad poliami, či zoznamami [9]. Do týchto dotazov je ešte možné integrovať napríklad aj **lambda funkcie**[7].

[6] LINQ – Language Integrated Query

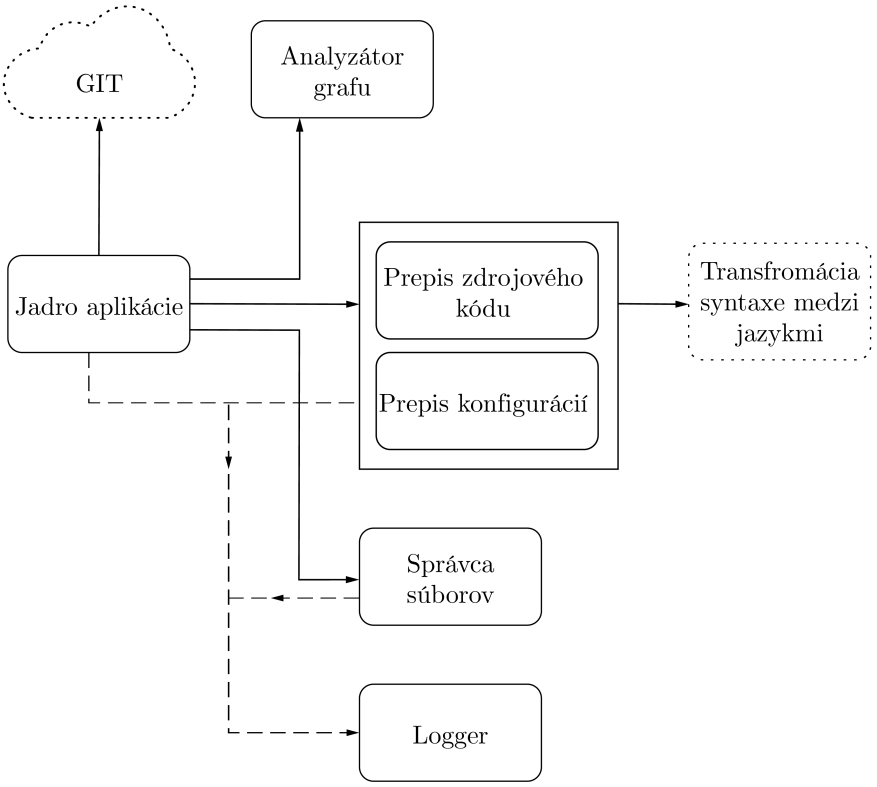
[7] Lambda funkcia – bezmenná metóda použitá ako inštancia delegátu [9].

Kapitola 5

Návrh Aplikácie

Cieľom predošlej snahy bolo nájsť algoritmy, postupy a nástroje, ktoré budú stavebnými kameňmi pre výslednú aplikáciu. Tá pozostáva z nasledujúcich častí:

- **Analyzátor grafu**, ktorý má na vstupe graf závislostí a za pomoci skôr definovaných algoritmov z kapitoly o redukcii grafu 3, dokáže vypočítať postupnosť kontrakcií vrcholov. Výsledkom analyzátora je postupnosť krokov pre redukciu grafu závislostí. Vstupom analyzátora sú tiež názvy projektov, ktoré nemajú byť do procesu zlučovania zahrnuté.
- **Syntaktický transformátor** používaný na prepis referencií, úpravu kódu pri kontrakcii vrcholov, či príprave projektov pred samotnou kontrakciou (Niektoré súbory jazyka VB.NET môžu mať implicitne nastavený menný priestor, čo je potrebné pred operáciou kontrakcie nastaviť.). Pod transformáciu syntaxe je možné zaradiť aj prepis konfiguračných súborov pre projekt a štruktúru *solution*.
- **Správca súborov** na presun súborov po tom, čo syntaktický transformátor dokončí svoju prácu a tým ukončiť operáciu kontrakcie projektov.
- **Logger** na zaznamenávanie všetkých zmien vykonaných syntaktickým transformátorom a správcom súborov, aby mal používateľ detailný prehľad o všetkých zmenách v súborovej štruktúre a vykonaných činnostiach programu.
- **GIT**, alebo podobný správca verzií, ktorý vytvorí pre potreby redukcie grafu novú vetvu, aby sa všetky zmeny dali bez problémov sledovať, testovať a vrátiť späť.
- **Jadro aplikácie** pre riadenie celého jej chodu a komunikáciu s ostatnými modulmi.



Obr. 5.1: Návrh výslednej aplikácie.

Kapitola 6

Implementácia

Na základe požiadaviek je výsledkom aplikácia príkazového riadku na refaktorizáciu *legacy* kódu. Pozostáva z dvoch hlavných komponentov, a to analyzátora grafu a komponentu na zlučovanie projektov. Ďalšími komponentami vo výslednej implementácii sú pomocný *Logger* pre výpisy do príkazového riadku a komponent na testovanie jednotlivých metód analýzy grafu.

6.1 Použité knižnice a štruktúra kódu

Štrukturálny a logický návrh kódu pri implementácii aplikácie sa snaží mať jednotlivé operácie čo najizolovanejšie, aby bolo v prípade potreby jednoduché projekt rozšíriť o želanú funkcionalitu a aby boli chyby pri zmenách lokalizovateľné. Nižšie je možné vidieť prehľad knižníc použitých pri implementácii (Všetky používajú licenciu *MIT*[1].):

- *Microsoft.Build*: Knižnica pre manipuláciu s projektovými konfiguračnými súbormi.
- *Microsoft.Build.Locator*: Balíček na lokalizáciu nástroja MSBuild.
- *Microsoft.CodeAnalysis*: Platforma Roslyn.
- *Microsoft.CodeAnalysis.CSharp*: Sada nástrojov platformy Roslyn pre jazyk C#.
- *Microsoft.CodeAnalysis.VisualBasic*: Nástroje platformy Roslyn pre jazyk VB.NET.
- *Microsoft.CodeAnalysis.CSharp.Workspaces*: Abstrakcia nad *solution* štruktúrou pre projekty jazyka C#.
- *Microsoft.CodeAnalysis.VisualBasic.Workspaces*: Abstrakcia nad *solution* štruktúrou pre projekty jazyka VB.NET.
- *MvsSln*: Knižnica pre manipuláciu s konfiguračným súborom pre štruktúru *solution*.
- *Newtonsoft.Json*: Nástroj pre prácu s dátami vo formáte JSON.
- *Microsoft.Extensions.Logging.Console*: Knižnica pre prispôsobenie práce s príkazovým riadkom.
- *Xunit*: Sada testovacích nástrojov pre platformu .NET.

[1]MIT – Slobodná softvérová licencia vytvorená inštitútom MIT (Massachusetts Institute of Technology)

- *Microsoft.Net.Test.Sdk*: Podporný balíček pre preklad a spustenie testov.
- *Coverlet.Collector*: Nástroj pre analýzu pokrytia kódu pri testoch.
- *Lib2GitSharp*: Balíček pre manipuláciu so systémom riadenia verzií Git.

6.2 Implementačné časti

6.2.1 Analýza grafu

Primárnou triedou na analýzu grafu za použitia algoritmov spomenutých v kapitole 2 je trieda *GraphAnalyzer*. Celý proces analýzy grafu je kvôli prehľadnosti izolovaný v samostatnom projekte s rovnakým názvom. Trieda *GraphAnalyzer* pracuje v dvoch režimoch. Prvým je získanie všetkých krokov postupného zlučovania grafu tak, aby došlo k jeho redukcii. Tým druhým je získanie všetkých aktuálne dostupných krokov v prípade iteratívneho režimu behu aplikácie, aby si používateľ mohol proces redukcie grafu prispôsobiť. Kroky sú používateľovi zobrazené podľa číselných identifikátorov, z ktorých si môže vybrať podľa potrieb. Zároveň má používateľ možnosť zvoliť si, ktoré projekty by mali zostať nedotknuté, odignorované pre kontrakciu vrcholov grafu.

Pretože jazyk C# neobsahuje dátové štruktúry, ktoré efektívne pracujú na bitovej úrovni, bol vytvorený vlastný typ, trieda reprezentujúca bitové pole, *BitField*, pre manipuláciu s bitmi. Najbližšie je k tomu v jazyku C# trieda *BitArray*, ktorá však reprezentuje jeden bit ako dátový typ *Boolean*, a preto zaberá až jeden bajt. Bitové pole potom usporadúva do matice trieda *BitMatrix*. Trieda *GraphAnalyzer* využíva všetky spomenuté druhy matíc na získanie informácii o tom, ktoré projekty môže byť zlúčené. Menovite sú to triedy *AdjacencyMatrix*, *ReachabilityMatrix* a *TransitivityMatrix*. Všetky tieto triedy obsahujú metódu Populate a Update, ktoré sú volané na konci, respektíve začiatku každého kroku analýzy. Trieda *GraphAnalyzer* tiež definuje algoritmus topologického usporiadania, aktualizuje graf závislostí a riadi celý proces redukcie grafu.

Výsledkom analýzy grafu je postupnosť krokov vo forme dvojíc projektov, ktoré sa majú postupne zlúčiť s využitím vopred stanovenej stratégie. Vopred stanovenou stratégiou je myslený spomínaný krokový, alebo úplný prístup k redukcii grafu so špecifikáciou projektov, ktoré nemajú byť do procesu redukcie zapojené.

```
Total number of projects before merge: 242
Total number of C# projects: 84
Total number of VB.NET projects: 158

Program is about to perform solution reduction with following
project merges:
1. CWorldRequest02 into CCMT03
2. Fnz.Web.UI into CCMT03
...
185. Demo.Project into Auto.Tests

Estimated number of projects after merge: 57
Estimated number of C# projects: 22
Estimated number of VB.NET projects: 35

Press ENTER to proceed with merge process.
Press Q to quit.
```

Výpis 6.1: Ukážka výstupu aplikácie s návrhom stratégie redukcie.

6.2.2 Zlučovanie projektov

Hlavnou súčasťou celej aplikácie je projekt s názvom *ProjectMerger*. Jadrom tohto projektu je trieda *Program*, ktorá slúži ako mediátor medzi analýzou grafu a zlučovaním projektov. Jej pomocnou triedou je *ArgumentParser*, ktorý načítavá argumenty príkazového riadku. Samotná kontrakcia jednotlivých projektov pozostáva z niekoľkých krokov. Všetky tieto kroky sú implementované v triedach s príponou „*Manager*“. Je možné na to nazerať, ako na návrhový vzor, kde pre každú väčšiu logickú súčasť zlúčenia projektov existuje samostatná trieda implementujúca rozhranie *IManager*.

6.2.3 Správa dokumentov

V rámci operácie zlúčenia projektov je potrebné presunúť všetky zdrojové súbory zo zložky jedného projektu do zložky druhého, s odignorovaním niektorých konfiguračných súborov. Medzi také konfiguračné súbory patrí napríklad „AssemblyInfo.vb“. Rovnako nie je žiadúce presúvať automaticky generované zložky vývojovým prostredím Visual Studio, ako „My Project“, alebo „Obj“.

Tento proces je implementovaný v triede *DocumentManager*. Na pridávanie súborov z projektu sa používa implementácia abstraktnej triedy *Workspace*, *MSBuildWorkspace*, spomenutá v kapitole 4. Inštancia *MSBuildWorkspace* pri pridávaní projektov pridáva do konfiguračného súboru projektu XML element „*CompileInclude=[Cesta k súboru]*“, čo používa platforma MSBuild na zahrnutie súborov určených k prekladu. Tento element je v niektorých novších verziách projektov implicitný, preto je potrebné ho po presunutí všetkých dokumentov odstrániť. *MSBuildWorkspace* podporuje len veľmi obmedzenú manipuláciu s projektovými konfiguračnými súbormi, preto sa na upravovanie jednotlivých XML elementov v programe používa knižnica nástroja *MSBuild*, menovite *Microsoft.Build.Evaluation*.

Odstrániť projektové súbory po zlúčení je následne nutné pomocou štandardných knižníc na prácu so súborovým systémom. Jednotlivé projekty majú v *legacy* štruktúre *solution* rôzne zanorenia a v projektových zložkách sa často nachádzajú aj ďalšie projekty. Preto

je nutné najprv všetky zložky rekurzívne prehľadať a následne odstrániť tie, v ktorých sa žiadne projektové konfiguračné súbory nenachádzajú.

6.2.4 Prepis implicitných menných priestorov

V prípade projektov jazyka VB.NET je možnosť nastavenia implicitného menného priestoru. Spravidla sa jedná o názov daného projektu. Znamená to, že akákoľvek deklarácia menného priestoru v zdrojovom súbore má v názve implicitne nastavenú predponu. V prípade, že v niektorom súbore chýba, prekladač to vyhodnotí tak, že celý súbor používa menný priestor s implicitným názvom. Implicitný názov je takisto uložený v konfiguračnom súbore pre projekt.

Takáto situácia môže pri presune súborov z projektu B do projektu A spôsobiť sémantické chyby v podobe nerozpoznaného menného priestoru. Po zlúčení projektu B s projektom A už nebude rozpoznávaná implicitná prípona všetkých menných priestorov v súboroch projektu B. Všetkým direktívam použitia a kvalifikovaným názvom v kóde tak chýba predpona názvu projektu. Navyše, objekty z projektu B môžu byť používané v akomkoľvek inom projekte v rámci štruktúry *solution*. Potenciálne konflikty rieši trieda *ImplicitNamespaceManager*. Situácia, ktorá môže vzniknúť je riešiteľná dvoma spôsobmi:

- Prvým je prepísanie všetkých direktív použitia *Import* a názvov objektov, kde sú objekty deklarované explicitne tak, aby neobsahovali predponu s názvom implicitného menného priestoru *Namespace*. Tento spôsob je ale pomerne obtiažny, pamäťovo náročný a obsahoval by obrovské množstvo kódových zmien, čo nie je žiadúce.
- Vhodnejšie je v konfiguračnom súbore odstrániť implicitný názov a v každom súbore pred presunom dopísať chýbajúcu časť názvu menného priestoru. V prípade chýbajúcej deklarácie priestoru, *Namespace*, pridať nový uzol typu *NamespaceBlockSyntax* do syntaktického stromu reprezentujúcej daný zdrojový súbor. Tento prístup používa výsledná aplikácia.

Riešenie problému implicitného menného priestoru ilustruje nasledujúca ukážka:

```
// Súbor A, projekt HelloProject
// Trieda používa implicitný priestor Namespace, HelloProject
Public Class SayHello
       Public Sub SayHello()
              ...
       End Sub
End Class
...
// Potreba zmeny syntaxe nasledovne
Namespace HelloProject
       Public Class SayHello
              ...
       End Class
End Namespace
```

Výpis 6.2: Ukážka prepisu menného priestoru Namespace.

Z hľadiska prepisovacej logiky je potrebné vysporiadať sa aj s vnorenými mennými priestormi a s tým, ktoré uzly majú byť zapuzdrené v bloku menného priestoru. Všetko

prepisovanie syntaxe programu je riešené pomocou implementácie abstraktnej triedy *VisualBasicSyntaxRewriter*, respektíve *VisualBasicSyntaxWalker*, ktorých princípy boli vysvetlené v kapitole 4. *Syntax Walker* sa v tomto konkrétnom prípade používa na získanie informácii o stave menného priestoru a následne *Syntax Rewriter* prevedie príslušné zmeny v syntaktickom strome. Pre potreby prepojenia medzi triedami pracujúcimi so syntaktickým stromom a inými riadiacimi triedami, v tomto prípade spomenutou triedou *ImplicitNamespaceManager*, bol vytvorený návrhový vzor *Caller*. Konkrétna trieda *Caller* slúži na vytvorenie inštancie tried *SyntaxWalker* a *SyntaxRewriter*, kedy z dokumentu, zdrojového súboru získa syntaktický strom, poskytne všetky informácie na prepis a následne zo zmeneného stromu vytvorí nový dokument.

Posledným krokom pre správne fungovanie projektu po zmene implicitného menného priestoru je prepis konfiguračných súborov automaticky generovaných vývojovým prostredím Visual Studio, ako napríklad *„Settings.Designer.vb“*, ktoré obsahujú niektoré z projektových nastavení. Pretože nástroj *MSBuild*, ani prostredie *Visual Studio* ich po zmene nedokážu modifikovať do správnej podoby, je potrebné zo súborov automaticky odstrániť všetky referencie o implicitnom mennom priestore.

6.2.5 Prepis projektových referencií

Ďalšia súčasť zaručujúca správne fungovanie programu po zlúčení projektov je prepis projektových závislostí v konfiguračných súboroch pre projekty. Analogicky, trieda modifikujúca tieto projektové súbory nesie názov *ProjectReferenceManager*. Účelom tejto triedy je prepísať referencie pre všetky priamo závislé projekty na projekte, ktorý je predmetom kontrakcie. Pre potreby tohto prepisu opäť nepostačuje štruktúra *MSBuildWorkspace*, ktorá síce dokáže pridávať, odstraňovať závislostí, no nedokáže získať výlučne priame projektové závislosti pre daný projekt. *MSBuildWorkspace* zobrazuje aj závislosti, kedy projekty medzi sebou nemajú závislosť priamo, teda analogicky podľa podkapitoly 3.3, zobrazuje tranzitívny uzáver závislostí. Preto je aj v tomto prípade podporne využívaná knižnica *Microsoft.Build.Evaluation*. Na prepis sa už však používa trieda *MSBuildWorkspace*, pretože automaticky vytvára element vo formáte XML s relatívnou cestou ku konfiguračnému súboru projektu. Navyše obsahuje metódy, ktoré testujú potenciálnu cyklickú závislosť po pridaní závislosti a overuje tak správnosť aplikácie.

6.2.6 Prepis globálnych konfigurácií prekladu

Ako bolo uvedené v podkapitole o programovacom jazyku VB.NET 2.2.2, konfigurácie prekladu ovplyvňujú, akým spôsobom sa jednotlivé časti kódu vyhodnocujú. Na to, aby nevznikli žiadne sémantické chyby pri preklade programu po zlúčení dvoch projektov jazyka VB.NET a aby boli všetky projekty medzi sebou konzistentné, je nutné v každom súbore vymazať konfiguráciu *Strict* a *Explicit*. Následne nastaviť tieto dve konfigurácie na vypnuté globálne a lokálne v zdrojových súboroch. Každé nastavenie prekladu má v zdrojových súboroch samostatný uzol v rámci syntaktického stromu nástroja *Roslyn*. Prechádzať zdrojové súbory je nutné, pretože špecifikácia konfigurácie prekladu v súbore má prednosť pred globálnym nastavením.

6.2.7 Prepis externých referencií

Okrem závislostí na projektoch obsahujú konfiguračné súbory pre projekty aj informácie o spomínaných balíčkoch *Nuget*, alebo externých súboroch zostavenia, napríklad knižnice,

ktoré vznikli v rámci externých štruktúr *solution*. Ďalej je možné k externým referenciám zaradiť globálne, implicitné direktívy použitia *import*, respektíve *using*, ktoré sa takisto môžu v konfiguračných súboroch vyskytovať. Tu sa nepoužíva *MSBuildWorkspace*, len spomínaná knižnica *Microsoft.Build.Evaluation*, na prepis jednotlivých XML elementov. Pri externých referenciách je zasa potrebné ich presunúť z projektu A do projektu B, avšak napríklad pri rozličných verziách balíčkov *Nuget* je možné, že sa vyskytnú konflikty. V tomto prípade sa do projektu, ktorý obsiahne iný projekt dostane aktuálnejšia verzia balíčka.

Externé balíčky zo servera *Nuget* sa spravidla ukladajú do jednej zložky s názvom „Packages“. Problémom je, že prostredie *Visual Studio* uloží to konfiguračného súboru projektu len relatívnu cestu k zložke. Preto je nutné zaznamenať úroveň zanorenia zložky s konfiguračným súborom projektu a pred zlúčením cestu k balíčkom upraviť.

Okrem cesty, či verzie balíčka môžu externé referencie obsahovať aj iné atribúty, napríklad *Alias* pre názov balíčka, *Description* pre popis, alebo *PublicKeyToken* pre identifikáciu zostavenia. Tieto atribúty však nespôsobujú konflikty, preto ich stačí presunúť.

Medzi implementácie rozhrania IManager, ktoré upravujú konkrétne druhy externých referencií patria *ExternalPackageManager* a *ExternalImportManager*.

6.2.8 Zmena solution

Ako bolo spomínané, zápis do konfiguračných súborov pre projekty bol pomocou triedy *MSBuildWorkspace* značne obmedzený a bolo nutné použiť knižnicu *Microsoft.Build.Evaluation*. V prípade zápisu do konfiguračného súboru pre štruktúru *solution* je situácia komplikovanejšia. Platforma *Roslyn*, ani nástroj MSBuild neponúkajú spôsob, ktorý umožňuje zapisovať do konfiguračného súboru pre *solution*, čo znamená, že tam nie je možné pridávať, ani odstraňovať projekty. Dôsledkom toho je ešte nutné premazať celú súborovú štruktúru odstraňovaného projektu a následne iným nástrojom upraviť konfiguračný súbor pre štruktúru *solution*. Na to sa v triede *SolutionManager* používa open-source nástroj *MvsSln* [16]. Pokusmi však bolo zistené, že pre niektoré druhy *legacy* štruktúr *solution* nie je tento nástroj stopercentne spoľahlivý a pri vyhodnocovaní závislostí projektov sa zacyklí. Jedná sa o situáciu, kedy konfiguračný súbor používa zastaralé vyhodnocovanie závislostí a nástroj na to nemá implementovanú funkcionalitu. Ak táto situácia nastane, je vymazanie riadkov s referenciou konkrétneho projektu prevedené pomocou porovnávania vzorov[2], na základe jedinečného identifikátora odstraňovaného projektu.

6.2.9 Zmena externých štruktúr solution

Doposiaľ boli popisované zmeny nutné pre fungovanie samostatnej štruktúry *solution* pri jej minimalizácii. Na to, aby bola naďalej funkčná celá kódová štruktúra, je potrebné previesť konfiguračné zmeny aj vo všetkých štruktúrach *solution*, ktoré používajú *legacy* projekty vo forme knižničných súborov s príponou „.dll“. V prípade, že je žiadúce odkazovať sa na externý projekt, jediná možnosť, ako to urobiť, je pridať jeho zostavenie, výsledok prekladu projektu vo forme súboru. Všetky informácie o používaných súboroch zostavenia, ako názov, typ, či cesta sú uložené v konfiguračných súboroch pre projekt. Preto pri zlučovaní projektov je potrebné uchovávať informácie o názvoch zostavení a následne ich v externých štruktúrach *solution* zmeniť. V opačnom prípade by neboli preložiteľné a aplikácie ako celky by nefungovali. Všetku funkcionalitu s ukladaním informácií o zostavení počas minimalizácie

[2]Porovnávanie vzorov (Pattern Matching) je technika hľadania vzorov v sekvencií dát, v tomto prípade znakov.

závislostí do súboru vo formáte *JSON* poskytuje trieda *AssemblyManager*, ktorá rovnako, keď program beží v režime pre zmenu zostavenia prepisuje konfiguračné súbory projektov v externých štruktúrach *solution*.

```
"Assembly":
{
       "OldName": "OldProj",
       "NewName": "NewProj",
       "OldPath": "../OldProj/Bin/OldProj.dll",
       "NewPath": "../NewProj/Bin/NewProj.dll",
}
```

Výpis 6.3: Ukážka uložených JSON dát o zostavení pri zlučovaní projektu.

6.2.10 Git

Z dôvodu nutnosti uchovávať informácie o premiestňovaných a prepisovaných súboroch, mali by byť tieto zmeny zachytávané systémom riadenia verzií. Najpoužívanejším je systém riadenia verzií *Git*. Pri presune súboru z jednej zložky do druhej *Git* zaznamená tento presun len ako vytvorenie nového súboru, čím sa prakticky celá jeho história vymaže. Preto je nutné presun realizovať pomocou príkazu „*git mv*“, ktorý daný súbor presunie v rámci jednej vetvy a tým sa história súboru uchová. Realizácia práce so systémom riadenia verzií prebieha v triede *GitManager*, ktorá na prácu s ním používa knižnicu s názvom *LibGit2Sharp*[3]. Pomocou tejto knižnice je možné spúšťať jednotlivé príkazy pre *Git*. Implementovanou funkcionalitou sú vytvorenie novej vetvy, spomenuté presuny súborov a zadávanie vstupných údajov.

Problémom pri presune súboru pomocou príkazu „git mv“ je, že takto presunutý súbor nie je možné presunúť opätovne. Nejedná sa o zriedkavú situáciu pri redukcii projektov, preto sa po každej kontrakcii zmeny prevedú aj vo vzdialenom repozitári.

V budúcnosti je plánované aj rozšírenie pre aplikáciu na správu verzií *Accurev*[4], ktorá sa v FNZ takisto, aj keď v menšej miere používa.

6.2.11 Interakcia s používateľom

Keďže sa jedná o aplikáciu príkazového riadku, spustenie a každá potrebná interakcia s používateľom prebieha výpismi do príkazového riadku. Pretože je program určený na automatizovanú minimalizáciu a kódové zmeny, používateľské rozhranie nie je žiadúce. Navyše, vizualizácia grafu závislostí je možná vo vývojovom prostredí Visual Studio a graf s vyše 240 vrcholmi je pre používateľa pomerne neprehľadný. O informácie o behu programu, detailný výpis chybových hlášok, prácu so systémom riadenia verzií a formátovanie zobrazeného textu sa stará samostatný projekt *Logger*. Pre čo najefektívnejšiu manipuláciu s príkazovým riadkom sa pre účely tohto projektu používa knižnica *Microsoft.Extensions.Logging*, ktorá umožňuje si výpisy prispôsobiť.

[3] `https://github.com/libgit2/libgit2sharp`

[4] `https://www.microfocus.com/en-us/products/accurev/overview`

6.2.12 Refaktorizácia syntaxe

Riešenie obsahuje aj modul, ktorý slúži na refaktorizáciu syntaxe. V triede *SyntaxRefactor* je implementovaný nástroj na automatický prepis menného priestoru pre projekt. Jedná sa o doplnok k minimalizácii grafu závislosti.

V prípade, že menný priestor neodpovedá umiestneniu zdrojového súboru, zmení jeho názov a následne si pomocou sémantického rozhrania platformy *Roslyn* vyhľadá všetky použitia pomocou triedy *SymbolFinder*. O funkcionalitu prepisovania menných priestorov a ukladanie dát sa stará trieda *NamespaceManager*. Trieda *UsingManager* potom vo všetkých súboroch, kde bol použitý starý, nesprávny názov menného priestoru, prepíše pomocou spomínaných implementácií tried *SyntaxRewriter* a *SyntaxWalker* direktívy použitia. Samozrejme, v kóde sa tieto direktívy nachádzať nemusia a jednotlivé triedy a ostatné spomínané typy môžu byť referencované svojim celým kvalifikovaným názvom, teda s predponou názvu menného priestoru, do ktorého patria. Referencie objektov s kvalifikovaným názvom je v kóde pomerne obtiažne hľadať pomocou triedy *SyntaxWalker*, keďže sa môžu vyskytovať na mnohých miestach. Jedná sa napríklad o volanie metódy statickej triedy, vytvorení inštancie triedy, deklarácií parametrov a mnoho ďalších prípadov. Spomínaná trieda *SymbolFinder* dokáže vyhľadať informácie o presnom umiestnení objektu v zdrojovom súbore. Preto ak sa nenájde direktíva použitia, ktorá bola pre daný objekt zmenená, program navštívi všetky miesta jeho použitia v zdrojovom súbore a prepíše kvalifikovaný názov.

V praxi takáto zmena zdrojového súboru vyzerá nasledovne:

```
// Súbor A sa nachádza v zložke projektu
// SimpleProject, v podzložke SimpleClasses
namespace SimpleProject.Classes
{
        public class SimpleClass
        {
                ....
// Potrebný prepis na
namespace SimpleProject.SimpleClasses
{
        ...

// Súbor B obsahuje odkaz na triedu SimpleClass
// Obdobne potrebný prepis direktívy použitia
using SimpleProject.Classes;
var simpleClass = new SimpleClass();

// Súbor C obsahuje kvalifikovaný názov
var simpleClass = new SimpleProject.Classes.SimpleClass();
// Prepis na new SimpleProject.SimpleClasses.SimpleClass();
```

Výpis 6.4: Ukážka úpravy názvoslovia menného priestoru Namespace.

Keďže menenú *legacy* štruktúru *solution* používajú aj mnohé externé entity, ako napríklad externé štruktúry *solution*, alebo *task* systém (2.1), ktorý v určitých časoch invokuje časti kódu, bude potrebné rozšíriť refaktorizačný nástroj aj o prepis externých entít. Mo-

mentálne je nástroj použiteľný pre interné zmeny v rámci štruktúry *solution* pre potreby úpravy názvov pred kontrakciou, čo výrazne pomáha predchádzaniu konfliktom.

Pre *legacy* komponent má teda využitie, kde dopĺňa funkciu nástroja *Resharper*. V prípade, že si pomocou nástroja *Resharper* používateľ zvolí úpravu menného priestoru vo všetkých súboroch projektu, môže zmeniť menný priestor aj v iných projektoch. Je to z toho dôvodu, že *Resharper* neopravuje menný priestor podľa projektu, ale podľa všetkých výskytov nesprávneho menného priestoru. Nástroj *Resharper* takisto predpokladá výskyt implicitného menného priestoru pri projektoch jazyka *VB.NET*, preto projekty, ktoré implicitný priestor nastavený nemajú, neprepíše do požadovanej podoby. Spomenuté rozšírenie aplikácie prepisuje menný priestor len na úrovni projektov a môže, ako bude spomenuté v nasledujúcej kapitole, napomôcť k minimalizovaniu nejasností v názvoch 7.0.1.

Kapitola 7

Testovanie

Testovanie výslednej aplikácie prebiehalo počas vývoja v samostatnom projekte na testy pre prácu s grafom a v samostatnej štruktúre *solution*, ktorá bola vytvorená pre testovanie aplikácie a obsahuje rôzne okrajové prípady. Výsledná aplikácia bola rovnako testovaná na sade *legacy* kódu, pričom každá sada obsahovala viac ako 240 projektov. Testy prebiehali konkrétne na troch takýchto sadách. Nasledujúca tabuľka zobrazuje údaje o testovacích sadách:

	Sada 1	Sada 2	Sada 3
Počet projektov	242	257	249
Projektov jazyka VB.NET	158	166	157
Projektov jazyka C#	84	91	92
Počet súborov	28225	30177	29199
Priemerný počet závislostí na projekt	10	9	10

Tabuľka 7.1: Štatistiky testovacích sád.

Hlavným cieľom práce bola čo najväčšia možná redukcia grafu závislostí projektov. Pri testoch na maximálnu možnú redukciu grafu závislostí boli dosiahnuté nasledujúce výsledky:

	Sada 1	Sada 2	Sada 3
Počet projektov pred	242	257	249
Počet projektov po	57	67	58
Počet projektov s implicitným menným priestorom	21	21	21
Projektov jazyka VB.NET pred	158	166	157
Projektov jazyka VB.NET po	35	44	39
Projektov jazyka C# pred	84	91	82
Projektov jazyka C# po	22	23	19
Počet premiestnených súborov	11329	13159	11226
Počet prepísaných súborov	2814	2572	2714
Čas minimalizácie v minútach	73	78	62

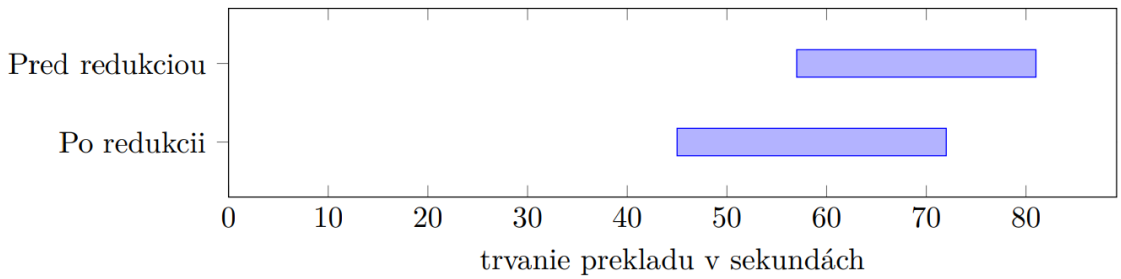
Tabuľka 7.2: Výsledky redukcie grafu závislostí.

Jednotlivé časové údaje v tabuľke vyššie reprezentujú čas behu programu, nie čas samotnej minimalizácie. Ako bude prezentované v nasledujúcej podkapitole 7.0.1, pri niektorých kontrakciách je nutné ručne prepisovať problematické časti kódu. Čas celého procesu minimalizácie preto typicky trval niekoľko hodín.

Výsledkom refaktorizácie *legacy* kódu je teda redukcia grafu závislostí o takmer 80 percent. Pomocou vývojového prostredia Visual Studio je možné zobraziť stav grafu závislostí medzi projektami pred a po minimalizácii.

Najdôležitejším faktorom pri testovaní je, ako dokáže minimalizovaný graf závislostí znížiť čas prekladu programu. Napriek tomu, že preklad jednotlivých projektov prebieha paralelne, projekty, ktoré sú závislé na iných, musia čakať na ich výsledný súbor zostavenia. Minimalizáciou grafu závislostí by sa toto čakanie na preklad projektov malo výrazne skrátiť. Rovnaký princíp, ako pri preklade programu pomocou platformy *MSBuild*, je použitý pri načítavaní štruktúry *solution* do vývojového prostredia Visual Studio. Pre čo najväčšiu pomoc vývojárom pristupuje Visual Studio napríklad k symbolom pri inteligentnom doplňovaní kódu. Takéto funkcie však značne trpia nielen na komplexný graf závislostí, ale na obrovské množstvo zdrojových súborov. Mnoho vývojárov kvôli tomuto výkonnostnému problému nepoužíva nástroj *Resharper*, ktorý by za normálnych okolností výrazne ušetril čas. Bolo by naivné si myslieť, že všetky problémy spomenuté vyššie zmiznú pri redukcii projektov. Faktom však je, že výsledný program dokázal znížiť dobu prekladu o približne 20 percent.

Meranie prebiehalo opakovaným spustením (10 krát) prekladu každej z testovacích sád projektov pred a po minimalizácii grafu. Graf nižšie demonštruje časové rozmedzia prekladu všetkých testovacích sád pred a po vykonanej redukcii:



Obr. 7.1: Porovnanie trvania prekladu pred a po minimalizácii všetkých testovacích sád v sekundách.

Kvôli robustnosti *legacy* kódu prebiehalo testovanie a následné ladenie nástroja pre rôzne okrajové prípade vyše dvoch mesiacov. Pretože všetky *legacy* komponenty obsahujú dohromady desaťtisíce projektov, stále je možné, že na nejakom projekte mimo testovacej sady budú špecifiká, ktoré si vyžadujú úpravu a rozšírenie funkcionality. Takýto pomalý posun v testovaní je daný najmä kvôli času, ktorý je potrebný na pretestovanie jednej sady. V niektorých prípadoch bolo pomerne zložité zistiť, kde presne v procese nastala chyba, preto sú jednotlivé časti programu čo najviac izolované. Navyše, pri iteratívnom testovaní môžu rôzne kombinácie zlúčenia spôsobiť rôzne problémy.

Niekedy sa dokonca jednalo o nedokonalosti, respektíve obmedzenia použitých knižníc. Na platforme *Roslyn* ešte stále prebieha vývoj a len v tomto momente je nahlásených vyše

5000 defektov[1]. Defekty, ktoré obmedzovali proces kontrakcie, alebo nútili k manuálnym zásahom bola napríklad nekompatibilita pre rôzne verzie projektov.

Do budúcna sa na platforme *Roslyn* plánujú rozsiahlejšie úpravy, ako napríklad zapisovanie do konfiguračného súboru pre štruktúru *solution*. V súčasnosti sa platforma skôr sústreďuje na samotné úpravy zdrojových súborov a neobsahuje nástroje na pridávanie a odstraňovanie projektov. Nastávali aj situácie, kedy nebolo možné odstrániť určitý XML element z konfiguračného súboru pre niektoré verzie projektov a chyby neobsahovali informatívnu chybovú hláškou. Často sa jednalo o chyby, ktoré sa neopakovali a chyby, ktoré sa pominuli reštartovaním vývojového prostredia Visual Studio.

Na základe výsledkov testov je však bezpečné povedať, že aplikácia dokáže analyzovať graf závislostí projektov a previesť redukciu grafu. V niektorých prípadoch je však proces redukcie potrebné zastaviť a ručne vyriešiť konflikty v kóde.

7.0.1 Nejasný kvalifikovaný názov

Situáciou, pri ktorej sú nutné manuálne zásahy do kódu po redukcii pre správne fungovanie, je výskyt objektov s konfliktným kvalifikovaným názvom. V týchto prípadoch je dôvodom rozpoznávanie nových menných priestorov projektu po zlúčení. Nejasnosti, respektíve duplicity sa môžu vyskytovať medzi projektami, ale aj medzi kvalifikovanými názvami v štandardných knižniciach a zlúčených častiach projektov. Často sa jedná o dve identické triedy, ktoré boli pred zlúčením v dvoch rozličných projektoch a preto nespôsobovali sémantické chyby.

Jedným z príkladov je implementovaná funkcia *„string.IsNullOrEmpty“* v jednom z *legacy* projektov. Funkciu s rovnakou signatúrou však obsahuje aj štandardná knižnica *System*. Detekcia a automatické odstraňovanie týchto sémantických chýb by si vyžadovalo hlbokú analýzu všetkých použitých symbolov. Rozhodnutie, ktoré časti kódu sa pri duplicitnom kvalifikovanom názve majú upraviť, respektíve odstrániť je vhodné nechať na používateľa. Dá sa na to nazerať ako na príležitosť odstrániť časti kódu, ktoré nie sú žiadúce.

Ďalším typom sémantickej chyby nejasného kvalifikovaného názvu je prednosť menného priestoru pred typovým názvom. V prípade, že projekt po zlúčení začal rozpoznávať viacero menných priestorov, môže sa stať, že prekladač už nerozpozná napríklad názov triedy, pretože má konfliktný názov s menným priestorom. Jedná sa o chybu, pri ktorej sa používa direktíva použitia a nie kvalifikovaný názov triedy.

```
using Core.Adapters

namespace Adapter.Usage
{
        public class AdapterUsingClass
        {
                var adapter = new Adapter();
                ....
```

Výpis 7.1: Ukážka nejasnosti medzi názvom triedy a menným priestorom.

Ukážka vyššie nech pochádza zo zdrojového súboru projektu A. Po zlúčení začal projekt A rozpoznávať aj menné priestory projektu X a jedným z menných priestorov projektu X je

[1] `https://github.com/dotnet/roslyn/issues`

Adapter. V tom prípade kód z ukážky nebude preložiteľný a je potrebné pri vytvorení inštancie triedy *Adapter* jej kvalifikovaný názov, teda *Core.Adapters.Adapter*.

Tieto nejasnosti sú zriedkavé, no z testovania bolo zistené, že sa vyskytujú každých, v priemere deviatich iteráciách. Z tohto dôvodu je pre používateľa vhodnejšie, minimálne v počiatočných fázach testovania sady *legacy* kódu, použiť iteratívny beh programu. Keďže *legacy* štruktúry *solution* vychádzajú z rovnakého kódového jadra, je veľmi pravdepodobné, že tieto sémantické chyby budú vo väčšine sád projektov rovnaké. Postupom času sa zaviedli štandardy vo forme pravidiel, ako písať kód a prebiehajú pravidelné posudzovania kódu (code reviews), preto je takmer vylúčené, že nový kód bude mať také nedostatky. V testovacej fáze sa preto zozbierali jednotlivé chyby, ktoré bude možné hromadne pred začiatkom redukcie závislostí upraviť a zabezpečiť plynulejší beh redukcie.

7.0.2 Poradie zlúčenia projektov

Ako bolo uvedené v podkapitole o poradí zlučovania susedných vrcholov grafu 3.3.7, existuje mnoho spôsobov, akými je možné zvoliť krok programu pri zlučovaní susedných projektov. Najdôležitejším kritériom bolo, ako poradie zlučovania ovplyvní konflikty pri nejasných kvalifikovaných názvoch. Kritéria ako počet presunutých dokumentov, alebo prepísaných projektových závislostí možno považovať za vedľajšie, pretože nenútili používateľa zastaviť proces a opraviť nejasnosti. Testované boli nasledujúce tri prístupy:

- Postupné zlučovanie od listových vrcholov.
- Postupné zlučovanie od koreňových vrcholov.
- Zlučovanie od vrcholov s najnižším počtom závislostí.

Na základe experimentov s uvedenými prístupmi bola vybraná možnosť pre zlučovanie od listových vrcholov. Zatiaľ čo pri stratégii od koreňových vrcholov nastávali konflikty každých, približne päť iterácii, pri zvyšných dvoch prístupoch to bolo v priemere identické, každých deväť iterácií. Rozhodla preto zložitosť algoritmu pre výber dvojice na zlúčenie, kde pri možnosti s najnižším počtom závislostí bolo v každom kroku potrebné porovnať všetky dvojice. Prirodzene, v pokročilých fázach, keď sa počet projektov znižoval a ich veľkosť zvyšovala, nastávali konflikty častejšie.

Kapitola 8

Zhodnotenie dosiahnutých výsledkov

Prvou úlohou zadania bolo preskúmať možnosti redukcie grafu závislostí, vrátane problematiky vrcholov, ktoré už nie je možné zredukovať a transformácie syntaxe medzi jazykmi. Na základe toho boli preštudované algoritmy na prácu s acyklickým orientovaným grafom a následne predložená stratégia, ktorá dostane graf do stavu, kedy už nie sú možné žiadne ďalšie redukcie. Pri uzloch, ktoré ďalej nie je možné redukovať, sa javí ako možnosť prepísať problematické časti kódu medzi projektami. Tento prístup je však problematický, pretože by si vyžadoval hlbokú syntaktickú, sémantickú analýzu a množstvo zásahov do kódu. Pre transformáciu syntaxe medzi jazykmi boli prezentované existujúce nástroje a princíp možného fungovania vlastného nástroja pre prípadné rozšírenie. Použitie transformácie je však vhodné skôr pred redukciou projektov kvôli lepšej testovateľnosti menších celkov.

Počas štúdia algoritmov a spôsobov, ktorými je možné redukovať graf závislostí neboli objavené existujúce riešenia pre konkrétny typ problému s dvomi druhmi vrcholov, ako v prípade tejto práce. Blízko má napríklad implementácia prekladača pre *Tensorflow*[1], ktorá obsahuje riešenie problému s rôznymi druhmi vrcholov, no nerieši poradie kontrakcie, ani vrcholy, ktoré spolu nesusedia. Zaoberá sa iba potenciálnym vznikom silne súvislých komponentov. Preto bol na základe existujúcich algoritmov pre acyklické orientované grafy navrhnutý nový spôsob na účely tejto práce.

V ďalších častiach sa práca zaoberá prehľadom .NET technológií a nástrojov pre automatickú manipuláciu s projektami. Najdôležitejšou súčasťou výslednej aplikácie je práca s platformou *Roslyn*, ktorá poskytuje rozhranie k prekladačom programovacích jazykov C# a VB.NET.

Na základe stratégie pre redukciu grafu, štruktúry oboch programovacích jazykov, možností nástroja *Roslyn* a platformy *MSBuild*, bola navrhnutá aplikácia pre redukciu závislostí projektov. Časť návrhu a implementácie preto popisuje, ako je nutné zmeniť zdrojové a konfiguračné súbory na to, aby redukcia projektov minimalizovala prípadné chyby.

Chyby však boli počas testovacej fázy odhalené, najväčší problém spôsobujú nejasnosti kvalifikovaných názvov, čo má za následok nutnosť manuálnych zásahov do procesu redukcie. Okrem toho boli odhalené aj niektoré z nedostatkov nástroja *Roslyn*.

Výslednú aplikáciu je možné použiť na redukciu závislostí medzi projektami a pre potreby zmien menných priestorov je zahrnutá aj možnosť prepisu menného priestoru. Aplikáa-

[1] `https://github.com/tensorflow/tensorflow/blob/r1.5/tensorflow/compiler/jit/mark_for_compilation_pass.cc`

ciu je možné v budúcnosti rozšíriť o spomínanú transformáciu syntaxe medzi programovacími jazykmi alebo prepis menných priestorov aj mimo *legacy* štruktúru *solution*. Rovnako je možné rozšíriť nástroj o detekciu mŕtveho kódu, kde je nutné kontrolovať použitie jednotlivých objektov aj mimo *legacy* komponentov.

Kapitola 9

Záver

Cieľom tejto bakalárskej práce bolo nájsť možnosti redukcie grafu závislostí .NET projektov, popísať metódy prípadnej ďalšej redukcie vrcholov grafu a následne implementovať aplikáciu, ktorá danú redukciu prevedie. Práca začína oboznámením sa s problémom, kódovou štruktúrou určenou na refaktorizáciu a základnými pojmami z prostredia .NET. Ďalej popisuje a porovnáva techniky na prácu s acyklickým orientovaným grafom, čoho výsledkom je algoritmus redukujúci graf. Neskôr je zostavený prehľad činnosti prekladača a základné použitie nástrojov pre analýzu a transformáciu syntaxe zdrojových súborov v prostredí .NET. Poslednou časťou je fáza návrhu, implementácie a testovania výslednej aplikácie.

Bolo nutné naštudovať si informácie o procese prekladu a zostavenia programu na platforme .NET, práci s acyklickým orientovaným grafom a tiež existujúce nástroje, ktoré dovoľujú manipuláciu s .NET projektami a zmenu syntaxe programovacích jazykov C# a VB.NET.

Na základe získaných informácií sa podarilo navrhnúť a zostaviť aplikáciu v súlade s požiadavkami a vytvoriť tak refaktorizačný nástroj. K nástroju bolo vytvorené aj rozšírenie v podobe automatického prepisu menných priestorov. Tento nástroj môže byť v budúcnosti rozšírený napríklad o transformáciu syntaxe medzi jazykmi alebo detekciu mŕtveho kódu. V zásade môže byť rozšírený o akúkoľvek funkcionalitu, ktorá vyžaduje zásah a informácie aj mimo upravovanú štruktúru .NET *solution*.

Najväčším problémom pri implementácii bolo zdĺhavé testovanie kvôli obrovským testovacím sadám a povahe nástroja *Roslyn*. Nástroj Roslyn je postavený na princípe nemennosti, čo robí aplikáciu podstatne časovo a pamäťovo náročnejšiu a napriek jeho vysokej využiteľnosti obsahuje stále chyby. Z hľadiska nedostatkov výsledná aplikácia nedokáže riešiť konflikty neurčitých názvov, ktoré vzniknú pri zlučovaní .NET projektov, čo má za následok nutnosť manuálneho prepisu kódu. Vyriešenie tohto problému by si vyžadovalo rozsiahlu syntaktickú a sémantickú analýzu celého *legacy* kódu a všetkých použitých knižníc.

Zadanie práce sa podarilo splniť, nástroj prešiel sériou testov a bude hromadne použitý na refaktorizáciu *legacy* kódu v spoločnosti FNZ. Z osobného hľadiska oceňujem nové poznatky z oblasti teórie grafov a preniknutie do hĺbky architektúry technológií .NET.

Literatúra

[1] *Get started with syntax analysis* [online]. Washington: Microsoft, 2021 [cit. 2023-03-20]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis.

[2] *Work with a workspace*. Washington: Microsoft, 2021 [cit. 2023-03-04]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-workspace.

[3] *Work with syntax*. Washington: Microsoft, 2021 [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-syntax.

[4] *An introduction to NuGet* [online]. Washington: Microsoft, 2022 [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/nuget/what-is-nuget.

[5] *Metadata and Self-Describing Components* [online]. Washington: Microsoft, 2022 [cit. 2023-04-17]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/metadata-and-self-describing-components.

[6] *Assemblies in .NET* [online]. Washington: Microsoft, 2023 [cit. 2023-03-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/assembly.

[7] *MSBuild* [online]. Washington: Microsoft, 2023 [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022.

[8] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: principles, techniques & tools*. Second edition. Boston: Addison Wesley, 2007. ISBN 0-321-48681-1.

[9] Albahari, J. a Albahari, B. *C# 6.0 in a nutshell*. Sixth edition. Sebastopol: O'Reilly, 2016. ISBN 978-1-491-92706-9.

[10] Bock, J. *.NET Development Using the Compiler API*. Shakopee, Minnesota: Apress, 2016. ISBN 978-1-4842-2111-2.

[11] Demel, J. *Grafy a jejich aplikace*. Vyd. 2., (Vlastním nákladem 1.). Libčice nad Vltavou: J. Demel, 2015. ISBN 978-80-260-7684-1.

[12] Griffiths, I., Flanders, J. a Sells, C. *Mastering Visual Studio .NET: Getting the Most Out of the Visual Studio .NET Environment*. First edition. Sebastopol: O'Reilly, 2003. ISBN 9781491949603.

[13] ICSharpCode. *CodeConverter* [online]. San Francisco: GitHub, 2022 [cit. 2023-01-10]. Dostupné z: https://github.com/icsharpcode/CodeConverter/wiki.

[14] JetBrains. *Resharper* [online]. [cit. 2023-04-17]. Dostupné z: https://www.jetbrains.com/resharper/.

[15] Kahn, A. B. Topological sorting of large networks. *Communications of the ACM*. New York: Associationd for Computing Machinery. 1962, zv. 5, č. 11, s. 558–562. DOI: 10.1145/368996.369025. Dostupné z: https://doi.org/10.1145/368996.369025.

[16] Kuzmin, D. *MvsSln* [online]. San Francisco: Github, 2022 [cit. 2023-04-08]. Dostupné z: https://github.com/3F/MvsSln.

[17] Lomax, P., Petrusha, R. a Steven Roman, P. *VB.NET Language in a Nutshell*. Second edition. Sebastopol: O'Reilly, 2002. ISBN 0-596-00308-0.

[18] Tangible Software Solutions, I. *The Most Accurate and Reliable Source Code Converters* [online]. [cit. 2023-01-10]. Dostupné z: https://www.tangiblesoftwaresolutions.com.

[19] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*. Society for Industrial and Applied Mathematics. zv. 1, č. 2. DOI: 10.1137/0201010. Dostupné z: https://doi.org/10.1137/0201010.

[20] VBConversions, L. *VB.Net to C# Converter* [online]. [cit. 2023-01-10]. Dostupné z: https://vbconversions.com.

[21] Češka, M., Vojnar, T., Smrčka, A. a Rogalewicz, A. *Teoretická informatika: Studijní text* [online]. Brno: FIT VUT, 2020 [cit. 2023-30-04]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

Príloha A

Obsah pamäťového média

Koreňový adresár
- App ... Spustiteľný súbor a potrebné knižnice.
- Dokumentacia.html ... Dokumentácia vygenerovaná nástrojom *Doxygen*.
- Navod.pdf ... Detailný návod na preklad a použitie programu.
- ProjectMerger ... Zložka so zdrojovými súbormi.
- Sprava.pdf ... Písomná správa bakalárskej práce.
- TestSolution ... Zložka s testovacou štruktúrou *solution*.

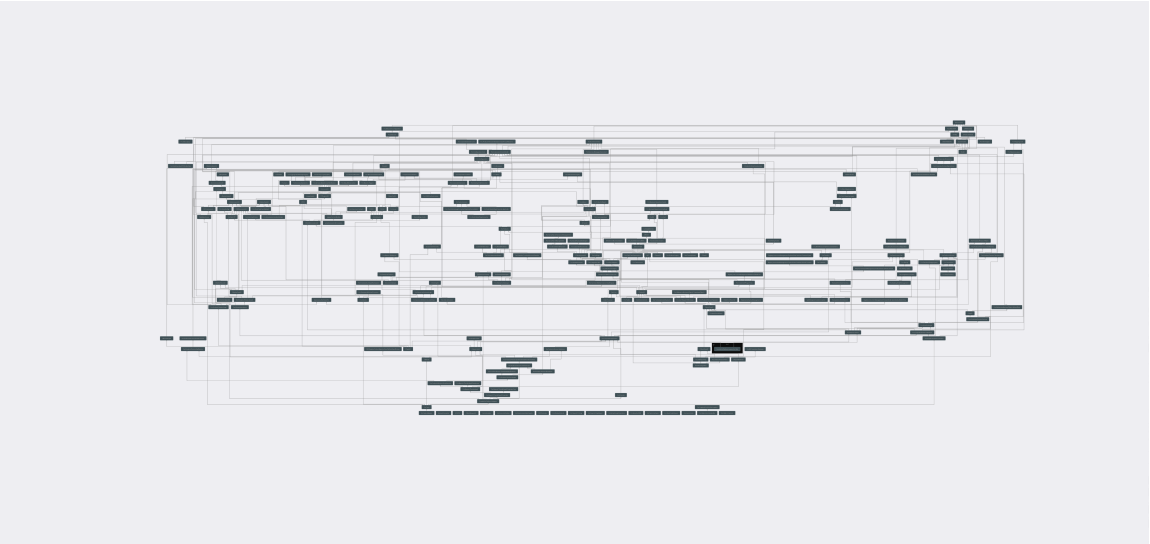
Príloha B

Štruktúra kódu

```
Štruktúra kódu
├── GraphAnalyzer ... Projekt na analýzu grafu.
│   ├── GraphAnalyzer.cs ... Základná logika analýzy grafu.
│   ├── BitArray.cs ... Definícia bitových polí.
│   ├── Enums ... Vymenované typy.
│   ├── Graphs ... Entity reprezentujúce graf.
│   ├── Helpers ... Pomocné triedy.
│   ├── Matrices ... Triedy pre jednotlivé matice.
│   └── Nodes ... Uzly grafu.
├── GraphTests ... Projekt pre testovanie funkcionality analýzy
│                  grafu.
├── Logger ... Projekt pre výpisy do príkazového riadku.
│   ├── EventLogger.cs ... Konkrétna implementácie triedy Logger.
│   ├── LogEvents ... Udalosti, pri ktorých nastávajú výpisy.
│   └── Formatter ... Formátovanie triedy Logger.
└── ProjectMerger ... Projekt pre samotnú kontrakciu grafov.
    ├── Callers ... Triedy typu caller, abstrakcia nad triedami
    │              (Re)Writer.
    ├── Managers ... Riadiace štruktúry dielčích operácií
    │               minimalizácie.
    ├── Rewriters ... Triedy na prepis kódu.
    ├── Walkers ... Triedy na navštívenie uzlov syntaktického stromu.
    ├── Enums ... Potrebné výčtové typy.
    ├── Entities ... Potrebné entity.
    ├── Helpers ... Pomocné triedy.
    ├── ArgumentParser.cs ... Procesovanie argumentov príkazového riadku.
    ├── Program.cs ... Vstupný bod programu.
    ├── ProjectContractor.cs ... Trieda pre kontrakciu projektov.
    ├── SolutionLoader.cs ... Trieda pre „lightweight" načítavanie grafu na
    │                         začiatku programu.
    └── SyntaxRefactorer.cs ... Rozšírenie pre refaktorizáciu syntaxe.
```

Príloha C

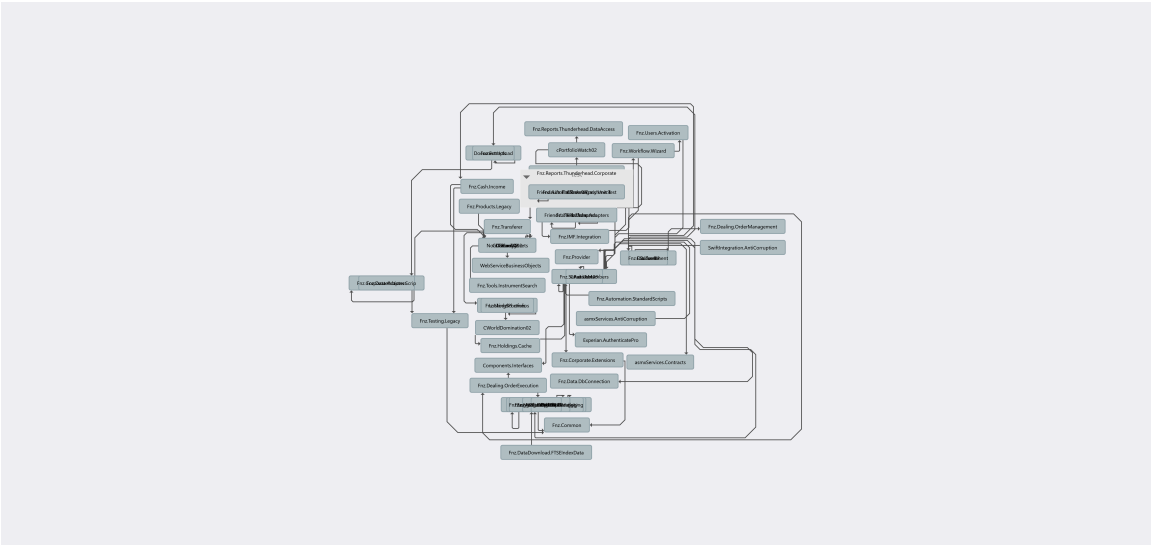
Graf závislostí pred redukciou



Obr. C.1: Graf závislostí pred redukciou zobrazený pomocou prostredia Visual Studio 2022.

Príloha D

Graf závislostí po redukcii



Obr. D.1: Graf závislostí po redukcii zobrazený pomocou prostredia Visual Studio 2022.