Izrada alata za generiranje C# programskog koda

Kranjčina, Karlo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Karlo Kranjčina

IZRADA ALATA ZA GENERIRANJE C# PROGRAMSKOG KÔDA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Karlo Kranjčina

Matični broj: 0016146923

Studij: Informacijski i poslovni sustavi

IZRADA ALATA ZA GENERIRANJE C# PROGRAMSKOG KÔDA

ZAVRŠNI RAD

Mentor:

Dr. sc. Marko Mijač

Varaždin, rujan 2023.

Karlo Kranjčina

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj završni rad tematizira problem pisanja ponavljajućeg programskog kôda koji produljuje vrijeme pisanja i smanjuje produktivnost programera. Takav kôd dovodi do pogrešaka u pisanju i rezultira u produljivanju procesa izrade softverskog rješenja. Budući da je generiranje programskog kôda širok pojam, spomenut će se i opisati bitni dijelovi koji bi trebali pomoći u razumijevanju cilja ovog rada. Cilj rada je prepoznati stanje i trenutne mogućnosti generiranja programskog kôda i dati detaljan uvid u princip rada Roslyn generatora, to jest kompilatora kôda. Sredstvo ostvarivanja navedenog cilja bit će i izrada alata za generiranje programskog kôda klasa C# programskog jezika. Rezultati rada jasno pokazuju kvalitetu i mnogobrojne mogućnosti generiranja Roslyn kompilatorom.

Ključne riječi: Roslyn; .NET; C#; Compiler; API; generiranje; programski kôd

Sadržaj

1. Uvod

Razvoj softverskog rješenja sastoji se od nekoliko dijelova. Ti dijelovi uključuju pisanje dokumentacije, specifikacije i dizajna rješenja te pisanje programskog kôda. Budući da su vremenski rokovi uvijek prisutni prilikom izrade softverskih rješenja, potrebno je pronaći način ubrzanja procesa. Iako se navedeni dijelovi većinski prave ručno, moguće je ubrzati proces izrade generiranjem određenih segmenata. Kad je u pitanju dizajn rješenja, postoje alati čija je svrha generirati dijagrame kojima se prikazuje arhitektura i struktura softverskog rješenja. Također, tijekom pisanja programskog kôda, moguće je generirati isječke kôda i time znatno smanjiti utrošeno vrijeme izrade.

Pisanje programskog kôda zahtjevna je zadaća koja iziskuje puno vremena, veliku količinu planiranja, mentalni napor i koncentraciju. Pisanje može postati ponavljajuće i zamarajuće, čime se produljuje vrijeme pisanja te smanjuje produktivnost. Za pomoć pri osiguravanju kvalitetnog radnog toka, izbjegavanje pisanja ponavljajućeg kôda i uštede vremena, koriste se alati koji će to napraviti za nas. Takvi alati generiraju nama potrebni programski kôd.

Generiranje programskog kôda širok je pojam te ima veliko područje primjene. U ovom radu pokrivena će biti tema generiranja programskog kôda, kako i zašto je krenulo te koje se prednosti i nedostaci mogu izdvojiti. Generiranje omogućava programerima da prebace fokus i koncentraciju na važnije komponente rješenja, a ostave trivijalne dijelove generatorima. Rad tematizira izradu jednog takvog alata. Izrađeni alat generirat će programski kôd C# programskog jezika, a generirani kôd tvorit će klase tog programskog jezika. Izradu i generiranje programskog kôda omogućit će generator, odnosno kompilator Roslyn. Roslyn ili .NET platforma kompilatora je platforma koja nudi alate za analizu programskog kôda te generiranje istog. [1] Osim opisa njegove strukture, detaljno će se razraditi mogućnosti i načini generiranja i analize kôda. Kako bi se jasnije shvatio koncept generiranja kôda Roslyn API-jima prikazat će se nekoliko primjera i opis što se u kojem dijelu izvršava.

Razlog odabira ove teme prije svega je zainteresiranost u sadržaj teme i dublje razumijevanje kompilatora čiji se rad u svakodnevnim aktivnostima pisanja kôda uzima olako i često prođe neprimijećen. Upravo neprimijećenost govori koliko su kompilatori poput Roslyna ukorijenjeni u razvojne okoline i kvalitetno integrirani. Iz toga proizlazi zainteresiranost za shvaćanje ovih moćnih i korisnih alata te kako tijek rada pisanja programskog kôda može biti olakšan generiranjem i analizom kôda kompilatorom.

Za izrađeni alat napisana je specifikacija koje mogućnosti alat treba sadržavati, odnosno koje zahtjeve pokrivati. Za navedene zahtjeve dano je, prije svega, rješenje u obliku

dizajna softverskog rješenja koje bi trebalo objasniti arhitekturu na višoj razini apstrakcije za lakše razumijevanje i uvod u implementaciju samog alata. Implementacija alata, projekti i klase od kojih se rješenje sastoji, detaljno će biti opisani tekstualno i vizualno.

2. Metode i tehnike rada

Za razradu teme teorijskog dijela završnog rada korišteni su izvori s interneta i znanstvena literatura pronađena pomoću Google Scholar alata. Izvori su bilježeni pomoću aplikacije Mendeley te je završni rad pisan u Word aplikaciji.

Za ostvarenje praktičnog dijela rada korištena je .NET platforma, odnosno C# programski jezik, u razvojnoj okolini Visual Studio, a za verzioniranje programskog kôda GitHub repozitorij. Baza podataka koja sadrži testne tablice za pomoć pri razvoju alata je locirana u aplikaciji SQL Server Management Studio.

Rukovanje alatom, odnosno implementiranom ekstenzijom, provodi se kroz uporabu online platforme Visual Studio Marketplace za upravljanje verzijama i isporukom ekstenzije.

3. Generiranje programskog kôda

Pisanje programskog kôda zahtjevno je, dugotrajno i često se sastoji od ponavljajućeg kôda. Razlog tome je sintaksa programskih jezika koja je u mnogim jezicima slična govornome jeziku. Ona također zahtjeva pisanje po pravilima, a žele li programeri da je njihov programski kôd lako čitljiv i razumljiv, potrebno je pratiti standard pisanja. Pisanje takvog programskog kôda iziskuje vremena jer programeri imaju naviku pisati kôd osebujnim stilom koji može odudarati od standarda i pravila.

Vrijeme je bitan i vrijedan aspekt u rješavanju programskih problema jer rješenje često sa sobom nosi rok u kojem treba biti završeno. Navedeni elementi pisanja kôda motivirali su programere da pronađu rješenje. Jedan pristup koji ubrzava proces je generiranje programskog kôda.

Dakle, ideja generiranja programskog kôda nastala je iz potrebe programera za efikasnijim provođenjem vlastitog radnog toka. Generatori kôda nude rješenje za navedene probleme. [2]

3.1. Vrste generatora kôda

Generiranje programskog kôda započelo je skromno, 50-ih godina prošlog stoljeća, a ideja je ponikla iz ranih kompilatora. [3] Prvi primjer generatora kôda koji je moguće pronaći naziva se „*The Last One*“. Ovaj generator generirao je kôd programskog jezika BASIC. Ideja je bila pružiti korisniku odabir opcija kroz izbornike koje vode prema generiranju željenog programskog kôda. [4] Takav generator nije imao mnogo mogućnosti te nije uštedio dovoljno vremena da bi bio u većoj mjeri koristan. Ali postojanje mogućnosti ubrzanja pisanja kôda, odnosno smanjenja pisanja ponavljajućeg kôda potaknulo je ljude izraditi bolje alate.

Generatori su danas široko rasprostranjeni. Možemo ih podijeliti u dvije kategorije, aktivni i pasivni generatori. [5] Uobičajena okolina u kojoj se danas piše kôd su uređivači teksta i integrirana razvojna okruženja ili IDE (skraćeno od engl. *Integrated Development Environment*) poput Visual Studio, Visual Studio Code, IntelliJ IDEA i sl. Sva popularnija razvojna okruženja imaju implementirano, u većoj ili manjoj mjeri, generiranje programskog kôda. Generatori koji se u njima nalaze često spadaju u kategoriju pasivnih generatora. Oni pružaju generiranje osnovne strukture komponenti bez zadiranja u složenije generiranje programskog kôda. Osim toga, generirani kôd u daljnjem je razvoju programskog rješenja podložan promjeni, programeri ga mogu izmijeniti po svojoj volji ukoliko je to potrebno. [5]

Aktivni generatori, za razliku od pasivnih, mogu biti korišteni od strane programera tako da pokrenu generator više puta, izmjenjujući potrebne parametre kako bi dobili ažurirani kôd koji im je potreban. [5] Aktivni generatori zanimljivi su i izrazito korisni jer pružaju više mogućnosti od pasivnih, daju uvid u svoj rad te omogućavaju programerima značajnu kontrolu nad cijelim procesom generiranja programskog kôda. Postoji nekoliko podjela aktivnih generatora.

Kako Jack Herrington spominje u svojoj knjizi „*Code Generation In Action*“ [5], postoje mnoge podjele aktivnih generatora, a on ih je podijelio prema unosu (engl. *input*) i rezultatu (engl. *output*) unosa. Prvi od njih je pretvarač koda (engl. *code munger*), zatim proširivač ugrađenog kôda (engl. *inline-code expander*), generiranje miješanog kôda (engl. *mixed-code generation*), generiranje djelomičnog kôda (engl. *partial-class generation*), generiranje slojeva (engl. *tier of layer generation*) i posljednji jezik za cjelokupno područje primjene (engl. *full-domain language*). [5] Svaki od njih nudi različite rezultate te je korišten u različite svrhe. Kao što je već navedeno, aktivni generatori pružaju uvid u svoj proces generiranja, a kako različiti generatori kôda funkcioniraju može se saznati kroz upoznavanje tehnika generiranja.

Generiranje programskog kôda korisno je tijekom razvoja softvera te se upravo u razvoju većih i složenijih procesa najčešće koristi. Različiti pristupi razvoju softvera zahtijevaju različite načine generiranja programskog kôda. Zbog toga je nastalo nekoliko tehnika generiranja koje su dio cjelokupnog procesa razvoja softvera. U sadržaju dokumentacije razvojnog procesa modeli i dijagrami česta su pojava iz razloga što su dobra vizualizacija konačnog rješenja na višoj razini apstrakcije. Vođeni tom činjenicom, generiranje temeljeno na modelima jedna je od tehnika generiranja programskog kôda.

U procesu razvoja temeljenog na modelima ili MDD (skraćeno od engl. *Model-driven Development*) i inženjerstva temeljenog na modelima ili MDE (skraćeno od engl. *Model-driven Engineering*) mogu se pronaći generatori koji generiraju programski kôd temeljen na modelima ili dijagramima iz specifikacije [3].

Generiranje kôda temeljeno na modelima ili MDCG (skraćeno od engl. *Model-Driven Code Generation*) dio je MDE, i kao što naziv upućuje, kôd se generira prema modelima. Jedan od najpoznatijih primjera takvih modela su UML (skraćeno od engl. *Unified Modeling Language*) modeli, odnosno dijagrami. [6]

Generiranje kôda temeljeno na predlošku ili TBCG (skraćeno od engl. *Template-based Code Generation*) je tehnika sinteze. Tehnika sinteze označava da se generiranje kôda izvršava prema specifikaciji više razine. Specifikacija više razine u ovoj tehnici bili bi predlošci. TBCG također je dio MDE. Budući da je tijekom izrade specifikacije manja vjerojatnost pojavljivanja pogrešaka nego tijekom procesa razvoja programa, te je korisniku programa

lakše razumjeti modele i specifikacije od programskog kôda, ovo je najveća prednost takvog generiranja kôda. [3]

Način generiranja koji odskače od generiranja temeljenog na specifikaciji je generator napisan u programskom jeziku opće namjene. Takvo generiranje zasnovano je na spajanju niza znakova (engl. *string*). Razina kompleksnosti takvih generatora i njegovih mogućnosti ovisi o odabiru programskog jezika. Implementacija generatora takve namjene olakšana je korištenjem API-ja. [7]

API, kako Jorges S. spominje, u takvom slučaju nudi pomoć za smanjenje kompleksnosti i nudi sve potrebne koncepte za generiranje kôda poput klasa i metoda. [7] Jedan od takvih API-ja je Roslyn čije će se mogućnosti detaljnije razraditi u sljedećem poglavlju. Nedostatak generiranja programskog kôda ovim putem je što takvi generatori ne nude puno mogućnosti, povoljni su samo u scenarijima u kojima nije potrebno generirati mnogo kompleksnog kôda jer ih je teže implementirati i održavati.

3.2. Prednosti i nedostaci generiranja kôda

Očita prednost generiranja programskog kôda je efikasnost pisanja kôda. Potreban programski kôd može se generiranjem ostvariti uz minimalan unos programera što donosi značajnu uštedu vremena. Generatori koji koriste umjetnu inteligenciju mogu sami prepoznati koji dio kôda bi bio prikladan s obzirom na kontekst i strukturu. Uštađeno vrijeme može se zauzvrat iskoristiti na drugim područjima poput planiranja rješenja i rezultata programiranja. Samim time povećava se produktivnost i kvaliteta rada. Programeri mogu razmisliti i provesti unaprjeđenje kvalitete kôda i pisati ga imajući na umu ponovnu iskoristivost programskog kôda.

Generiranje programskog kôda nudi i povećanje produktivnosti te smanjenje broja grešaka. Budući da su generatori kôda implementirani tako da je standard pisanja kôda prioritet, generirani kôd omogućava navedeno unaprjeđenje, ali i lakše razumijevanje strukture softvera prilikom prvog susreta s generiranim kôdom novim članovima tima na projektnim zadacima.

Osim lakšeg razumijevanja, generirani kôd osigurava konzistentnost. Programeri mogu biti sigurni kako će dobiti kôd strukturiran na način na koji su navikli. Čestom korištenju generatora pridonijelo je lako korištenje takvih alata. Intuitivni su te lako dostupni što ih čini vrijednim sredstvom za manje iskusne programere koji kroz njih mogu učiti o pravilima te ubrzati svoj rad. No, s tim dolazi rizik pretjeranog korištenja i korištenja bez provjere je li

generirani programski kôd valjan. Može utjecati na smjer razvoja programskog rješenje što rezultira boljim ili lošijim ishodom.

Iako je generiranje programskog kôda uvelike napredovalo u posljednjih nekoliko godina, i dalje postoji mnogo nedostataka. Generirani programski kôd može uštedjeti vrijeme ukoliko je napisan korektno, bez pogrešaka. Pogreške koje možemo očekivati pri generiranju kôda uglavnom su pogreške logičkog tipa. Ako je kôd jednostavan i generiraju se osnovne komponente poput kostura petlji i klasa, očekujemo rijetke pogreške ili nijednu s obzirom na standard koji današnji generatori kôda pružaju.

Kad se generatorima prepusti generiranje kompleksnijih komponenti, vrlo često se taj kôd ne može iskoristiti, to jest mora se prilagoditi što opet oduzima vrijeme. Ukoliko se odluči iskoristiti generirani kôd bez da se primijete greške, ono može produljiti utrošeno vrijeme te smanjiti produktivnost. Zato je opreznost prilikom korištenja ključna.

Generirani kôd može donijeti mnogo korisnog te je u određenim procesima razvoja neophodan za rad, ali nosi sa sobom i pogreške koje mogu u potpunosti negirati dobrobiti. Osim kôda s logičkim pogreškama, laka dostupnost generiranju može dovesti do neumjerenog korištenja generatora što može prouzročiti negativne posljedice.

4. .NET Compiler Platform („Roslyn“)

Iako je službeni naziv .NET Compiler Platform, ovaj projekt poznatiji je pod nazivom „Roslyn“. Projekt uključuje veći broj kompilatora otvorenog izvornog kôda (engl. *open-source*) i API-ja za analizu kôda za C# i Visual Basic programske jezike. [8] Razvoju Roslyn projekta na službenom GitHub repozitoriju doprinijelo je nekoliko stotina programera, uključujući članove tima tvrtke Microsoft i vanjskih suradnika. [9]

Kroz poglavlja koja slijede sažeto će se opisati ideja koja je potakla Roslyn projekt te njegova povijest, a detaljnije su razmotrene mogućnosti Roslyn kompilatora. Kroz primjere će biti prikazati osnovni slučajevi korištenja prilikom generiranja programskog kôda.

4.1. Povijest Roslyn projekta

Prema M. Torgersenu [10], o projektu nalik Roslynu govorilo se unutar tvrtke Microsoft nekoliko godina prije njegova početka. Kompilatori za C# i Visual Basic programske jezike bili su napisani u C++ programskom jeziku i zahtijevali su promjenu. Kako M. Torgersen navodi, odluka za pisanjem novih kompilatora nije donesena zbog poznatih izazova prilikom kreiranja istih, ali i tadašnjeg intenzivnog rada na operacijskom sustavu Windows 8. Također, Microsoft tad nije bio poznat po kreiranju projekata otvorenog kôda, odnosno softvera čiji je izvorni kôd dostupan svima za čitanje i izmjenu.

Nakon završetka rada na Windows 8 sustavu, Microsoft odlučuje krenuti u smjeru otvorenog kôda te suočiti se s izazovima koji su ranije sprječavali razvoj željenih kompilatora. Roslyn je bio projekt koji je donio prekretnicu. Cilj projekta bio je napraviti kompilatore otvorenog kôda te ih razviti u C# programskom jeziku. [10]

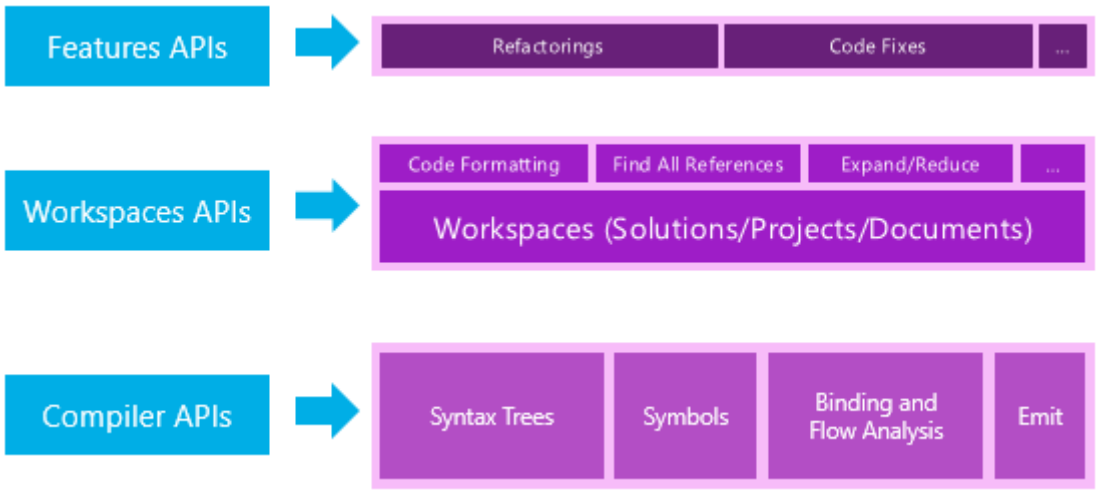
Za razliku od Roslyna, tradicionalni kompilatori radili su na principu crne kutije (engl. *black box*) [11]. Ni u kojem trenutku izvođenja kompilacije programa nije postojala mogućnost trenutnog pregleda onog što se unutar njega odvija niti se znao sadržaj izvornog kôda. Nakon što bi kompilator završio s kompilacijom, rezultat njegova rada bio bi prikazan u obliku izvršne datoteke, a put koji vodi tom rezultatu ostao bi nepoznat. [11]

Motivacija za razvoj kompilatora koji nudi takvu količinu transparentnosti je jasna. Kompilator sa navedenim svojstvima, koji do tad nisu bili uobičajeni, donio bi veliku korist korisnicima i dopustio vanjskim suradnicima da pridonesu stvaranju poboljšanih verzija i kreiranju novih funkcionalnosti. Za razliku od prijašnjih kompilatora, Roslyn nudi upravo sve navedeno te daje korisniku mogućnost za uvid i interakciju s cijelim procesom kompilatora. [12]

Početkom travnja 2014. godine, Roslyn je objavljen na Codeplexu kao javno dostupan projekt otvorenog kôda. [10] Od tad, Roslyn je u cijelosti prebačen na GitHub repozitorij. Osim napretka kompilatora, mogućnost otvorenog kôda donijela je programskom jeziku C# mnoge funkcionalnosti koje su, s pristupom izvornom kôdu, mogli razviti i implementirati vanjski suradnici. [10]

4.2. Struktura API slojeva Roslyn kompilatora

Roslyn se sastoji od dva glavna sloja API-ja, API-ji kompilatora (engl. *Compiler APIs*) i API-ji radnih prostora (engl. *Workspaces APIs*). Dodatan sloj naziva se API-ji funkcionalnosti (engl. *Features APIs*). [11]



Slika 1: API slojevi (izvor: [11])

Prvi sloj, API-ji kompilatora tvoreni su od sintaksnog stabla, modela simbola, analize povezivanja i izdavanje. On sadrži objektne modele koji su u skladu s informacijama svake faze cjevovoda kompilatora. Osim toga sadrži nepromjenjivu snimku (engl. *snapshot*) jednog poziva (engl. *invocation*) kompilatora. Taj poziv uključuje reference na sklopove, opcije kompilatora i datoteke izvornog kôda. Roslyn razlikuje dva API-ja na ovoj razini, jedan za C# programski jezik, a drugi za Visual Basic. API-ji za dijagnostiku (engl. *Diagnostic APIs*) i API-ji za skriptiranje (engl. *Scripting APIs*). dio su API-ja kompilatora. Kompilator može tijekom svoje analize generirati određenu količinu dijagnostike koja uključuje sintaksu, semantiku, pogreške, upozorenja i informacije. Toj dijagnostici može se pristupiti kroz API-je za dijagnostiku. API-ji također omogućavaju korisnicima uključivanje vlastitih analizatora u kompilaciju kako bi se dijagnostika definirana s njihove strane generirala zajedno s ostalom dijagnostikom. Korist API-ja za dijagnostiku moguće je vidjeti kroz razne alate unutar razvojne okoline Visual Studio.

Dijagnostika koju API-ji pružaju pomaže korisnicima prikazati pogreške tako da se problematične linije u kôdu označe valovitim linijama te sugeriraju rješenja koja bi riješila nastale pogreške. [11] API-ji za skriptiranje koriste se pri izvođenju isječaka kôda i za prikupljanje konteksta izvršavanja za vrijeme izvođenja programa. Ove API-je koristi REPL (skraćeno od engl. *Read-Eval-Print-Loop*). [11] Detaljniji opis strukture ovog sloja slijedi u sljedećem potpoglavlju.

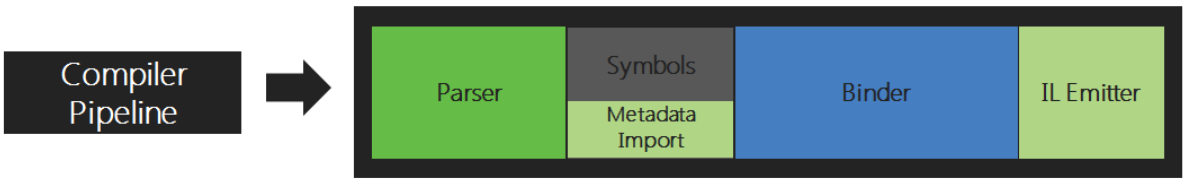
Sloj radnih prostora (engl. Workspaces layer) sadrži API-je radnih prostora. Navedeni se API-ji koriste za upravljanje rješenjem, projektima unutar rješenja te za analizu kôda i refaktoriranje cijelog rješenja. Oni sastavljaju sve informacije o projektima koji se nalaze u rješenju u objektni model. Radeći to, omogućavaju korisnicima pristup modelima objekata sloja kompilatora bez parsiranja datoteka, konfiguracije opcija te upravljanje ovisnostima između projekata. [11]

Alati za analizu kôda i refaktoriranje koji se nalaze u razvojnoj okolini Visual Studio implementirani su korištenjem API-ja koje izlaže sloj radnih prostora, a uključeni u to su API-ji generiranja kôda. [11]

4.3. Struktura API-ja kompilatora

Struktura sloja API-ja kompilatora nalikuje onoj u tradicionalnim kompilatorima. Prema Roslyn službenoj dokumentaciji [11], Roslyn daje pristup analizi kôda pisanog u C# ili Visual Basic programskom jeziku putem API sloja koji oponaša tradicionalni proces kompilacije.

Na sljedećoj slici moguće je vidjeti prvi dio strukture Roslyn API-ja kompilatora, odnosno cjevovod, te redoslijed kojim se proces unutar njega odvija.



Slika 2: Cjevovod kompilatora (izvor: [11])

Srž Roslyn API-ja kompilatora sastoji se od četiri manja, povezana dijela. Svaki dio je zasebna komponenta čiji je završni rezultat ulaz sljedećoj komponenti. Na početku procesa u kompilatoru se odvija faza analize (engl. *parse phase*) koja sadrži parser koji analizira kôd za kompilaciju. Kôd se parsira u tokene i sintaksu programskog jezika. Ovu fazu moguće je prikazati koristeći klasu *SyntaxFactory*, koja će detaljno biti objašnjena u nastavku, i njezinu

metodu *ParseTokens*. Metoda *ParseTokens* uzima varijablu tipa *string* kao parametar i iz njezinog sadržaja dijeli znakove u tokene. Kôd vlastite metode *ParserPhase* je sljedeći:

```
private void ParserPhase()
{
    string code = @"
        using System;

        namespace RoslynParserPhase
        {
            public class Parser
            {
                public void ParserMethod()
                {
                }
            }
        }";

    var tokens = SyntaxFactory.ParseTokens(code);
    tokens.ToList().ForEach(t =>
    {
        Console.WriteLine(t.Text);
    });
}
```

U konzolu se ispisuju tokeni jedan ispod drugog.

```
using
System
;
namespace
RoslynParserPhase
{
public
class
Parser
{
public
void
ParserMethod
(
)
{
}
}
}
```

Slika 3: Ispis tokena u konzoli

Druga faza je tzv. faza deklaracije (engl. *declaration phase*), gdje se deklaracije iz izvornog kôda i uvezenih metapodataka analiziraju kako bi se formirali imenovani simboli. [11] Na sličan način kao i za prvu, moguće je rad ove faze prikazati kroz jednostavnu metodu *SymbolsPhase*.

```
private void SymbolsPhase()
{
    string code = @"
        using System;

        namespace RoslynSymbolsPhase
        {
            public class Symbols
            {
                public int SymbolsProperty { get; set; }

                public void SymbolsMethod()
                {
                }
            }
        }";
    SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(code);
    var compilation = CSharpCompilation.Create("MyCompilation")

.AddReferences(MetadataReference.CreateFromFile(typeof(object).Assembly.
Location))
        .AddSyntaxTrees(syntaxTree);

    var semanticModel = compilation.GetSemanticModel(syntaxTree);
    var classDeclaration = syntaxTree.GetRoot()
        .DescendantNodes()
        .OfType<ClassDeclarationSyntax>()
        .FirstOrDefault();

    if (classDeclaration != null)
    {
        foreach (var member in classDeclaration.Members)
        {
            if (member is MethodDeclarationSyntax methodDeclaration)
            {
```

```
                var methodSymbol =
semanticModel.GetDeclaredSymbol(methodDeclaration);
                Console.WriteLine($"Method: {methodSymbol.Name}");
            }
            else if (member is PropertyDeclarationSyntax
propertyDeclaration)
            {
                var propertySymbol =
semanticModel.GetDeclaredSymbol(propertyDeclaration);
                Console.WriteLine($"Property: {propertySymbol.Name}");
            }
        }
    }
}
```

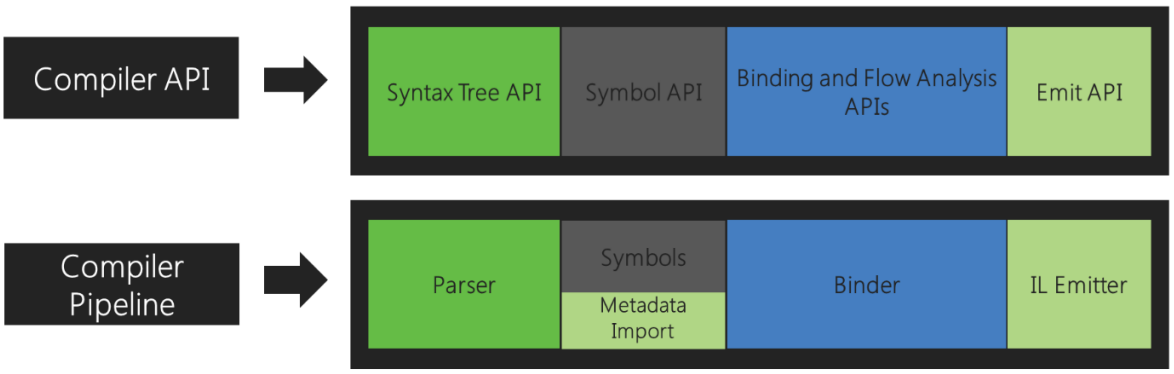
Ova metoda također sadrži varijablu tipa *string* kao metoda *ParserPhase*, u kojoj se nalazi kôd klase s jednim svojstvom i jednom metodom. Upravo ova metoda traži simbole svojstva i metode unutar danog kôda klase i ispisuje u konzolu njihove nazive. U deklaraciji klase, koja je tipa *ClassDeclarationSyntax*, pretražuju se njezini članovi kako bi se pronašle deklaracije metode i svojstva. Budući da kôd klase sadrži oba člana, konzola sadrži sljedeći ispis:

```
Property: SymbolsProperty
Method: SymbolsMethod
```

Slika 4: Ispis pronađenih simbola iz danog kôda

Pretposljednja faza naziva se faza povezivanja (engl. *bind phase*). Tijekom ove faze identifikatori u kôdu usklađuju se sa simbolima. Naposlijetku dolazi faza izdavanja (engl. *emit phase*) tijekom koje se prikupljaju svi rezultati prijašnjih faza i izdaju se u obliku sklopova (engl. *assembly*). [11]

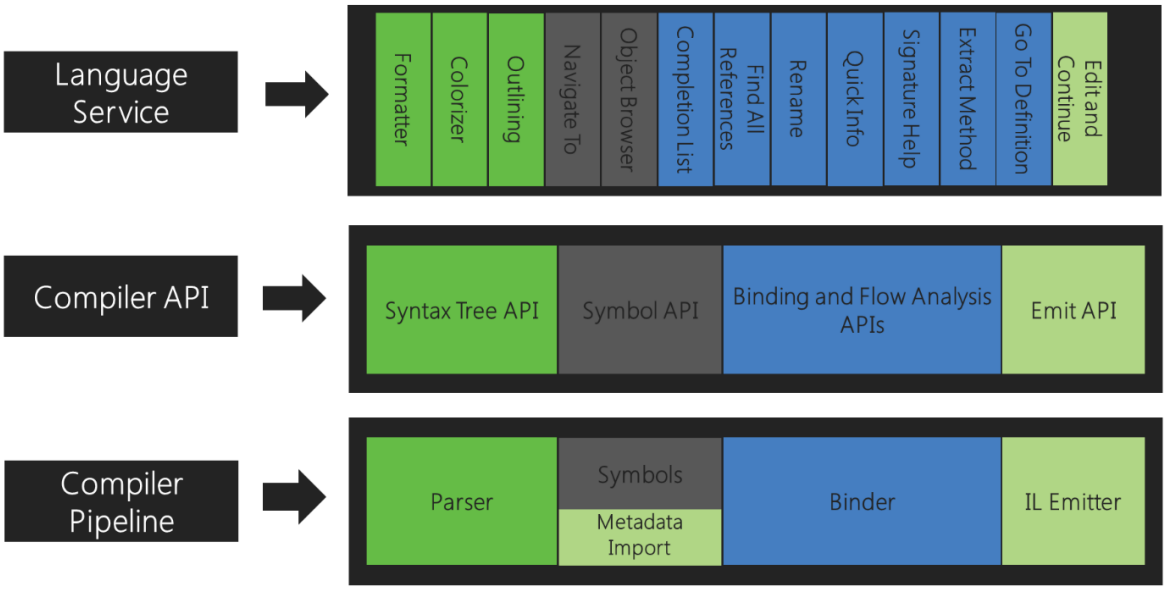
Na sljedećoj slici uspoređeni su cjevovodi kompilatora i sloja Roslyn API kompilatora koji ga oponaša i služi korisnicima kao pristup kompilatoru.



Slika 5: Cjevovod kompilatora i struktura API-ja kompilatora (izvor: [11])

Kao što se da primijetiti gledajući sliku, svaka faza koju API kompilator sadrži u skladu je s fazama cjevovoda kompilatora. Ovo je drugi dio API-ja kompilatora zadužen za komunikaciju s prvim dijelom. Za svaku fazu cjevovoda kompilatora izlaže se objektni model u API-ju kompilatora koji daje pristup informacijama u toj fazi. Faza analize izlaže se kao sintaksno stablo (engl. *syntax tree*) i unutar API-ja dobiva naziv „Syntax Tree API“. Faza deklaracije predstavljena je kao hijerarhijska tablica simbola te se naziva „Symbol API“. Treća faza po redu, faza povezivanja, u API-ju kompilatora sad je „Binding and Flow Analysis APIs“. Ona se izlaže u objektnom modelu kao rezultat semantičke analize kompilatora. Posljednji dio API kompilatora je „Emit API“ i u njemu se faza izdavanja izlaže kao API za generiranje IL (skraćeno od engl. *Intermediate Language*) bajtnog kôda. [11]

Treći i posljednji dio Roslyn kompilatora jest jezična usluga (engl. *Language Service*). Gornji sloj Roslyn kompilatora sadrži najviše komponenti. Na slici su prikladnim bojama prikazani dijelovi, odnosno komponente ove usluge koje koriste određene dijelove API-ja kompilatora. API je i u ovome slučaju korišten za komunikaciju, sada za komunikaciju između cjevovoda kompilatora i jezične usluge. [11]



Slika 6: Potpuna struktura Roslyn kompilatora (izvor: [11])

Prvi dijelovi jezične usluge, obojeni zeleno, dijelovi su zaduženi za strukturiranje i oblikovanje programskog kôda. Oni koriste sintaksno stablo kako bi oblikovali kôd na prepoznatljiv način unutar uređivača teksta, to jest kôda u razvojnoj okolini. Preglednik objekata (engl. *Object Browser*) i značajke zadužene za navigaciju koriste tablicu simbola. Mnoge značajke razvojne okoline koriste API koji izlaže objektni model treće faze kompilatora. Te značajke uključuju „Idi na definiciju“ (engl. *Go To Definition*), preimenovanje, pronalazak referenci, refaktoriranje i koriste semantički model. Na kraju je značajka „Uredi i nastavi“ (engl. *Edit and Continue*) koja koristi sve navedene, uključujući API za izdavanje. [11]

Svaki od tri opisana dijela zajednički tvore sloj API-ja kompilatora. Međusobnom komunikacijom ovih dijelova ostvarene su mnoge korisne značajke razvojne okoline Visual Studio. Roslyn omogućava analizu kôda tijekom pisanja, oblikuje ga te daje mogućnost pisanja vlastitih pravila pisanja i značajki kroz ovaj sustav.

4.4. Generiranje kôda korištenjem Roslyna

Roslyn je moćan alat, ali kad je u pitanju generiranje kôda, on je u suštini ograničen na manipulaciju sintaksom kôda. Njegove mogućnosti generiranja programskog kôda svode se na rukovanje različitim tokenima. Svaki token ima svoje mjesto u kôdu i svoje značenje. Tokeni su sve skupine znakova ili samostalni znakovi koji nose značenje, uključujući nazive klasa i njezinih članova, razmaci, zarezi. Slaganjem tokena u smislenu cjelinu, dobiva se sintaksno stablo iz kojeg proizlazi funkcionirajući programski kôd. Roslyn može zatim kreirati nove

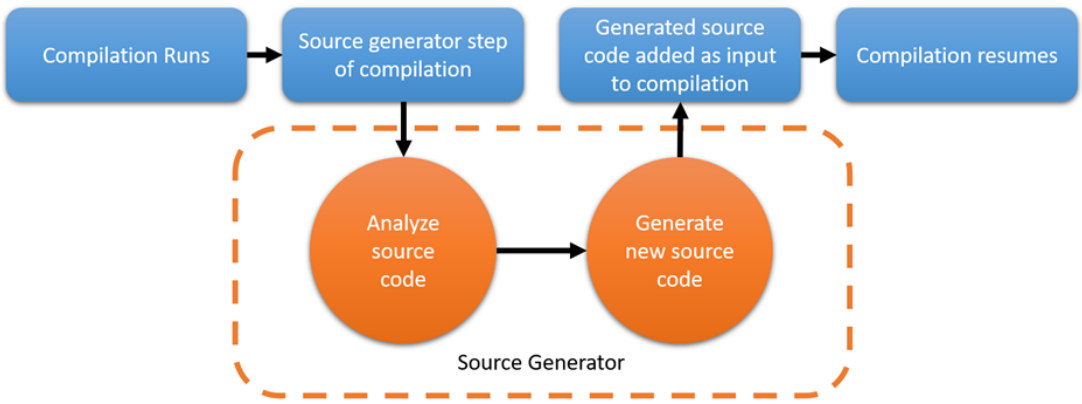
sintaksne čvorove, ukloniti ih i time stvoriti novi programski kôd. Koristeći redoslijed i sadržaj tokena, Roslyn prepoznaje strukturu kôda. Na primjer, pomoću tokena kao što je znak točka sa zarezom, Roslyn prepoznaje da linija kôda završava s tim tokenom. Dakle, iz tokena izvlači kontekst i osigurava ispravne semantičke i sintaktičke vrijednosti programskog kôda. Ovim alatom omogućeno je generiranje isječaka kôda, ali i kompleksnijih struktura. Detaljniji opis strukture sintakse, kao i različitih generatora sadržan je u poglavljima koja slijede.

4.4.1. Generatori izvornog kôda

Generatori izvornog kôda (engl. *Source Generators*) pružaju programerima mogućnost da pregledavaju korisnički kôd dok se kompilira. Takav generator može dinamički stvarati izvorne datoteke koje se dodaju u korisničku kompilaciju. S obzirom na to, postoji kôd koji se izvršava tijekom kompilacije i provjerava program kako bi dodao izvorne datoteke koje se mogu kompilirati s ostatkom kôda. [13]

Prema službenoj dokumentaciji s GitHub repozitorija [13], generatori izvornog kôda omogućavaju programerima da dohvate objekt kompilacije (engl *compilation object*) koji predstavlja sav korisnički kôd koji se kompilira. Pomoću ovog objekta moguće je pisanje kôda koji radi s modelima sintakse i semantike za kôd koji se kompilira. [13] U dokumentaciji je kao druga važna stvar navedeno već opisano dodavanje izvornih datoteka. Te dvije stvari čine generatore izvornog kôda vrlo korisnim dodatkom programerima C# programskog jezika.

Na sljedećoj slici vidljive su faze kompilacije kôda. Na njoj su bojama odvojene aktivnosti kompilacije od onih aktivnosti generatora izvornog kôda. Počevši od aktivnosti označenim plavom bojom, prvo je vidljiv početak kompilacije. U sljedećoj aktivnosti dolazi korak kompilacije generatora. Nakon analize izvornog kôda, generator generira novi izvorni kôd te se u sljedećoj aktivnosti izvornome kôdu kao unos dodaje generirani kôd i kompilacije nastavlja dalje uobičajeno.



Slika 7: Dijagram radnog toka generatora kôda (izvor: [13])

Područja koja bi generatori izvornog kôda mogli poboljšati pronalaze su u pristupima koja danas postoje za inspekciju korisničkog kôda i generiranja informacija ili kôda prema odrađenoj analizi. Prvi pristup koji danas to programerima omogućava je refleksija tijekom izvođenja (engl. *Runtime reflection*).

Refleksija tijekom izvođenja korisna je prilikom pokretanja aplikacije jer se tad može analizirati korisnički kôd i iskoristiti prikupljene informacije kako bi se generirale potrebne stvari. Koristeći generatore izvornog kôda analiza bi se mogla izvršiti tijekom kompilacije. Generator bi u tom slučaju analizirao izvorni kôd i generirao kôd potreban za povezivanje aplikacije što bi rezultiralo kraćim vremenom pokretanja. Analiza i generiranje odradila bi se tijekom kompilacije, a ne u vremenu pokretanja.

U Microsoft službenoj dokumentaciji područja koja se još spominju su korištenje MSBuild zadataka (engl. *Juggling MSBuild tasks*) te presijecanje IL kôda (engl. *Intermediate Language weaving*) kojima bi u budućnosti moglo smanjiti potrebno vrijeme izvođenja pomoću generatora.

Implementacija ovih generatora moguća je kroz sučelje „ISourceGenerator“, ali preporučeno je zamijeniti je implementacijom inkrementalnih generatora opisanih u nastavku. [14]

4.4.2. Inkrementalni generatori

Inkrementalni generatori novi su API koji postoje uz generatore izvornog kôda. Ovi generatori nude mogućnost korisnicima da odrede strategije generiranja koje se mogu primjenjivati na učinkovit način putem sloja za postavljanje (engl. *hosting layer*). [15]

Cilj dizajna koji vrijedi spomenuti, koji se nalazi u službenoj dokumentaciji [15], je podržati generiranje više elemenata osim izvornog kôda. Također, implementacija generatora izvornog kôda zastarjela je te se preporučuje implementacija putem sučelja inkrementalnog generatora naziva „IIncrementalGenerator“.

4.4.3. Struktura sintakse

Prije generiranja, potrebno je prikazati dijelove koji generiranje kôda omogućavaju. Osnovna podatkovna struktura koju izlaže API kompilator je sintaksno stablo. Sintaksna stabla reprezentiraju leksičku i sintaktičku strukturu izvornog kôda. Njihovu strukturu čine čvorovi, tokeni i ostali konstrukti koji se mogu pronaći u kôdu, kao što su komentari, razmaci i prazne linije. Njihova svrha može se podijeliti u dva dijela, a oba se dotiču alata razvojne okoline. [11] Prvi dio je omogućavanje alatima kao što su alati za analizu kôda i refaktoriranje da vide i

obrađuju sintaktičku strukturu izvornog kôda u projektu. Drugi dio bio bi omogućavanje alatima da kreiraju i rade izmjene na izvornom kôdu bez izravnih izmjena teksta. Alati to mogu kreirajući i mijenjajući stabla. [11]

Sintaksna stabla strukture su korištene za kompilaciju, analizu kôda, povezivanje, refaktoriranje, funkcionalnosti razvojne okoline i generiranje kôda. [11] Sintaksna stabla imaju tri glavne značajke.

Prva značajka je da sintaksna stabla zadržavaju sve informacije izvornog kôda bez gubitaka, odnosno u njima se nalaze sve informacije točno onako kako su zapisane u kôdu. To uključuje svaki dio teksta, kao i komentare i direktive. Pogreške koje se nalaze u izvornome kôdu prikazuju se u sintaksnom stablu kao preskočeni tokeni, to jest tokeni koji nedostaju. [11]

Promjene izvornog teksta moguće je napraviti pomoću sintaksnog stabla, što je druga bitna značajka. Ako se uzme postojeće stablo i na temelju njega napravi novo koje sadrži promjene, tekst je izmijenjen. [11]

Treća značajka koju posjeduju sintaksna stabla je ta da se promjene ne mogu vršiti direktno na njima, točnije na njihovim snimkama. Te snimke sadrže stanje kôda i one se ne mijenjaju već se tijekom promjena stvaraju nove snimke. Snimke može koristiti više korisnika u isto vrijeme bez zaključavanja (engl. *locking*) i dupliciranja. [11]

Sintaksni čvorovi (engl. *Syntax Nodes*) elementi su koji služe za izgradnju sintaksnih stabla. Oni predstavljaju deklaracije, naredbe, klauzule i izraze, što su sintaktički konstrukti u kôdu. U kôdu, svaki navedeni konstrukt predstavljen je zasebnom klasom iz klase *SyntaxNode*. Kao i ostale strukture podataka tipa stabla, sintaksni čvorovi imaju čvor roditelja i čvorove djecu.

Sintaksni tokeni (engl. *Syntax Tokens*) manje su sintaktičke tvorevine od sintaksnih čvorova. Tokeni se nalaze na krajevima grana stabla i nikad nisu roditelji drugim čvorovima ili tokenima. Oni se sastoje od ključnih riječi (engl. *keywords*), identifikatora (engl. *identifiers*), literala i interpunkcijskih znakova. [11]

Tokeni se razlikuju po svojim svojstvima koji sadrže podatke o njima. Dakle, svi su tokeni iste strukture, ali element teksta koji predstavlja može se razlikovati pomoću sadržaja svojstava. [11]

Sintaksna trivijalnost (engl. *Syntax Trivia*) uključuje sve dijelove kôda, to jest teksta koji nemaju značajnu vrijednost za razumijevanje kôda. U ovu kategoriju spadaju razmaci, tabulatori, linije bez kôda, komentari i predprocesorske direktive. [11]

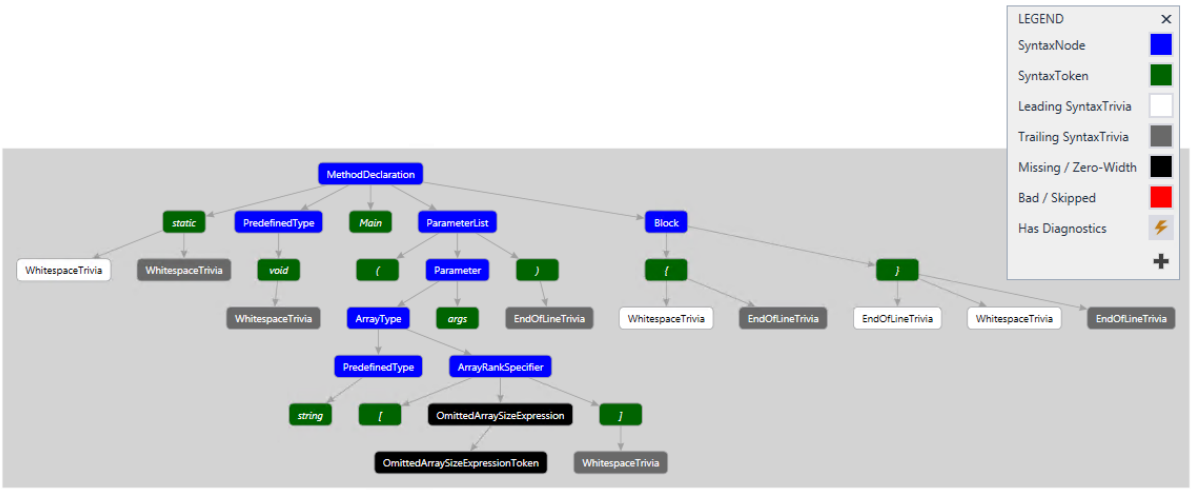
Ovi elementi uključeni su u sintaksno stablo jer su dio sintakse kôda, ali ne kao čvorovi stabla. Tokeni sadrže kolekcije kao dio svojih svojstava u kojima se zapisuje sintaksna

trivijalnost. Isto kao tokeni, ova sintaksna kategorija je vrijednosnog tipa i sve elemente opisuje isti tip, a to je *SyntaxTrivia* tip. [11]

Još jedno bitno svojstvo koje čvorovi i tokeni imaju jest raspon (engl. *Span*). Svaki element sintaksnog stabla ima poziciju i broj znakova od kojih se sastoji. Oba svojstva iskazana su cjelobrojnim (engl. *integer*) tipom podataka, točnije 32-bit integer tip. Pomoću raspona, svaki element zna na kojem mjestu počinje i koliko mjesta zauzima, odnosno zna svoju poziciju u tekstu. [11]

Kako bi se raspoznali različiti elementi sintaksnog stabla, postoji i svojstvo tipa ili vrste (engl. *Kind*). To je svojstvo tipa *System.Int32* kojim se identificiraju pojedini sintaksni elementi. [11]

Kôd često sadrži sintaktičke pogreške i u tim slučajevima u koristi dolazi svojstvo raspona. Ako se tijekom parsiranja teksta primijeti da nedostaje token, on se može unijeti u sintaksno stablo na lokaciju gdje je očekivan. Umetnutom tokenu svojstvo raspona je prazno kako bi se označila pogreška sintakse. Osim toga, svojstvo *IsMissing* vraća *bool* vrijednost točno (engl. *true*).



Slika 8: Sintaksni graf u izradi alata Syntax Visualizer (izvor: [16])

Graf koji se nalazi na slici iznad napravljen je pomoću alata Syntax Visualizer koji pomaže vizualno prikazati sintaksna stabla i sve dijelove od kojih je konstruiran. Na grafu su bojama razlikuju sintaksni čvorovi, tokeni i ostala sintaksa koja se može naći u sintaksnom

stablu. Na takav je način izvorni kôd spremljen te se na sličan način slažu elementi prilikom generiranja programskog kôda.

Kompilacija (engl. *Compilation*) je struktura koja predstavlja sve potrebno za kompilaciju programa napisanog u C# ili Visual Basic programskom jeziku. Kompilacija uključuje reference na sklopove (engl. *assembly reference*), opcije kompilatora i izvorne datoteke. [11]

Svaki deklarirani tip, član ili varijabla koja se nalazi u izvornom kôdu, kompilacija predstavlja kao simbol (engl. *symbol*). Za lakše pronalaženje i povezivanje simbola i elemenata u izvornome kôdu ili simbola iz metapodataka sklopova, postoje mnoge metode koje u tome pomažu te se nalaze u sklopu kompilacije. [11]

Kao što su sintaksna stabla nepromjenjiva (engl. *immutable*) i mogu se jedino stvarati nova koja tad sadrže promjene, na isti način su zamišljene kompilacije. Za bilo kakvu željenu promjenu, nova kompilacija se kreira i u nju se spremaju promjene. [11]

Simboli (engl. *Symbols*) su važni elementi izvornog kôda jer predstavljaju sve što se u njemu nalazi. Simboli obuhvaćaju i predstavljaju svaki imenski prostor, tip, poput klase i sučelja, metodu, polje (engl. *field*), događaj (engl. *event*), parametar ili varijablu koja se u kôdu može pronaći. [11]

Svaki tip simbola predstavlja njegovo zasebno sučelje izvedeno iz sučelja *ISymbol* te ono sadrži informacije koje su kompilatoru potrebne za rad. Dakle, za razliku od elemenata sintaksnog stabla koje je odvajalo svojstvo, simboli su zasebni tipovi, to jest vrste.

4.4.4. Generiranje kôda korištenjem klase SyntaxTree

SyntaxTree je klasa u čije je varijable moguće spremiti sintaksno stablo. Za kreiranje sintaksnog stabla napisanog u C# programskom jeziku koristi se klasa *CSharpSyntaxTree*. Njezina metoda *Create* kreira sintaksno stablo. Metodi se prosljeđuje varijabla tipa *SyntaxNode* što znači da se stablo kreira iz proslijeđenog sintaksnog čvora. Za prikaz korištenja navedene metode potrebno je koristiti klasu *SyntaxFactory* za kreiranje sintaksnog čvora. U primjeru koji slijedi klasa će biti korištena za kreiranje jednostavnog čvora, a detaljnije objašnjenje klase *SyntaxFactory* nalazi se u sljedećem poglavlju. Osim kreiranja, moguće je parsirati tekst koji se zatim sprema u sintaksno stablo metodom *ParseText*. U sljedećim primjerima prikazat će se korištenje metoda *Create* i *ParseText*. Rezultat njihova rada spremit će se u varijablu tipa *SyntaxTree*.

Prvi primjer prikazuje način korištenja metode *ParseText*. Ova metoda je jednostavnija od metode *Create* jer ne zahtijeva korištenje druge klase već samo pisanje kôda kao niz znakova.

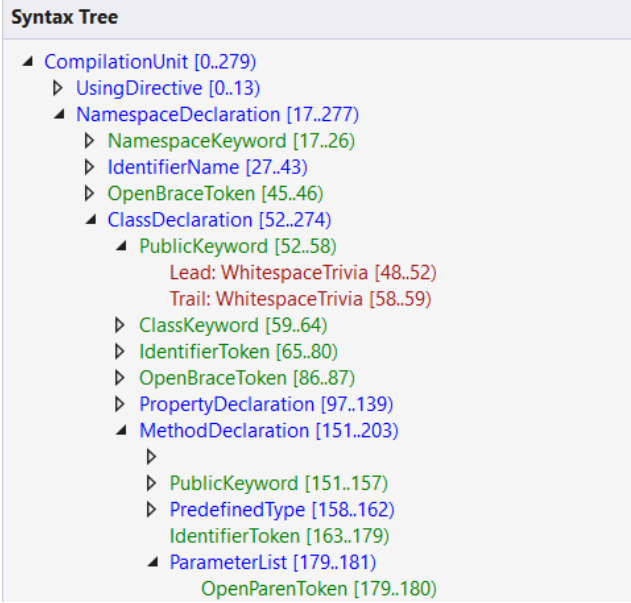
```
SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(@"
                using System;

                namespace RoslynSyntaxTree
                {
                    public class GeneratedSyntax
                    {
                        public int GeneratedProperty { get; set; }

                        public void GeneratedMethod1()
                        {
                        }

                        public void GeneratedMethod2()
                        {
                        }
                    }
                }"
            );
```

U prvoj liniji deklarira se varijabla *syntaxTree* u koju se sprema parsirani tekst klase. Napisana klasa sadrži osnovne elemente koji se mogu pronaći u datoteci klase C# programskog jezika. Sadrži using direktivu, imenski prostor, deklaraciju klase, svojstvo i metode. Prilikom ispisa varijable *syntaxTree* metodom *ToString* u konzolu, dobivamo sljedeći rezultat:

```
using System;

namespace RoslynSyntaxTree
{
    public class GeneratedSyntax
    {
        public int GeneratedProperty { get; set; }

        public void GeneratedMethod1()
        {
        }

        public void GeneratedMethod2()
        {
        }
    }
}
```

Slika 9: Ispis varijable *syntaxTree*

U ispisu vidimo strukturu koja je ista onoj napisanoj u primjeru. Ukoliko se želi provjeriti što je upisano u sintaksno stablo varijable *syntaxTree*, ono je moguće korištenjem alata *Syntax Visualizer*. Ovaj alat spomenut je u prošlom poglavlju i ono služi za prikaz svih elemenata sintaksnog stabla. Sljedeća slika prikazuje strukturu sintaksnog stabla iz varijable *syntaxTree* sa svim čvorovima, tokenima, razmacima i praznim linijama.



Slika 10: Ispis svojstva generirane klase

Primjer koji slijedi prikazuje korištenje metode *Create*. U ovome jednostavnom primjeru deklarira se metoda imena *GeneratedMethod* i sintaksno stablo koje sadrži sve elemente te metode, uključujući modifikator pristupa, povratni tip i blok, odnosno tijelo metode koje je u ovom slučaju prazno.

```
MethodDeclarationSyntax methodDeclaration =
SyntaxFactory.MethodDeclaration(

SyntaxFactory.PredefinedType(SyntaxFactory.Token(SyntaxKind.VoidKeyword)
),

SyntaxFactory.Identifier("GeneratedMethod"))

.WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.Pu
blicKeyword)))

.WithBody(SyntaxFactory.Block());


SyntaxTree syntaxTree = CSharpSyntaxTree.Create(methodDeclaration);
```

Prva linija sadržava deklaraciju varijable *methodDeclaration* koja je tipa *MethodDeclarationSyntax*. Metodom *PredefinedType* klase *SyntaxFactory* lako je odrediti

povratni tip generirane metode. Njoj je potrebno proslijediti token tipa *VoidKeyword* što će rezultirati u metodu s povratnim tipom *void*. Metoda *Identifier* služi za određivanje imena metode, a u danom primjeru proslijeđen je niz znakova „GeneratedMethod“. Na nju se nadovezuje metoda *WithModifiers* kojom se određuje modifikator pristupa, a proslijeđuje joj se token tipa *PublicKeyword*, sličan pristup kao s metodom *PredefinedType*. I posljednja metoda je metoda *WithBody* koja služi za stvaranje bloka tijela metode, a u ovome primjer ono je prazno. U posljednjoj liniji kreira se sintaksno stablo metodom *Create* kojoj se prosljeđuje varijabla *methodDeclaration*.

Ispis varijable *syntaxTree* metodom *ToString* daje sljedeći rezultat.

```
publicvoidGeneratedMethod(){}
```

Slika 11: Ispis varijable *syntaxTree*

Iako kreirano sintaksno stablo sadrži sve što je u kôdu napisano, nema razmaka među riječima. Problem nastaje iz razloga što se za ispis sintaksnog stabla poziva metoda *ToString* s kojom ne možemo ispisati cijeli niz znakova koji uključuje razmake, to jest ne ispisuje se trivijalni dio čvorova. Za to je potrebna metoda *NormalizeWhitespace* koja se nalazi u sklopu klase *SyntaxNode*. Za dobivanje sintaksnog čvora i ispisa cijelog stabla, potrebno je dohvatiti korijen stabla. Korijen stabla dohvaća se metodom *GetRoot* koji je tipa *SyntaxNode* te se tad može iskoristiti metoda *NormalizeWhitespace*.

```
SyntaxNode root = syntaxTree.GetRoot();
Console.WriteLine(root.NormalizeWhitespace());
```

Što rezultira sljedećim ispisom:

```
public void GeneratedMethod()
{
}
```

Slika 12: Ispis koristeći *NormalizeWhitespace* metodu

Ispis sad nalikuje na metodu C# programskog jezika, sa svim riječima odvojenim razmacima te vitičastim zagradama u zasebnim linijama.

Nakon upoznavanja *GetRoot* metode, na redu su metode *DescendantNodes* i *OfType*. Metoda *DescendantNodes* dohvaća sve čvorove određenog sintaksnog stabla i također je dio klase *SyntaxNode*. Lista čvorova je tad filtrirana metodom *OfType* koja vraća samo elemente traženog tipa.

```
string code = @"
                using System;

                namespace RoslynSyntaxTree
                {
                    public class GeneratedSyntax
                    {
                        public void GeneratedMethod()
                        {
                        }
                    }
                }";

SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(code);

var methodDeclarations =
syntaxTree.GetRoot().DescendantNodes().OfType<MethodDeclarationSyntax>();

foreach (var method in methodDeclarations)
{
        Console.WriteLine($"Method name: {method.Identifier.ValueText}");
}
```

U varijablu code tipa string upisana je klasa. Klasa *GeneratedSyntax* sadrži jednu metodu, *GeneratedMethod*. Niz znakova iz varijable code parsira se u sintaksno stablo. Varijabla *methodDeclarations* će sadržavati sve čvorove koji su tipa *MethodDeclarationSyntax*. Na kraju je za ispis pronađenih metoda korištena petlja *foreach* za prolazak svih pronađenih čvorova.

U konzolu se ispisalo ime metode što znači da su metode *DescendantNodes* i *OfType* uspješno filtrirale sve čvorove i pronašle onaj čvor koji je početak sintaksnog stabla metode *GeneratedMethod*.

```
Method name: GeneratedMethod
```

Slika 13: Ispis rezultata

Generiranje programskog kôda temelji se na kreiranju sintaksnih stabla. Klasom *SyntaxTree* generiranje sintaksnog stabla je moguće, ali generiranje kôda u pravom smislu tih

riječi nije u potpunosti omogućeno metodama ove klase. Za naprednije generiranje kôda koristi se klasa *SyntaxFactory*.

4.4.5. Generiranje kôda korištenjem klase SyntaxFactory

Generiranje programskog kôda moguće je ostvariti korištenjem *SyntaxFactory* klase koja je dio *Microsoft.CodeAnalysis.CSharp* imenskog prostora (engl. *namespace*). *SyntaxFactory* je klasa koja sadrži metode za konstrukciju sintaksnih čvorova, tokena i ostalih dijelova kôda poput bijelog prostora (engl. *whitespace*), direktiva i komentara. [17]

Kroz primjere će biti prikazano pojedino generiranje osnovnih elemenata klase C# programskog jezika. Zatim će se prikazati kako dobiti cjelovitu generiranu klasu pomoću *SyntaxFactory* klase i njezinih pripadajućih metoda.

Direktive su elementi koji se nalaze u prvim linijama klasa. Deklaracija direktiva za generiranje ostvarivo je metodom *UsingDirective* za koju je potrebno proslijediti samo ime direktive. Ime se parsira metodom *ParseName*, prosljeđuje se metodu *UsingDirective* i sprema u varijablu tipa *UsingDirectiveSyntax*.

```
UsingDirectiveSyntax generatedUsingDirective =
SyntaxFactory.UsingDirective(SyntaxFactory.ParseName("System"));
```

Deklaracija direktive može se napisati u jednoj liniji kôda. U ovom primjeru varijabla *generatedUsingDirective* tipa *UsingDirectiveSyntax* sadržavat će direktivu „System“.

Kako bi ispis varijable *generatedUsingDirective* izgledao ispravno korištena je metoda *NormalizeWhitespace*. Metoda će kreirati sintaksni čvor sa bijelim prostorom zamijenjenim formatiranom sintaksnom trivijalom što rezultira sljedećim ispisom.

```
using System;
```

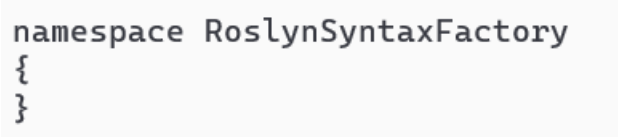
Slika 14: Generirana using direktiva

Direktiva je uspješno i ispravno ispisana u konzoli. Da se primijetiti kako je za generiranje ove direktive bila potrebna samo jedna linija kôda i upis imena direktive koju želimo generirati.

Za deklaraciju imenskog prostora koristi se metoda *NamespaceDeclaration*. Ova metoda ima pet mogućih preopterećenja kojima prihvaća različite parametre, a osnovni slučaj prosljeđivanja naziva imenskog prostora izgleda ovako:

```
NameSyntax namespaceName =
SyntaxFactory.ParseName("RoslynSyntaxFactory");

NamespaceDeclarationSyntax generatedNamespace =
SyntaxFactory.NamespaceDeclaration(namespaceName);
```

Ispis varijable *generatedNamespace* prikazan je na sljedećoj slici gdje se vidi ključna riječ *namespace* i naziv proslijeđen metodi *NamespaceDeclaration*. Osim toga prikazan je i blok vitičastih zagrada.



Slika 15: Generiran imenski prostor

Ispravan format ispisa imenskog prostora ponovno je omogućen korištenjem metode *NormalizeWhitespace*.

Za deklaraciju klase koristi se metoda *ClassDeclaration* kojoj se kao i metodi *NamespaceDeclaration* prosljeđuje naziv. Razlika u ovom primjeru je ta da se koristila metoda *Identifier* umjesto metode *ParseName*. Metoda *Identifier* prihvaća varijablu tipa *string* i od nje kreira token tipa *SyntaxToken*. Taj token se kasnije prosljeđuje metodi *ClassDeclaration* koja kreira instancu tipa *ClassDeclarationSyntax*.

```
SyntaxToken className = SyntaxFactory.Identifier("Person");

SyntaxTokenList classModifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));

ClassDeclarationSyntax generatedClass =
SyntaxFactory.ClassDeclaration(className)
                .WithModifiers(classModifiers);
```

U prvoj liniji kreira se varijabla *className* pomoću spomenute metode *Identifier*. Nakon toga, kako bi klasa sadržavala modifikator pristupa, kreirana je varijabla *classModifiers* tipa *SyntaxTokenList*. Za to je iskorištena metode *TokenList* kojoj se prosljeđuje token tipa *PublicKeyword* kako bi generirana klasa imala modifikator javnog pristupa. Na kraju se kreira varijabla *generatedClass* kojoj se modifikator dodaje pomoću metode *WithModifiers*. Metodi se prosljeđuje kreirana varijabla *classModifiers* u prethodnoj liniji kôda.

Osim korištenja metode *Identifier*, možemo i jednostavno proslijediti ime klase kao varijablu tipa *string* koje se na isti način proslijedi i parsira u sintaksne čvorove te bi tad kôd izgledao ovako:

```
string className = "Person";
SyntaxTokenList classModifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));
ClassDeclarationSyntax generatedClass =
SyntaxFactory.ClassDeclaration(className)
                .WithModifiers(classModifiers);
```

Generirana klasa ima svoj osnovni kostur, sadrži modifikator pristupa, ključnu riječ *class*, naziv „Person“ te označen blok tijela klase otvorenom i zatvorenom vitičastom zagradom.

```
public class Person
{
}
```

Slika 16: Generirani „kostur“ klase

Preostali dijelovi za generiranje su svojstvo klase i metoda. Za deklaraciju svojstva klase koristi se metoda *PropertyDeclaration*. Kako bi svojstvo imalo sve potrebne elemente, prvo je potrebno kreirati neke druge varijable. Prva od njih je *dataType* varijabla. Ova varijabla tipa je *TypeSyntax*. Ovom klasom moguće je kreirati objekte koji predstavljaju sintaksne čvorove za tip. Enumeracija *SyntaxKind* omogućava jednostavan odabir ključnih riječi tipa podataka koji postoje u C# programskom jeziku. Osim nje, iskorištene su već spomenute metode *PredefinedType* i *Token* kako bi se generirao tip svojstva.

Za ime svojstva opet je iskorištena metoda *Identifier* te enumeracija *SyntaxKind* i metode *TokenList* i *Token* za generiranje željenog modifikatora pristupa.

Svojstva, osim modifikatora pristupa, tipa podatka i naziva posjeduju get i set pristupne modifikatore. Prilikom kreiranje svojstva koristi se metoda *PropertyDeclaration* koja prihvaća dataType varijablu kako bi se znao generirani tip podatka za svojstvo, naziv svojstva te metoda WithModifiers za modifikator pristupa.

Kako bi se deklarirali *get* i *set* modifikatori iskorištena je metoda *WithAccessorList*. Njoj se prosljeđuje lista *SyntaxList* tipa *AccessorDeclarationSyntax*. Iz tog razloga kreira se lista sintaksnih čvorova tog tipa metodom *List*. Za kreiranje *get* modifikatora iz enumeracije *SyntaxKind* odabire se *GetAccessorDeclaration*. Uz to, odabran je token za znak točke sa zarezom kako bi se odvojio *get* od *set* modifikatora. Isti postupak potrebno je napraviti za *set* i s time završava kreiranje svojstva klase.

```
TypeSyntax dataType =
SyntaxFactory.PredefinedType(SyntaxFactory.Token(SyntaxKind.IntKeyword))
;

SyntaxToken propertyName = SyntaxFactory.Identifier("Age");

SyntaxTokenList modifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));


PropertyDeclarationSyntax generatedProperty =
SyntaxFactory.PropertyDeclaration(dataType, propertyName)

     .WithModifiers(modifiers)

     .WithAccessorList(SyntaxFactory.AccessorList(

     SyntaxFactory.List(new[]

     {
SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration)
.WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)),

SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration)
.WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken))

     })));
```

Sa svim potrebnim podacima upisanim u varijablu generatedProperty, u konzoli se može ispisati generirano svojstvo.

```
public int Age { get; set; }
```

Slika 17: Generirano svojstvo

Generirano svojstvo sadrži sve elemente, uključujući *get* i *set* modifikatore sa znakom točka sa zarezom iza svakog od njih.

Za deklaraciju metode koristi se metoda *MethodDeclaration*. Kako bi generirana metoda imala povratni tip on se generirana kombinacijom metoda *PredefinedType*, Token i odabirom elemente enumeracije. Ime se prosljeđuje metodi *Identifier* kao i u prijašnjim primjerima. Metoda osim toga sadrži i popis parametara, no za potrebe primjera, kreirana je instanca prazne liste parametara metodom *ParameterList*. Modifikator pristupa generira se na isti način, korištenjem *TokenList* i *Token* metoda te odabirom ključne riječi iz enumeracije za željeni modifikator. Tijelo metode sastavlja se u varijabli tipa *BlockSyntax*.

Generirana metoda tipa je *MethodDeclarationSyntax* i osim metode *MethodDeclaration* iskoristiti treba metode koje će uključiti listu parametara, modifikator pristupa i tijelo metode. To je moguće korištenjem metoda *WithParameterList*, *WithModifiers* i *WithBody* kojima se zasebno prosljeđuju varijable potrebnog tipa generirane ranije u kôdu.

```
TypeSyntax returnType =
SyntaxFactory.PredefinedType(SyntaxFactory.Token(SyntaxKind.VoidKeyword)
);
SyntaxToken methodName = SyntaxFactory.Identifier("MyGeneratedMethod");
ParameterListSyntax parameterList = SyntaxFactory.ParameterList();
SyntaxTokenList modifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));
BlockSyntax methodBody = SyntaxFactory.Block();


MethodDeclarationSyntax generatedMethod =
SyntaxFactory.MethodDeclaration(returnType, methodName)
     .WithParameterList(parameterList)
     .WithModifiers(modifiers)
     .WithBody(methodBody);
```

U nastavku je vidljiv ispis generirane klase naziva „Walk“ sa svim pripadajućim elementima.

```
public void Walk()
{
}
```

Slika 18: Generirana metoda

Posljednji dio potreban za generiranje klase je jedinice kompilacije (engl. *Compilation Unit*). Jedinica kompilacije tipa je *CompilationUnitSyntax* i kreira se metodom *CompilationUnit* klase *SyntaxFactory*.

```
CompilationUnitSyntax compilationUnit = SyntaxFactory.CompilationUnit();
```

Jedinicu kompilacije nije moguće ispisati jer se u njoj ne nalazi tekst već ona služi za unos svih elemenata koji će tvoriti strukturu generirane klase.

Ukoliko se naprave izmjene metoda iz primjera koje kreiraju elemente klase tako da vraćaju generirani kôd, moguće ih je redom pozivati u zasebnoj metodi i generirati cjelovitu klasu. U kôdu to izgleda ovako:

```
CompilationUnitSyntax compilationUnit = SyntaxFactory.CompilationUnit();
var generatedUsingDirectives =
GenerateCode_SyntaxFactory_UsingDirective();
var generatedNamespace = GenerateCode_SyntaxFactory_Namespace();
var generatedClass = GenerateCode_SyntaxFactory_Class();
```

```
var generatedProperty = GenerateCode_SyntaxFactory_Property();
var generatedMethod = GenerateCode_SyntaxFactory_Method();


generatedClass = generatedClass.AddMembers(generatedProperty,
generatedMethod);
generatedNamespace = generatedNamespace.AddMembers(generatedClass);
compilationUnit =
compilationUnit.AddUsings(generatedUsingDirectives).AddMembers(generated
Namespace);


string generatedCompleteClass =
compilationUnit.NormalizeWhitespace().ToFullString();
```

Za početak, u kôdu je kreirana jedinica kompilacije. Ona će služiti kako bi se u nju zapisao generirani kôd ostalih elemenata. Nakon jedinice kompilacije, u kôdu je kreirana, odnosno generirana direktiva, imenski prostor, kostur klase, svojstvo i metoda. Svaki dio generiran je na način koji je prikazan u prethodnim primjerima te je svaki spremljen u zasebnu varijablu.

Nakon generiranja elemenata potrebno ih je urediti i spremiti u jednu logičku cjelinu. Prvi korak je dodavanja svojstva i metode u klasu. U već postojeću varijablu *generatedClass* spremaju se varijable *generatedProperty* i *generatedMethod* koristeći metodu *AddMembers*. One su spremljene kao tip *MemberDeclarationSyntax* i u tom su trenutku dio sintaksnog stabla klase.

Kad su članovi klase dodani, slijedi dodavanje same klase u imenski prostor. U varijablu *generatedNamespace*, ponovno koristeći metodu *AddMembers*, dodaje varijabla *generatedClass*.

Naposlijetku, u jedinicu kompilacije prvo se metodom *AddUsings* dodaje generirana direktiva zapisana u varijabli *generatedUsingDirectives*, a zatim *generatedNamespace* koji sadrži generirani kôd klase zajedno sa generiranim svojstvom i metodom još jednom s metodom *AddMembers*.

Redoslijed pozivanja metoda nije presudan, ali redoslijed dodavanja članova u pojedine čvorove je važan dio postupka generiranja kôda klase. Dakle, dodavanje članova ima logičan slijed, kreće se od unutarnjih članova koji se dodaju vanjskim sve dok se ne dođe do posljednjeg, to jest direktive. Kako bi generirana klasa imala ispravan format, spremanje generiranog kôda zaključuje se metodama *NormalizeWhitespace* i *ToFullString*.

Nakon što je cijeli programski kôd generiran i spremljen kao niz znakova, ispisom varijable *generatedCompleteClass* dobiva se sljedeći rezultat:

```
using System;

namespace RoslynSyntaxFactory
{
    public class Person
    {
        public int Age { get; set; }

        public void Walk()
        {
        }
    }
}
```

Slika 19: Cjelovita generirana klasa

S nekoliko linija kôda uspješno je generirana klasa s cjelokupnom strukturom elemenata. Ovaj jednostavan primjer prikazuje samo osnovne mogućnosti generiranja kôda Roslyn kompilatorom.

4.5. Ostale mogućnosti Roslyna

Roslyn plaftorma razvijena je kako bi unaprijedila kvalitetu i efikasnost rada programera i smanjila utrošeno vrijeme na trivijalne elemente programiranja. Osim što Roslyn pruža moćne alate i API-je za generiranja kôda, ova platforma također nudi analizatore koji rade automatsku analizu i obradu napisanog kôda.

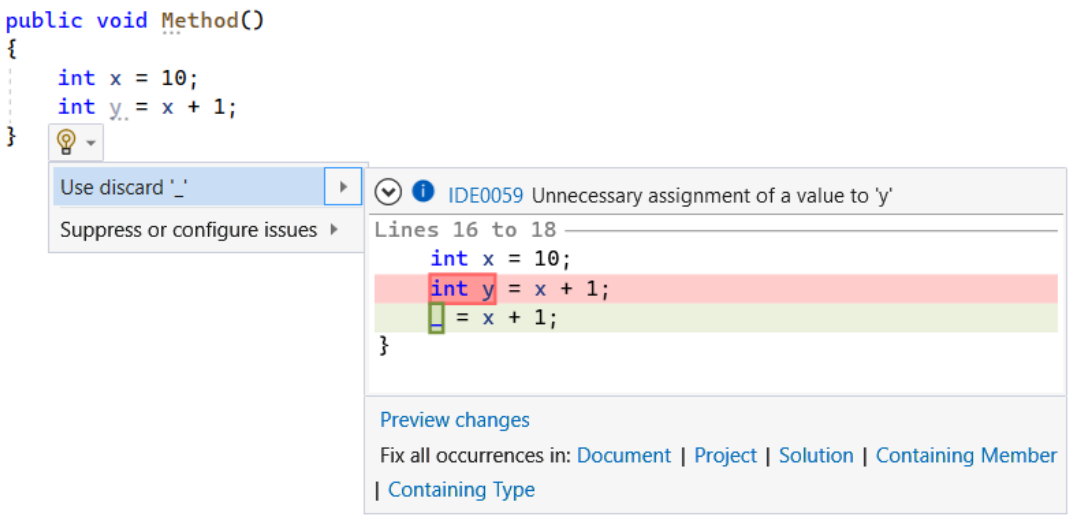
4.5.1. Analiza kôda Roslyn kompilatorima

Analizatori kôda (engl. *Code Analyzers*) provjeravaju kôd tijekom tzv. vremena dizajniranja (engl. *design time*) u svim otvorenim datotekama. Oni analiziraju kôd za stil, kvalitetu, održivost, dizajn i ostale probleme i pogreške. [18]

Analizatori se dijele u tri skupine. Svaka skupina analizatora zadužena je za drugo područje kôda koje treba analizirati. Prva skupina analizatora kôda koju se može izdvojiti su analizatori stila kôda. [18]

Analizatori stila kôda (engl. *Code style analyzers*) ugrađeni su u razvojnu okolinu Visual Studio. Svako pravilo ugrađeno u ove analizatore ima svoj jedinstveni identifikacijski broj ili ID. Format po kojem se pravila zapisuju je IDExxxx, kao što je na primjer IDE1000. [18] Ova pravila pomažu održati konzistentnost stila kôda. [19]

Primjer analize analizatori stila kôda i jednog njihovog pravila može se vidjeti na sljedećoj slici:



Slika 20: Pravilo analizatora stila kôda

Slika prikazuje pogrešku identifikatora IDE0059 kojom se označava nepotrebno dodjeljivanje vrijednosti varijabli.

Sljedeća skupina analizatora su analizatori kvalitete kôda (engl. *Code quality analyzers*). Ovi analizatori su također uključeni u razvojnu okolinu Visual Studio. Njihov format je CAxxxx. Analiza koju analizatori kvalitete kôda rade vezana je uz sigurnost, performanse, dizajn i ostale slične probleme. [19] Razina analize koju analizatori obavljaju može se odabrati, postoji nekoliko postavki analize. Postavke se mogu podesiti na način da se analizom ne provjerava ni jedno pravilo, njih nekoliko odabranih ili se može postaviti tako da analizom kôda traži kršenje svih pravila koja postoje. [19]

Posljednja skupina analizatora su analizatori koji nisu uključeni u razvojnu okolinu Visual Studio. Ovi analizatori se nazivaju vanjski analizatori (engl. *external analyzers*). Njih se dohvaća kao NuGet paket ili kao Visual Studio ekstenzija. Neki od ovih analizatora su StyleCop, Roslynator, XUnit Analyzers i Sonar Analyzer. [18]

Razine ozbiljnosti analizatora (engl. *Severity levels of analyzers*) su razine kojima se želi prikazati koliko je ozbiljno kršenje pravila u kôdu. Postoji sedam razina ozbiljnosti.

Prva razina koja postoji je razina u kojoj nema nikakve ozbiljnosti, odnosno ne krši se ni jedno pravila.

Zadana (engl. *Deafult*) ozbiljnost odgovara zadanoj ozbiljnosti pravila, a ta vrijednost određena za pojedino pravilo može se pronaći u njegovim svojstvima.

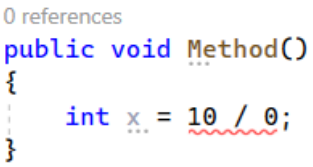
Postoji razina koju korisnik razvojne okoline neće vidjeti jer je skrivena (engl. *hidden*). Iako nije vidljiva korisniku, dijagnostički sustav razvojne okoline (engl. *IDE diagnostic engine*) će zaprimiti tu dijagnostiku.

Pravila stila kôda i pravila kvalitete kôda mogu biti informativne razine. Kôd koji krši ova pravila označen je u trima sivim točkama te može biti obojan u sivu boju. Takav kôd ne mora biti ispravljen već se rješenje unutar razvojne okoline nudi samo kao prijedlog, a ne kao obavezna promjena.

Kôd kojim se krši ozbiljnija razina pravila označen je kao upozorenje (engl. *warning*).

Razine najveće ozbiljnosti je pogreška (engl. *error*).

Na sljedećoj slici prikazano je kako se u kôdu označava pogreška i prijedlog. Prijedlog, kako je navedeno, se prepoznaje po trima sivim točkama ispod kôda koji krši pravilo. Naziv metode „Method“ ispod sebe ima tri točke, ali za razliku od deklarirane varijable imena „x“ nije obojan u sivo. Pogreška, kojom se označava kršenje najveće razine ozbiljnosti označena je crvenom bojom upravo kako bi naglasila tu ozbiljnost. Ispod dijela kôda u kojem se pokušava podijeliti broj s nulom označenim je crvenim, valovitim linijama. Pokretanje kôda s ovakvim pogreškama nije moguće. Pogreške koje uzrokuju nemogućnost pokretanja programa često su tipa logičkih pogrešaka ili pogreške sintakse.



Slika 21: Pogreška i prijedlog označeni u kôdu

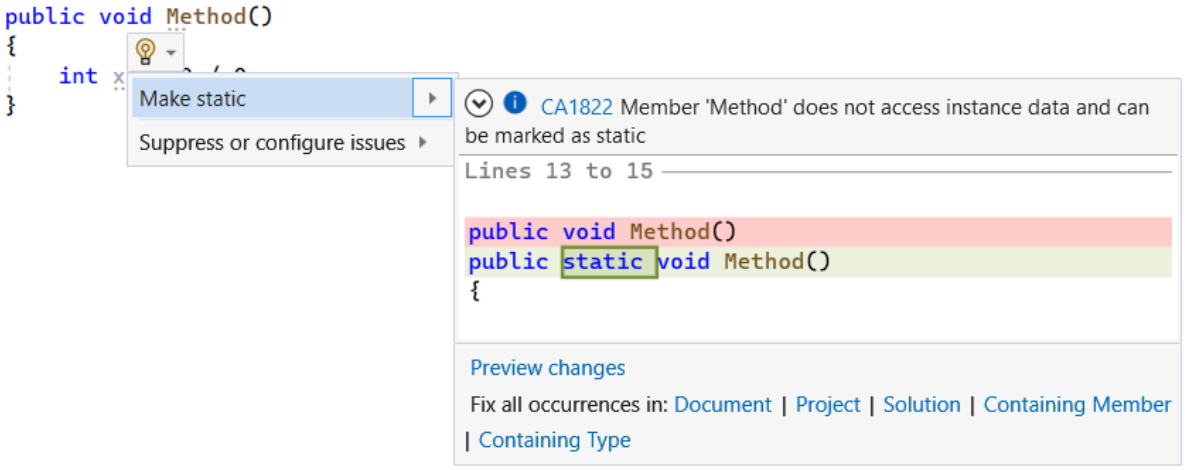
Upozorenje je u razvojnoj okolini Visual Studio označeno zelenim valovitim linijama ispod dijela kôda koji je problematičan. Prijedlozi i upozorenja često nastaju zbog korištenja neinicijaliziranih varijabli i kôda koji je deklariran, ali neiskorišten. Zatim pisanje kôda koji je u određenim slučajevima nepokriven tijekom izvođenja selekcija tipa „if-else“ i ostale slične pogreške. Programski kôd s prijedlozima i upozorenjima nije idealan, ali će ih kompilator zanemariti i pokrenuti.

Primjer upozorenja u kôdu nalazi se na sljedećoj slici.

```
foreach (var usingDirective in usingDirectives)
{
    if (!identifiers.Contains(usingDirective.Name.ToString()))
    {
        Console.WriteLine($"Unused using directive: {usingDirective.Name}");
    }
}
```

Slika 22: Upozorenje označeno u kôdu

Osim što se analizom kôda mogu uočiti pogreške, Roslyn kompilatori za analizu mogu i predložiti rješenje problema nastalog u kôdu. Jedan primjer danog rješenja prikazan je na slici , a prijedlog ispravka kôda sa slike prikazan je u nastavku.



Slika 23: Ponuđeni ispravak

Sa slike je vidljivo da je osim ispravka dana i identifikacija pravila koje se prekršilo. To je pravilo CA1822 i spada pod pravila kvalitete kôda. Odmah ispod moguće je vidjeti i pretpregled promjena koje će se izvršiti ukoliko se odabere ponuđeno rješenje.

4.5.2. Refaktoriranje kôda Roslyn kompilatorima

Refaktoriranje kôda (engl. *Code Refactoring*) je tehnika koja služi za poboljšanje kôda bez da se promijeni njegovo ponašanje. [20]

Refaktoriranjem se može povećati čitljivost i održivost kôda, a Roslyn kompilatori omogućavaju refaktoriranje s nekoliko ugrađenih alata u Visual Studiju.

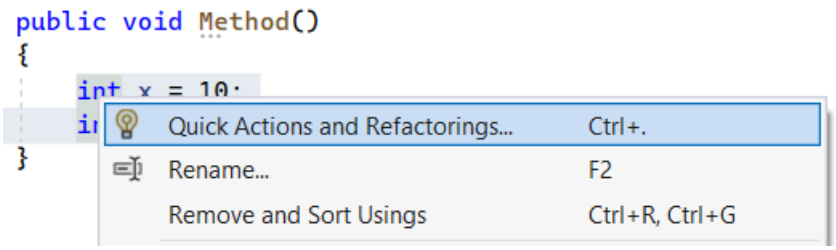
Česta tehnika kojom se povećava čitljivost i održivost je stavljanje dijelova kôda koji čine zasebne logičke cjeline u pojedine metode. Alat *Extract Method* radi upravo to.

Alat *Rename* za preimenovanje naziva koristan je alat jer ne mijenja naziv samo označenoj varijabli već svim referencama koje se nalaze u projektu.

Često se tijekom programiranja isprobavaju različita rješenja koja sa sobom nose različite *using* direktive. Kad one više nisu potrebne, odnosno ni jedan dio kôda ne koristi određenu direktiva, kompilator će to prepoznati i ponuditi ukloniti sve direktive koje se više ne koriste.

Za više opcija moguće je pritisnuti opciju koja je označena žaruljom za brze radnje i refaktoriranje (engl. *Quick Actions and Refactorins*) kôda. Ukoliko postoji potreba za promjenama u kôdu ovaj alat će prikazati rješenja.

Opisanim alatima može se pristupiti na više načina u Visual Studio, a na sljedećoj slici prikazan je jedan od njih.



Slika 24: Alati za refaktoriranje

Ovo su neke od mogućnosti refaktoriranja koje nudi Visual Studio uz pomoć Roslyn kompilatora.

5. Izrada generatora programskog kôda

Nakon upoznavanja osnova generiranja programskog kôda te Roslyn kompilatora, u ovome poglavlju fokus će biti na izradi generatora. Uz to će detaljno biti opisani zahtjevi za implementaciju generatora, dizajn rješenja i projekti koji će činiti izrađeni generator.

5.1. Specifikacija zahtjeva

Generator programskog kôda bit će korišten kao potpora produktivnosti programerima za brže kreiranje potrebnih klasa i smanjenje utrošenog vremena. Generator bi trebao omogućiti korisniku dva načina generiranja programskog kôda. U oba načina krajnji cilj je generirati klase programskog jezika C#. Podatke iz kojih će se klase generirati dani su alatu od strane korisnika ručnim unosom ili dohvaćanjem metapodataka baze podataka. Za drugi način, generator bi trebao moći prihvatiti pristupne podatke (engl. *connection string*) bazi podataka i pomoću njih dohvatiti metapodatke baze o postojećim tablicama u njoj.

Generator bi trebao moći za oba načina kreirati klase u programskom jeziku C#. Ono što podrazumijeva kreiranje klase je sljedeće. Prvo je potrebno podatke tablice ili ručnog unosa pretvoriti u podatke koje C# programski jezik može koristiti. Nakon toga slijedi generiranje strukture klasa koje sadrže svojstva preuzeta iz tablica ili ručnog unosa. Generirani kôd sprema se u datoteku i formatira kako bi klasa C# jezika imala svoju uobičajenu strukturu. Kreirana datoteka se na kraju dodaje u željeni projekt.

5.2. Dizajn rješenja

Alat za generiranje programskog kôda, odnosno generator, bit će pisan pomoću .NET tehnologije u C# programskom jeziku. Generator će biti podijeljen u dva projekta. Prvi projekt bit će projekt Visual Studio ekstenzije, odnosno VSIX (skraćeno od engl. *Visual Studio Integration Extension*) projekt. Taj projekt, odnosno njegova klasa *MyCommand*, služit će kao pokretač drugog projekta. Klasa *MyCommand* povezana je s drugim projektom tako što pokreće glavni prozor u kojem korisnici upisuju ručno podatke ili pristupne podatke za bazu.

Projekt u kojem se nalazi prezentacijski dio i poslovna logika vezana uz alat generiranja programskog kôda je WPF (skraćeno od engl. *Windows Presentation Foundation*) projekt. Programsko rješenje sadrži tri prozora, *MainWindow*, *FetchedTablesWindow* i *SaveClassWindow*. Dakle, prozor *MainWindow* pokreće prvi projekt, u klasi *MyCommand*. Ukoliko korisnik odluči generirati klase iz tablica baze, upisuje pristupne podatke za bazu u

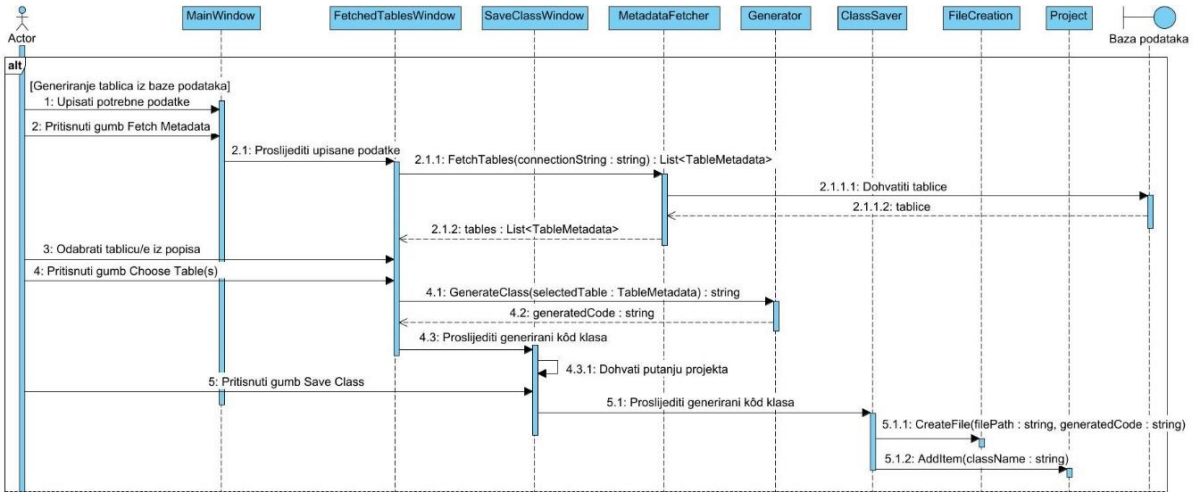
tekstualno polje (engl. *textbox*) i otvara prozor *FetchedTablesWindow*. Nakon odabira tablica koje će se generirati, otvara se treći prozor, *SaveClassWindow*, u kojem se odabire projekt u koji će se spremiti generirana klasa.

Osim toga, u rješenju je klasa *TableMetadata* koja služi kao model po kojem će se podaci iz tablica pretvarati u podatke generiranih klasa. Ovaj projekt sadrži i mapu naziva *Generator_BLL* u kojoj se nalazi nekoliko klasa. Kao što se iz imena može zaključiti, ova mapa sadrži klase koje su dio poslovne logike (engl. *business logic*) rješenja. U mapi se nalaze dva sučelja (engl. *interface*) *IDatabaseDataTypeMapper* i *IDatabaseMetadataFetcher*. Prvo navedeno sučelje sadrži deklaraciju metode kojom će se prilikom realizacije sučelja u klasi implementirati mapiranje tipova podataka za određenu bazu podataka. Klasa *SSMSDataTypeMapper*, koja implementira navedeno sučelje i dio je ovog projekta, mapira tipove podataka tablica *SQL Server Management Studio* u tipove podataka C# klase, odnosno onih tipova koji se mogu pronaći u C# programskom jeziku. Drugo sučelje, odnosno njegova realizacija u klasi *SSMSMetadataFetcher* služi za dohvaćanje metapodataka baze, u ovom slučaju također za *SQL Server Management Studio*.

Preostale tri klase su *ClassSaver*, *FileManager* i *Generator*. Klasa *Generator* povezana je s prozorima *MainWindow* i *FetchedTablesWindow*, a kao što ime odaje, u njoj se izvršava generiranje programskog kôda. Nakon što se generira kôd i odabere projekt prozoru *SaveClassWindow* u koji će biti spremljena klasa, korisnik pritišće gumb za spremanje te se poziva klasa *ClassSaver*. Ova klasa dohvatit će projekt za spremanje klase te spremiti generiranu klasu u njega. Klasa u međuvremenu poziva i *FileManager* klasu kojoj je svrha kreirati datoteku i u nju zapisati generirani kôd prije nego što se spremi u projekt.

Za generiranje programskog kôda bit će korišten .NET Compiler API, odnosno Roslyn. Klasa *SyntaxFactory* i njezine metode koristiti će se za generiranje svojstava i ostalih elemenata koji čine klasu C# programskog jezika.

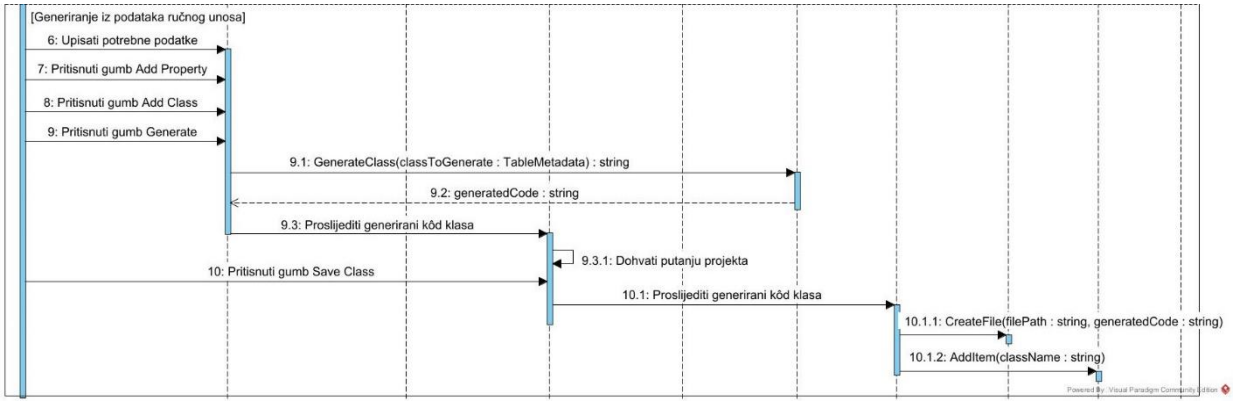
Vizualni prikaz opisanog rješenja moguće je prikazati dijagramima. Sljedeća slika prikazuje UML dijagram slijeda.



Slika 25: Dijagram slijeda generiranja kôda iz tablica

Dijagram slijeda za rješenje ima u sebi alt fragment te je ovdje prikazan slijed događaja prilikom generiranja klasa iz tablica baze podataka. Može se vidjeti kako korisnik, na dijagramu označen riječju „Actor“, komunicira samo s prozorima u projektu kroz koje unosi podatke i prima povratne informacije. Prozori prosljeđuju dane podatke ostalim klasama projekta koje ih obrađuju i koriste u različite svrhe, na načine koji su opisani ranije.

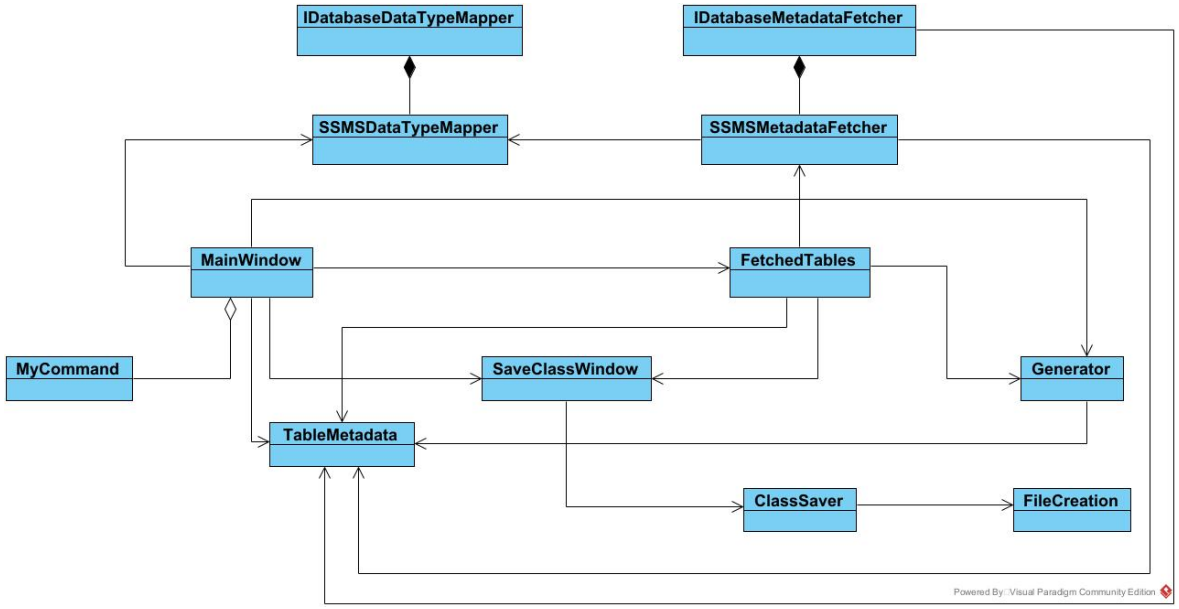
Drugi dio dijagrama slijeda je onaj slučaj kad korisnik unese podatke ručno. Generiranje kôda podacima iz ručnog unosa slično je prijašnjem slučaju samo što rješenje ovdje ne ostvaruje komunikaciju s bazom podataka već odmah kreće u generiranje kôda i kreiranje datoteke.



Slika 26: Dijagram slijeda generiranja kôda podacima iz ručnog unosa

Dijagram klasa jedan je od osnovnih dijagrama kojim se prikazuju veze i odnosi među klasama softverskog rješenja. Na dijagramu su priložene sve klase koje se u projektu nalaze te je njihova povezanost označena vezama asocijacije. Dva sučelja koja su dio projekta i

njihove implementacije povezane su vezom kompozicije kako bi se označila čvrsta povezanost ovih cjelina. Vezom agregacije povezana je klasa *MyCommand* projekta ekstenzije s glavnim prozorom *MainWindow* drugog projekta. Budući da su oba projekta zasebne cjeline, ove su klase povezane slabijom vezom.

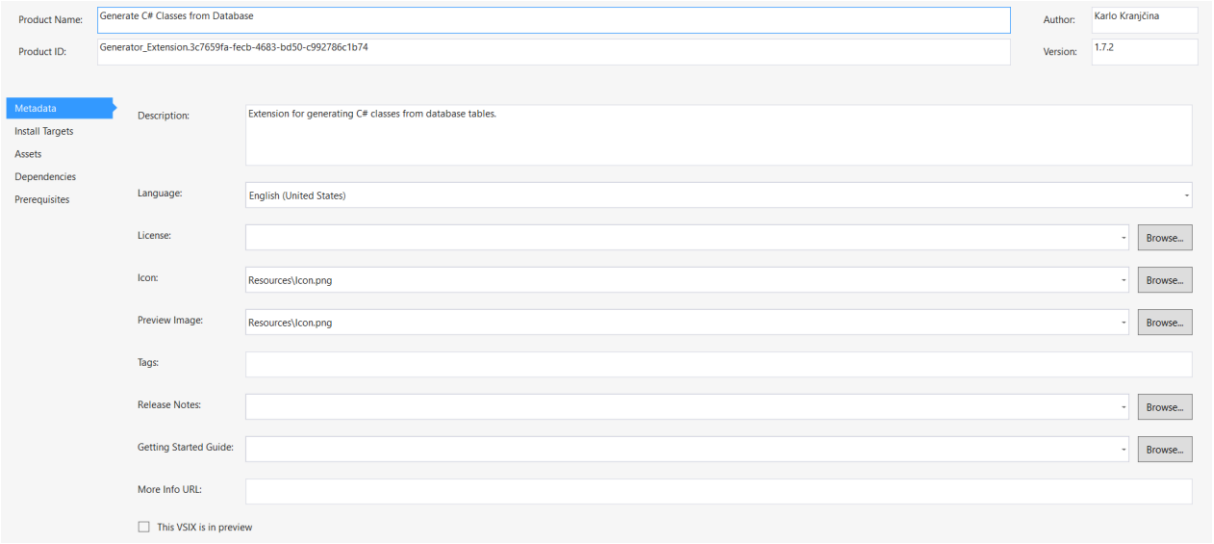


Slika 27: Dijagram klasa

5.2.1. Projekt ekstenzije

Izradu ekstenzije započinjemo kreiranjem tzv. VSIX projekta. Projektne datoteke koje je važno spomenuti su *source.extension.vsixmanifest*, *VSCommandTable.vsct* i *MyCommand.cs*.

U datoteku *source.extension.vsixmanifest* moguće je unijeti podatke o ekstenziji kao što su naziv ekstenzije, autor, verzija, opis te ostalo, što je vidljivo na slici ispod.



Slika 28: Prikaz manifest datoteke projekta ekstenzije

VSCommandTable.vsct je datoteka pisana u XML jeziku čiji je kôd moguće modificirati kako bi se postavilo željeno mjesto pristupa ekstenziji unutar Visual Studija. Za potrebe vlastite ekstenzije, pristup je omogućen kroz *Tools* izbornik.

```
<?xml version="1.0" encoding="utf-8"?>
<CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-18/CommandTable" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <Extern href="stdidcmd.h"/>
  <Extern href="vsshlids.h"/>
  <Include href="KnownImageIds.vsct"/>
  <Include href="VSGlobals.vsct"/>

  <Commands package="Generator_Extension">
    <Groups>
      <Group guid="Generator_Extension" id="MyMenuGroup" priority="0x0600">
        <Parent guid="VSMainMenu" id="Tools"/>
      </Group>
    </Groups>

    <Buttons>
      <Button guid="Generator_Extension" id="MyCommand" priority="0x0100" type="Button">
        <Parent guid="Generator_Extension" id="MyMenuGroup" />
        <Icon guid="ImageCatalogGuid" id="GenerateTable" />
        <CommandFlag>IconIsMoniker</CommandFlag>
        <Strings>
          <ButtonText>Generate Classes</ButtonText>
          <LocCanonicalName>.Tools.GenerateClasses</LocCanonicalName>
        </Strings>
      </Button>
    </Buttons>
  </Commands>

  <Symbols>
    <GuidSymbol name="Generator_Extension" value="{f7e3565e-dca9-49a9-88cc-1b3aea14afaa}">
      <IDSymbol name="MyMenuGroup" value="0x0001" />
```

```
        <IDSymbol name="MyCommand" value="0x0100" />
      </GuidSymbol>
    </Symbols>
  </CommandTable>
```

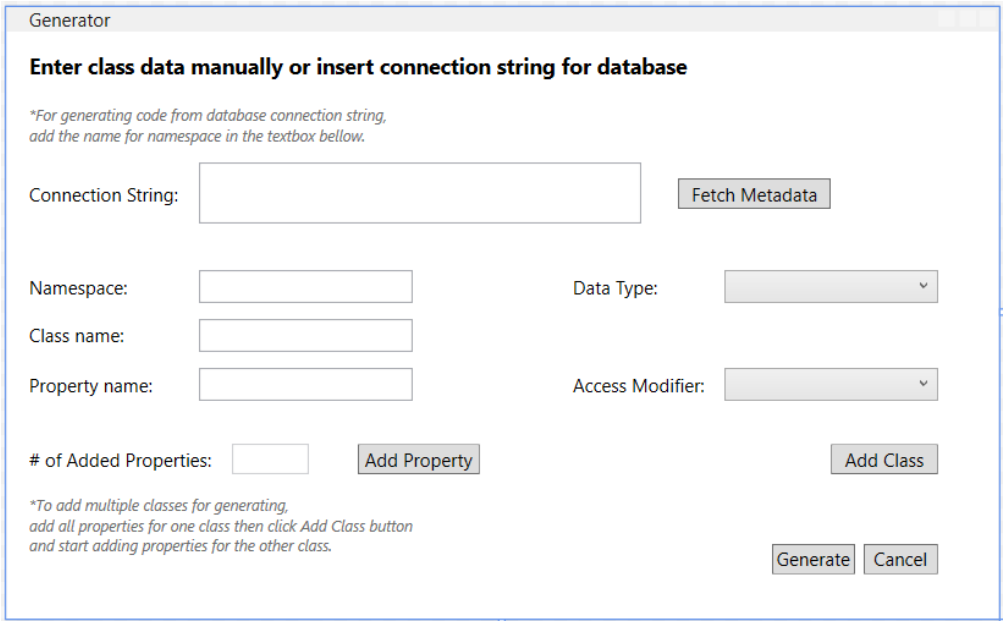
Klasa *MyCommand* nasljeđuje *BaseCommand* klasu koja služi za lakše upravljanje naredbama (engl. *commands*). *MyCommand* sadrži nadjačanu metodu *Execute* čiji se kôd izvršava prilikom pokretanja ekstenzije. Ova metoda pokreće glavni prozor WPF projekta u kojem se nalazi kompletna logika generiranja programskog kôda. Spomenuti projekt dodan je kao referenca projektu ekstenzije.

```
[Command(PackageIds.MyCommand)]
    internal sealed class MyCommand : BaseCommand<MyCommand>
    {
        protected override void Execute(object sender, EventArgs e)
        {
            MainWindow mainWindow = new MainWindow();
            mainWindow.Show();
        }
    }
```

5.2.2. Projekt alata za generiranje programskog kôda

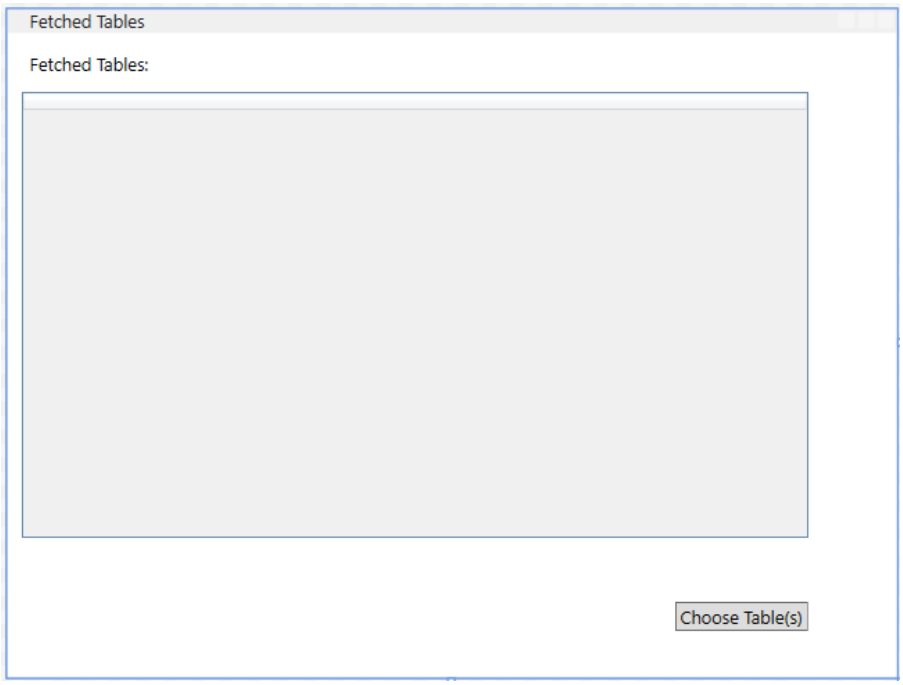
Za početak rada s Roslynom, treba uključiti NuGet paket *Microsoft.CodeAnalysis* u projekt. Također, za unos ispravne sintakse potrebna je *SyntaxFactory* klasa koja se nalazi unutar *Microsoft.CodeAnalysis.CSharp*. [21]

Počevši od vizualnog aspekta projekta, prvi prozor koji je prikazan korisniku nakon pokretanja ekstenzije je *MainWindow*. Ovdje možemo vidjeti oznake (engl. *label*) koje upućuju korisniku koje podatke treba unijeti kako bi se klasa ispravno generirala. Kad korisnik unese podatke koristeći polja za unos teksta, unos se zatim provjerava i kroz okvire s porukama (engl. *Message box*) obavještava korisnika ukoliko je potrebno izmijeniti neispravan unos. Ako je korisnik odabrao ručno unijeti podatke klase, podaci se pohranjuju te prosljeđuju prozoru *SaveClassWindow*. Ako se korisnik odluči na generiranja klase prema metapodacima baze podataka, primoran je napisati niz za povezivanje s bazom te naziv imenskog prostora. Ovi se podaci prosljeđuju prozoru *FetchedTablesWindow*.



Slika 29: *MainWindow* prozor

Nakon prosljeđivanja podataka prozoru *FetchedTablesWindow*, rješenje pokušava ostvariti vezu s bazom i dohvatiti podatke tablica koji se u njoj nalaze. Dohvaćene tablice prikazuju su u prozoru *FetchedTablesWindow* pomoću *DataGrid* korisničke kontrole te se na prozoru još nalazi oznaka i gumb za odabir tablica za generiranje.



Slika 30: FetchedTablesWindow prozor

Kad se korisnik odluči za generiranje klase, pritišće gumb *Choose Tables(s)* čime se poziva metoda klase *Generator* za generiranje kôda. Ona prima kao parametar cijelu varijablu tipa *TableMetadata*.

Klasa *TableMetadata* predstavlja model po kojem se mapiraju svi elementi iz tablica ili ručnog unosa u elemente klase C# programskog jezika. Ona se sastoji od klasa *TableMetadata* i *ColumnMetadata*. U objekt klase *TableMetadata* moguće je zapisati naziv tablice, listu stupaca tipa *ColumnMetadata*, a za potrebe implementacije, može se zapisati i naziv za imenski prostor, to jest posjeduje svojstvo *Namespace*. U objekt klase *ColumnMetadata* zapisuju se podaci stupaca tablice. U podatke spadaju nazivi stupaca, njihov tip podataka i modifikator pristupa.

```
public class TableMetadata
    {
        public string Name { get; set; }
        public List<ColumnMetadata> Columns { get; set; }
        public string Namespace { get; set; }
    }

    public class ColumnMetadata
    {
        public string Name { get; set; }
        public string AccessModifier { get; set; }
        public string DataType { get; set; }
    }
```

Implementacija klase *Generator* sadrži šest metoda. Ona najbitnija, koja je jedina dio javnog sučelja, naziva se *GenerateClass*.

```
public string GenerateClass(TableMetadata classToGenerate)
        {
            string formattedClassName =
Regex.Replace(classToGenerate.Name, @"\s", "_");
            string formattedNamespace =
Regex.Replace(classToGenerate.Namespace, @"\s", "_");


            UsingDirectiveSyntax generatedUsingDirective =
GenerateUsingDirectiveCode();
            ClassDeclarationSyntax generatedClass =
GenerateClassCode(formattedClassName);
            PropertyDeclarationSyntax property;
```

```
            foreach (ColumnMetadata column in classToGenerate.Columns)
            {
                SyntaxTokenList modifiers = GetAccessModifier(column);

                string formattedDataType =
Regex.Replace(column.DataType, @"\s", "_");
                string formattedColumnName = Regex.Replace(column.Name,
@"\s", "_");
                property = GeneratePropertyCode(modifiers,
formattedDataType, formattedColumnName);
                generatedClass = generatedClass.AddMembers(property);
            }

            NamespaceDeclarationSyntax generatedNamespace =
GenerateNamespaceCode(formattedNamespace, generatedClass);
            var compilationUnit =
SyntaxFactory.CompilationUnit().AddUsings(generatedUsingDirective).AddMe
mbers(generatedNamespace);
            string generatedCode =
compilationUnit.NormalizeWhitespace().ToFullString();

            return generatedCode;
        }
```

Nakon što se ona pozove iz prozora *FetchedTablesWindow*, provjeri se ima li u nazivima klase i imenskom prostoru razmaka. Ukoliko ima, stavlja se znak „_“ umjesto razmaka. Nakon toga slijedi generiranja kôda klase. Prva na redu je metoda *GenerateUsingDirectiveCode*. Ova metoda ima za povratni tip *UsingDirectiveSyntax* koji se dobiva pozivom metode *UsingDirective* iz klase *SyntaxFactory*. Za ovaj generator nije moguće izabrati direktive klase već se svakoj automatski dodaje direktiva „System“.

```
private UsingDirectiveSyntax GenerateUsingDirectiveCode()
{
    return
SyntaxFactory.UsingDirective(SyntaxFactory.ParseName("System"));
}
```

Sljedeća metoda koja se poziva je metoda *GenerateClassCode* kojom se generira osnovna struktura i prazan blok klase s proslijeđenim nazivom. Za to je korištena metoda *ClassDeclaration*.

```
private ClassDeclarationSyntax GenerateClassCode(string
formattedClassName)
```

```
{
    return SyntaxFactory.ClassDeclaration(formattedClassName);
}
```

Za svako svojstvo koje klasa sadrži pozivaju se metode *GetAccessModifier* i *GeneratePropertyCode*. Metodom *GetAccessModifier* kreiraju se tokeni koji će zapravo biti modifikatori pristupa pojedinog svojstva.

```
private SyntaxTokenList GetAccessModifier(ColumnMetadata column)
{
    SyntaxTokenList modifiers;
    if (column.AccessModifier == "Private")
    {
         modifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PrivateKeyword));
    }
    else if (column.AccessModifier == "Protected")
    {
        modifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.ProtectedKeyword)
);
    }
    else
    {
        modifiers =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));
    }

    return modifiers;
}
```

Metoda *GeneratePropertyCode* prima nekoliko ulaznih parametara. Prima pripadajući modifikator svojstva kreiran u prijašnjoj metodi, tip podatka i naziv svojstva. Metoda *PropertyDeclaration* prvo zapisuje tip podatka i naziv, a zatim se metodama *WithModifiers* i *WithAccessorList* generiraju modifikatori pristupa i *get* i *set* metode. *Get* i *set* metode spremaju se u listu i za njihovo se generiranje još koristi metoda *AccessorDeclaration* te metoda *WithSemicolonToken* za generiranje znaka točka sa zarezom. Ova svojstva dodaju se u već generirani blok klase metodom *AddMembers*.

```
private PropertyDeclarationSyntax GeneratePropertyCode(SyntaxTokenList
modifiers, string formattedDataType, string formattedColumnName)
```

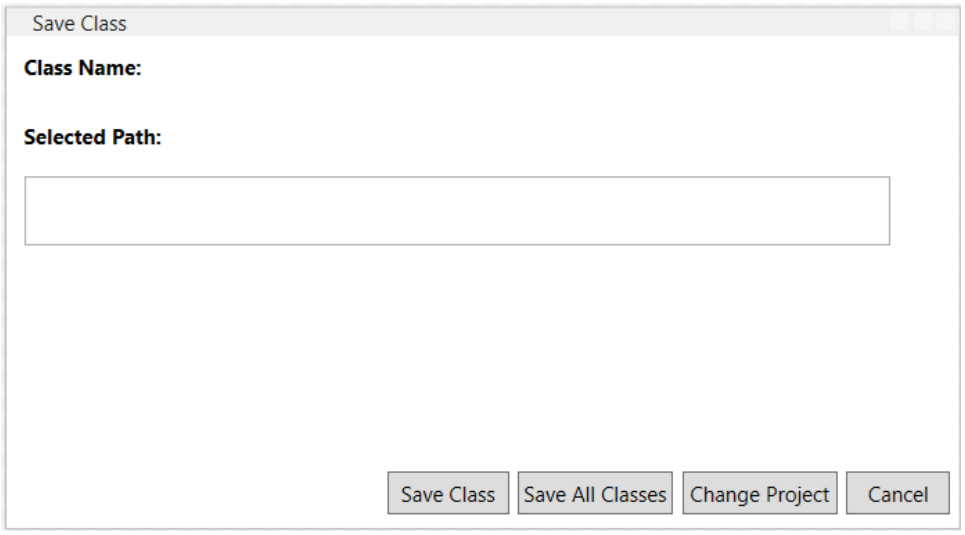
```
{
    return
SyntaxFactory.PropertyDeclaration(SyntaxFactory.ParseTypeName(formattedD
ataType), formattedColumnName)
                        .WithModifiers(modifiers)
                        .WithAccessorList(SyntaxFactory.AccessorList(
                            SyntaxFactory.List(new[]
                            {

SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration)

.WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)),

SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration)

.WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken))
                            }))));
}
```

Posljednja metoda naziva se *GenerateNamespaceCode*. Struktura imenskog prostora generira se metodom *NamespaceDeclaration* kojoj se prosljeđuje naziv imenskog prostora, a metodom *AddMembers* dodaje se generirani kôd klase sa svojstvima.

```
private NamespaceDeclarationSyntax GenerateNamespaceCode(string
formattedNamespace, ClassDeclarationSyntax generatedClass)
{
    return
SyntaxFactory.NamespaceDeclaration(SyntaxFactory.ParseName(formattedName
space))
                        .AddMembers(generatedClass);
}
```

Ukupni generirani kôd vraća se kao podatak tipa string klasi prozora *FetchedTablesWindow* koji se zatim prosljeđuje prozoru *SaveClassWindow*.

Ovaj prozor služi za odabir projekta u koji će se generirana klasa spremiti. Izgled prozora može se vidjeti na sljedećoj slici.



Slika 31: *SaveClassWindow* prozor

Odmah prilikom otvaranja prozora, poziva se metoda *GetProjectPath* klase *ClassSaver*. U toj metodi kreira se dijalog za odabir projektne datoteke. Dohvati se putanja do tog projekta te se pritiskom na gumb *Save Class* sprema generirana klasa u odabrani projekt. Za to se također koristi klasa *ClassSaver* i njezine metode. Ukoliko postoji više klasa za spremanje koji se žele sve odmah spremiti moguće je ostvariti pritiskom gumba *Save All Classes*. Ako se želi promijeniti projekt za spremanje klasa, može se ponovno otvoriti dijalog pritiskom gumba *Change Project*.

Klasa *ClassSaver* sadrži četiri metode. Jedna od njih, *GetProjectPath*, opisana je u prethodnom odlomku, a njezina implementacija vidljiva je na slici ispod. Ono što vrijedi spomenuti da će dijalog otvoriti mapu *source* ili *repos* ukoliko se nalaze u mapi korisnika računala.

```
public string GetProjectPath(string filePath)
{
    using (OpenFileDialog openFileDialog = new OpenFileDialog())
    {
        openFileDialog.Filter = "Project Files (*.csproj)|*.csproj";
        openFileDialog.Title = "Select Project File";

        if(filePath == "")
        {
            filePath =
Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
            string source = "source";
            string sourceDir = Path.Combine(filePath, source);
```

```
            if (Directory.Exists(filePath))
            {
                filePath = sourceDir;
            }
            string repos = "repos";
            string reposDir = Path.Combine(sourceDir, repos);
            if (Directory.Exists(filePath))
            {
                filePath = reposDir;
            }
        }
        openFileDialog.InitialDirectory = filePath;

        DialogResult result = openFileDialog.ShowDialog();

        if (result == DialogResult.OK)
        {
            return openFileDialog.FileName;
        }
        else
        {
            return "";
        }
    }
}
```

Preostale tri metode služe se manipulaciju projektom u koji se treba spremiti klasa i pozivanje metode za kreiranje datoteke za upis generiranog kôda.

Metoda *SetupProject* služi za kreiranje objekta Project, objekta klase *FileManager* i dohvaćanja mape u kojoj se projekt nalazi.

Generirani kôd klase mora se spremiti u datoteku kako bi klasa mogla biti spremljena u projekt. U metodi *SaveClass* poziva se metoda *CreateFile* kojoj se prosljeđuje putanja do lokacije gdje je potrebno spremiti datoteku i uz to generirani kôd.

Nakon što se kreira datoteka i doda u projekt, u metodi *SaveAndUnloadProject* projekt se sprema metodom *Save*. Potrebno ga je i ukloniti iz kolekcije projekata kako bi se prilikom sljedeće aktivacije generatora u ovu kolekciju mogao umetnuti odgovarajući projekt.

```
public void SetupProject(string projectPath)
```

```
{
    project = new Project(projectPath);
    fileManager = new FileManager();
    projectDirectory = Path.GetDirectoryName(projectPath);
}

public void SaveAndUnloadProject()
{
    project.Save();
    ProjectCollection.GlobalProjectCollection.UnloadProject(project);
}

public void SaveClass(string className, string generatedCode)
{
    className += ".cs";
    string filePath = Path.Combine(projectDirectory, className);

    fileManager.CreateFile(filePath, generatedCode);

    ProjectItem existingItem = project.GetItems("Compile")
                                .FirstOrDefault(item =>
item.EvaluatedInclude.Equals(className,
StringComparison.OrdinalIgnoreCase));

    if (existingItem == null)
    {
        project.AddItem("Compile", className);
    }
}
```

Spomenuta metoda *CreateFile* pripada klasi *FileManager*.

```
public class FileManager
{
    public void CreateFile(string filePath, string generatedCode)
    {
        try
        {
            using (var fileStream = new FileStream(filePath,
FileMode.Create))
            {
```

```
                using (var streamWriter = new StreamWriter(fileStream))
                {
                    streamWriter.Write(generatedCode);
                }
            }
        }
        catch (Exception ex)
        {
            System.Windows.MessageBox.Show("Error creating file: " +
ex.Message, "File Creation", (MessageBoxButton)MessageBoxButtons.OK,
(MessageBoxImage)MessageBoxIcon.Error);
        }
    }
}
```

Za kreiranje datoteke korišten je *FileStream*, dok za upis u datoteku *StreamWriter*. *FileStream* za prvi parametar prima proslijeđenu putanju do mjesta na kojem će se datoteka kreirati, a za drugi parametar potrebno je upisati modifikator otvaranja datoteke. Za ovu implementaciju rješenja iskorišten je modifikator „Create“. *StreamWriter* prima jedan parametar, a to je varijabla tipa *FileStream*. Prilikom kreiranja objekta proslijeđena je varijabla kreirana u prethodnom koraku, a kako bi se zapis izvršio, pozvana je metoda *Write* te njoj proslijeđena varijabla tipa *string* u kojoj je zapisana cijela struktura generirane klase.

Kako je opisano dijagramima u dizajnu rješenja, jedina razlika prilikom generiranja klasa iz tablica baze podataka je komunikacija s bazom i mapiranje podataka. Implementacija dohvaćanja metapodataka i mapiranja ostvarena je korištenjem sučelja i njihove realizacije.

U sučelju *IDatabaseMetadataFetcher* deklarirana je samo jedna metoda, *FetchTables* čiji je povratni tip lista objekata tipa *TableMetadata*, a ulazni parametar su pristupni podaci bazi podataka.

```
public interface IDatabaseMetadataFetcher
{
    List<TableMetadata> FetchTables(string connectionString);
}
```

IDatabaseDataTypeMapper sučelje također ima deklaraciju samo jedne metode, a to je metoda *MapDatabaseDataTypeToCSharpType*. Njezin povratni tip je tip podatka *string*, dok je ulazni parametar naziv tipa podatka baze.

```
public interface IDatabaseDataTypeMapper
```

```
{
    string MapDatabaseDataTypeToCSharpType(string databaseType);
}
```

Implementacija ovih sučelja ostvarena je u dvjema klasama, SSMSMetadataFetcher i SSMSDataTypeMapper. Prefiks SSMS označava da su ove klase implementirane za bazu podataka u *SQL Server Management Studiju*. Metoda *FetchTables* klase *SSMSMetadataFetcher* implementirana je na način da dohvaća sve tablice koje se nalaze u bazi podataka i sprema njihove nazive u listu tipa *TableMetadata*, odnosno atribute „Name“ tog tipa. Za to je korištena *SqlConnection* klasa kojom se ostvaruje veza s bazom za što je potrebno proslijediti pristupne podatke za povezivanje. Klasom *SqlCommand* kreira se naredba prema bazi za dohvaćanje određenih podataka.

```
public List<TableMetadata> FetchTables(string connectionString)
{
    string query = "SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE
TABLE_TYPE = 'BASE TABLE'";

    using (SqlConnection connection = new
SqlConnection(connectionString))
    {
        command = new SqlCommand(query, connection);
        connection.Open();
        reader = command.ExecuteReader();
        while (reader.Read())
        {
            var table = new TableMetadata();
            table.Columns = new List<ColumnMetadata>();
            table.Name = reader["TABLE_NAME"].ToString();
            tables.Add(table);
        }
        reader.Close();

        foreach (var table in tables)
        {
            FetchColumns(table);
        }

        connection.Close();
    }
```

```
    return tables;
}
```

Nakon što su zapisane sve tablice, prolazi se kreiranom listom kako bi se zapisali podaci o stupcima pojedine tablice, a za to služi metoda *FetchColumns*. Metoda na temelju naziva tablice pronalazi sve stupce koji se u njoj nalaze te ih zapisuje u objekt tipa *ColumnMetadata* koji će prilikom generiranja kôda postati svojstvo klase.

```
private void FetchColumns(TableMetadata table)
{
    command.CommandText = $"SELECT * FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = '{table.Name}'";
    reader = command.ExecuteReader();

    while (reader.Read())
    {
        column = new ColumnMetadata();
        column.Name = reader["COLUMN_NAME"].ToString();

        column.DataType =
dataTypeMapper.MapDatabaseDataTypeToCSharpType(reader["DATA_TYPE"].ToStr
ing());

        table.Columns.Add(column);
    }
    reader.Close();
}
```

Metoda *MapDatabaseDataTypeToCSharpType* klase *SSMSDataTypeMapper* provjerava proslijeđeni tip podatka baze i vraća naziv tipa podatka koji mu odgovara u C# programskom jeziku.

```
public class SSMSDataTypeMapper : IDatabaseDataTypeMapper
{
    public string MapDatabaseDataTypeToCSharpType(string dataType)
    {
        switch (dataType.ToLower())
        {
            case "bigint":
                return "long";
            case "binary":
```

```
case "varbinary":
    return "byte[]";
case "bit":
case "bool":
    return "bool";
case "char":
case "character":
    return "char";
case "nchar":
case "text":
case "ntext":
case "varchar":
case "nvarchar":
case "string":
    return "string";
case "date":
case "datetime":
case "datetime2":
    return "DateTime";
case "decimal":
case "numeric":
    return "decimal";
case "float":
    return "float";
case "double":
    return "double";
case "int":
case "integer":
    return "int";
case "real":
    return "float";
case "smallint":
    return "short";
case "time":
    return "TimeSpan";
case "tinyint":
    return "byte";
case "uniqueidentifier":
    return "Guid";
default:
```

```
                return "";
            }
        }
    }
```

Ostatak toka isti je kao i prilikom generiranja iz ručnog unosa, otvara se prozor *SaveClassWindow* i generirana klasa sprema se u projekt. Kako to uistinu izgleda prikazat će se u sljedećem poglavlju.

5.3. Primjeri korištenja alata

Radi lakšeg razumijevanja funkcioniranja alata, opisana će biti dva primjera, tekstom i slikama. Prvi primjer bit će primjer generiranja dvije klase iz tablica baze podataka. Baza sadrži dvije tablice. Prva tablica naziva se *MobilePhones* te sadrži stupce *Id*, *Name*, *RAM*, *LaunchDate* i *OS*. Dizajn tablice nalazi se na sljedećoj slici:

	Column Name	Data Type	Allow Nulls
🔑	Id	int	☐
	Name	varchar(50)	☑
	RAM	int	☑
	LaunchDate	date	☑
	OS	varchar(50)	☑

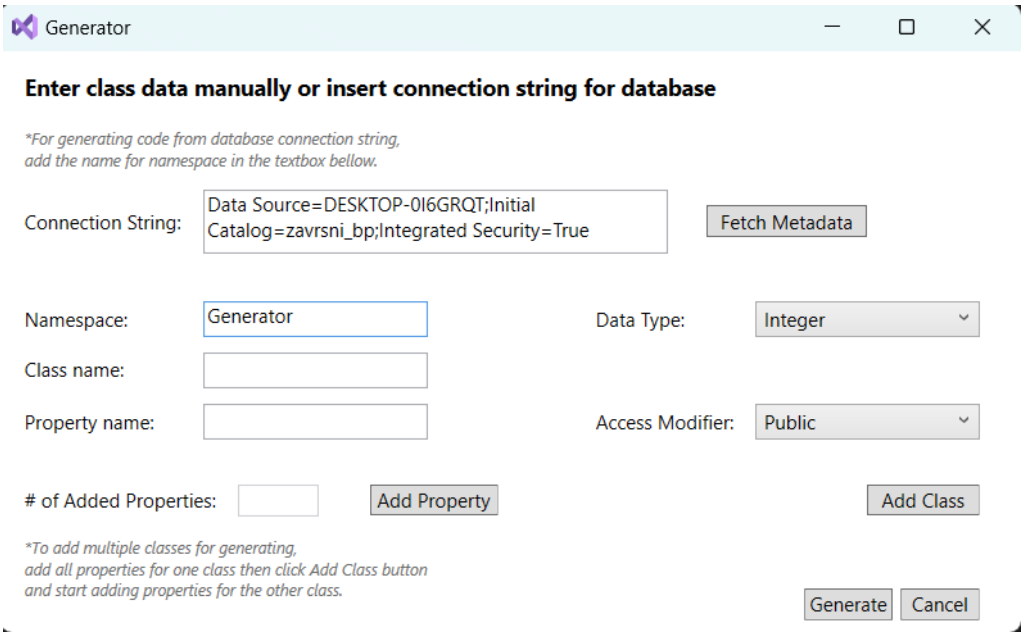
Slika 32: Tablica *MobilePhones*

Druga tablica naziva se *Contacts* i sadrži stupce *Id*, *Name*, *TelephoneNumber* i *Email*.

	Column Name	Data Type	Allow Nulls
🔑	Id	int	☐
	Name	nvarchar(50)	☑
	TelephoneNumber	int	☑
	Email	nvarchar(50)	☑

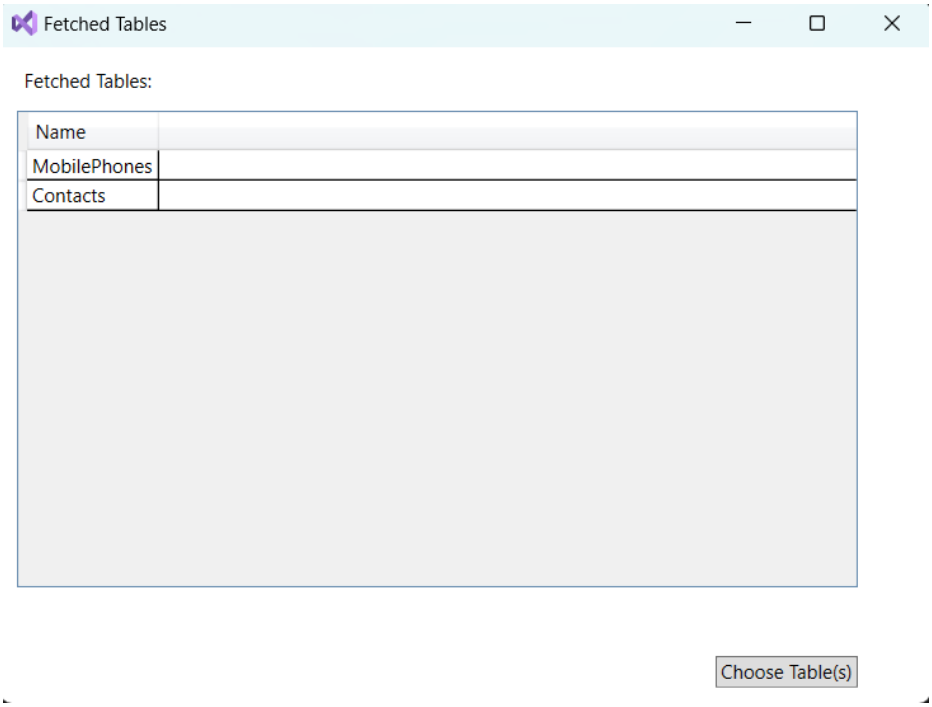
Slika 33: Tablica *Contacts*

Nakon pokretanja alata, otvara se *MainWindow* prozor i u prostor predodređen za unos pristupnih podataka za bazu, potrebno je unijeti niz za povezivanje s testnom bazom.



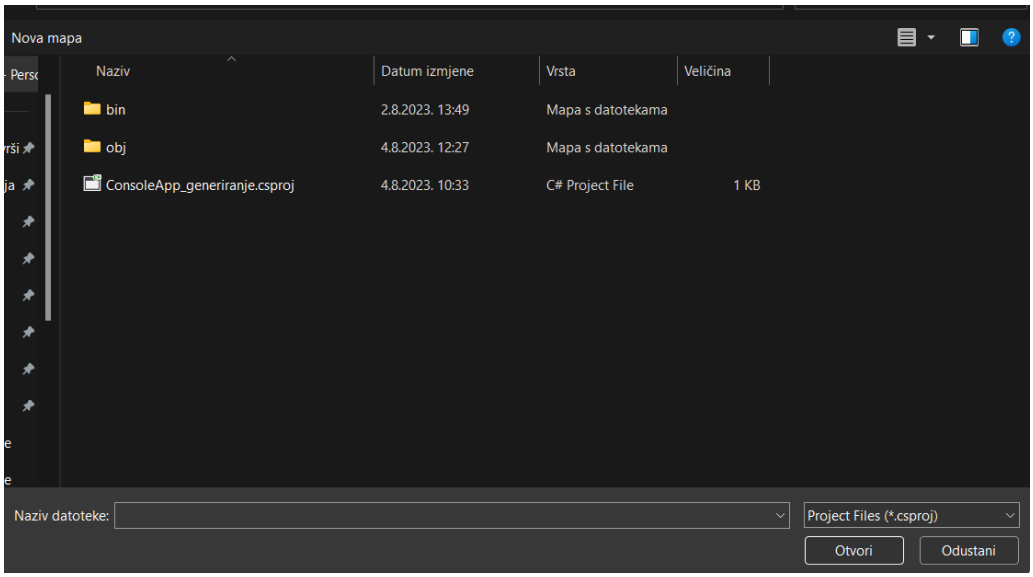
Slika 34: *MainWindow* prozor s upisanim podacima za povezivanje s bazom

Nakon pritiska na gumb *Fetch Metadata* otvara se prozor *Fetched Tables* s prikazom svih dohvaćenih tablica iz baze.



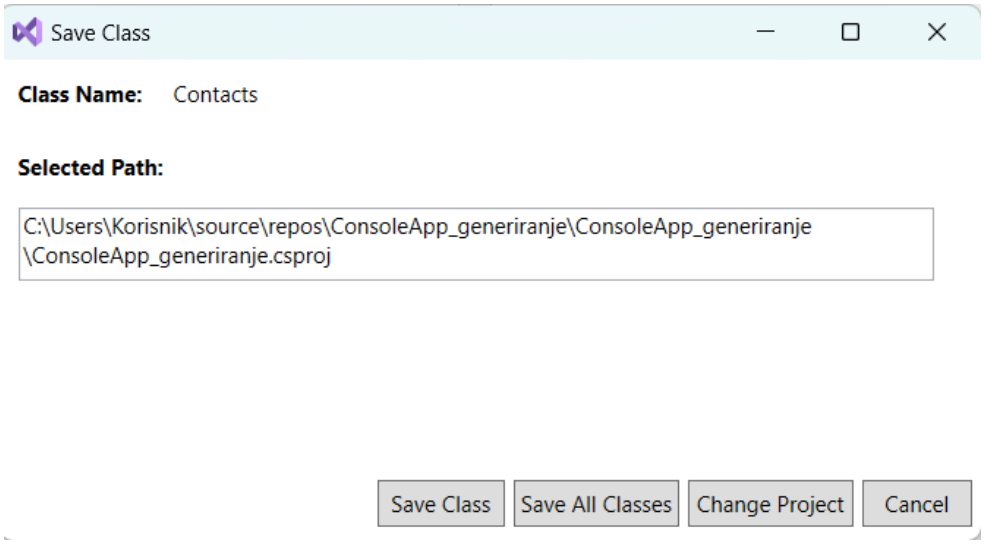
Slika 35: Dohvaćene tablice iz baze

Na prozoru su prikazani nazivi dviju tablica, *MobilePhones* i *Contacts*. Nakon odabira obje tablice i pritiska na gumb *Choose Table(s)*, otvara se prozor *SaveClassWindow* i dijalog za odabir projekta u koji će se spremiti generirane klase. Zapravo se odabire .csproj datoteka i sprema se putanja koja vodi do nje.



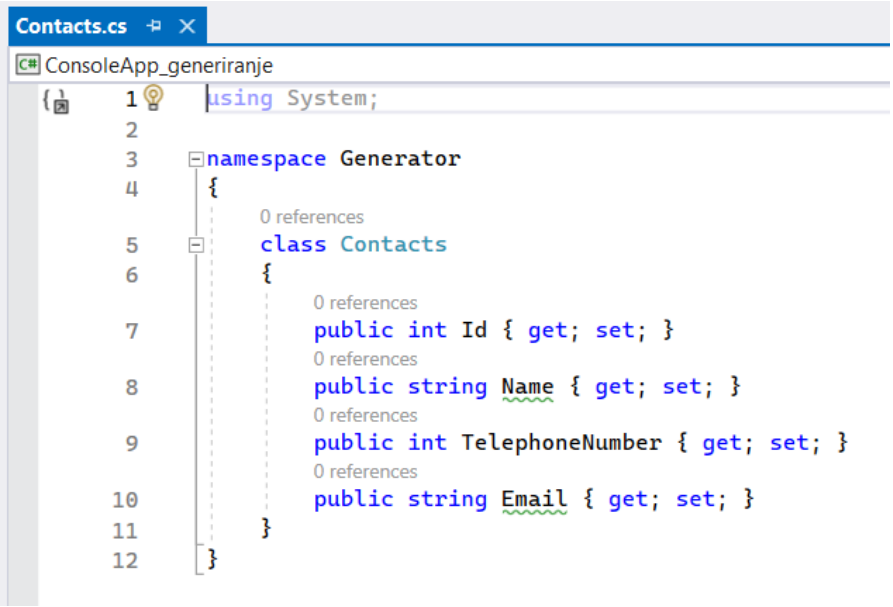
Slika 36: Dijalog za odabir projektne datoteke

Projekt je odabran i putanja do njegove projektne datoteke prikazana je u Save Class prozoru, točnije unutar tekstualnog polja. Pritiskom na gumb *Save Class* sprema se prva generirana klasa u projekt te na red dolazi druga. Pritiskom na gumb *Save All Classes* moguće je obje odmah spremiti.



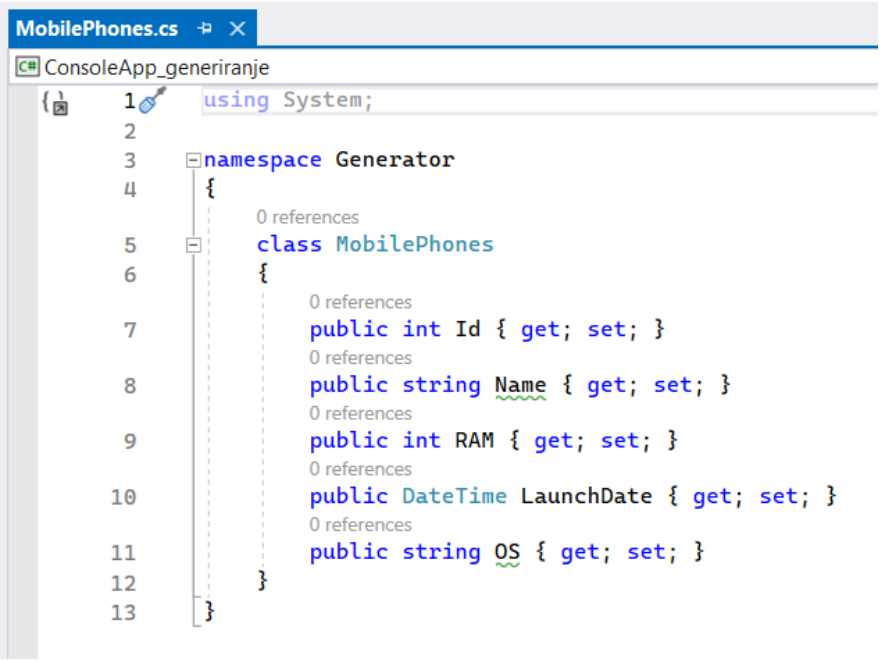
Slika 37: Prozor *Save Class* za klasu *Contacts*

U nastavku je prikazano kako klasa *Contacts* izgleda nakon što je spremljena u željeni projekt.



Slika 38: Klasa *Contacts*

I klasa *MobilePhones* također je spremljena, sadržavajući sve stupce i njihova svojstva iz tablice baze podataka.



Slika 39: Klasa *MobilePhones*

Kako bi se primijetila razlika unutar .csproj datoteke, na sljedećoj je slici prikazano kako ona izgleda prije dodavanja generiranih klasa:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Build.Framework" Version="17.6.3" />
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="4.6.0" />
    <PackageReference Include="Microsoft.CodeAnalysis.Workspaces.Common" Version="4.6.0" />
    <PackageReference Include="Microsoft.CodeAnalysis.Workspaces.MSBuild" Version="4.6.0" />
  </ItemGroup>
</Project>
```

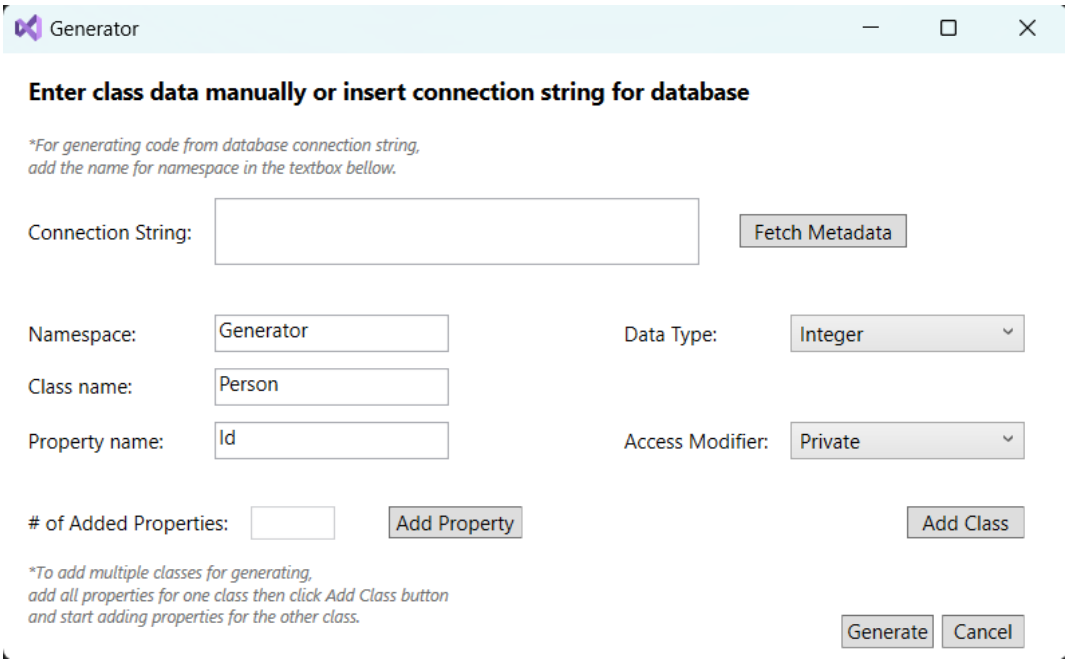
Slika 40: Projektna datoteka prije dodavanja generiranih klasa u projekt

Ako zavirimo u .csproj datoteku projekta nakon dodavanja klasa, uočljivo je da je dodana nova grupa stavaka *ItemGroup* te su kao *Compile* dio unutar te grupe dodane reference na generirane klase u projektu.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Build.Framework" Version="17.6.3" />
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="4.6.0" />
    <PackageReference Include="Microsoft.CodeAnalysis.Workspaces.Common" Version="4.6.0" />
    <PackageReference Include="Microsoft.CodeAnalysis.Workspaces.MSBuild" Version="4.6.0" />
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Contacts.cs" />
    <Compile Include="MobilePhones.cs" />
  </ItemGroup>
</Project>
```

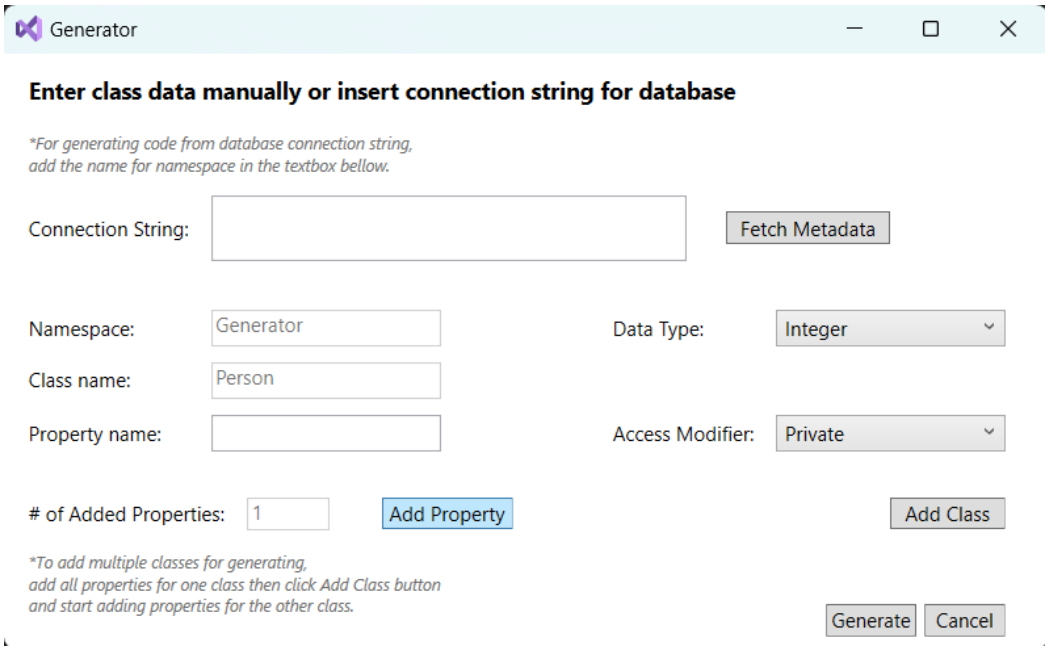
Slika 41: Projektna datoteka nakon dodavanja generiranih klasa u projekt

Drugi primjer pokazat će generiranje klase iz ručno unesenih podataka. Na početku, alat je pokrenut i unutar prozora *MainWindow* sva tekstualna polja su prazna. Dodavanjem naziva za imenski prostor, klasu i svojstvo te odabir tipa podatka i modifikatora pristupa za svojstvo, dobiva se sljedeći prikaz.



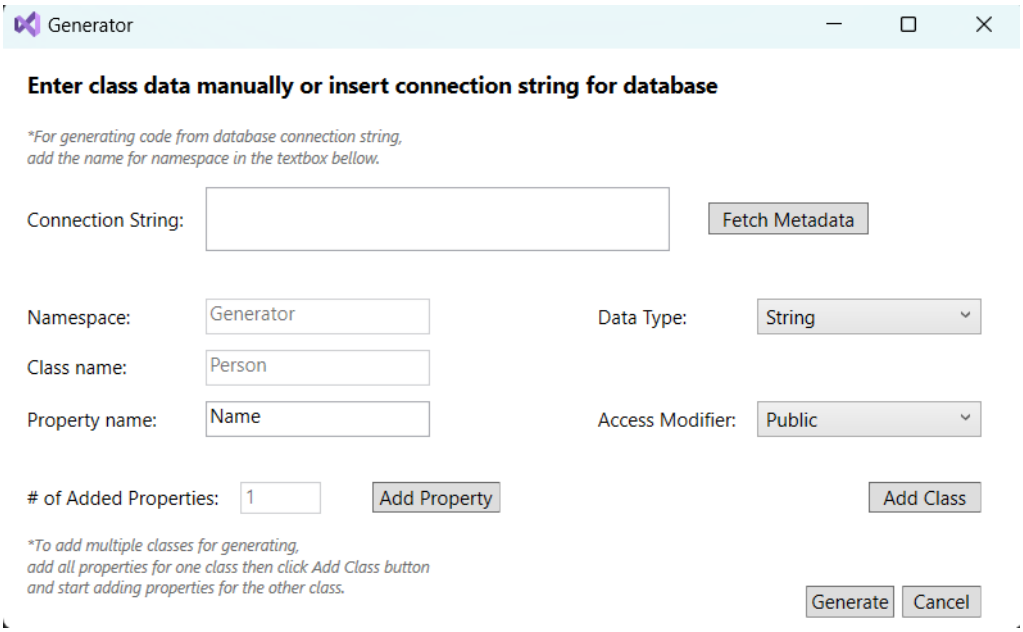
Slika 42: *MainWindow* prozor s ručno upisanim podacima

Kako bi se dodalo svojstvo *Id* i onemogućila promjena imenskog prostora i naziva klase, pritišće se gumb *Add Property* i broj dodanih svojstava sad je jednak jedan.



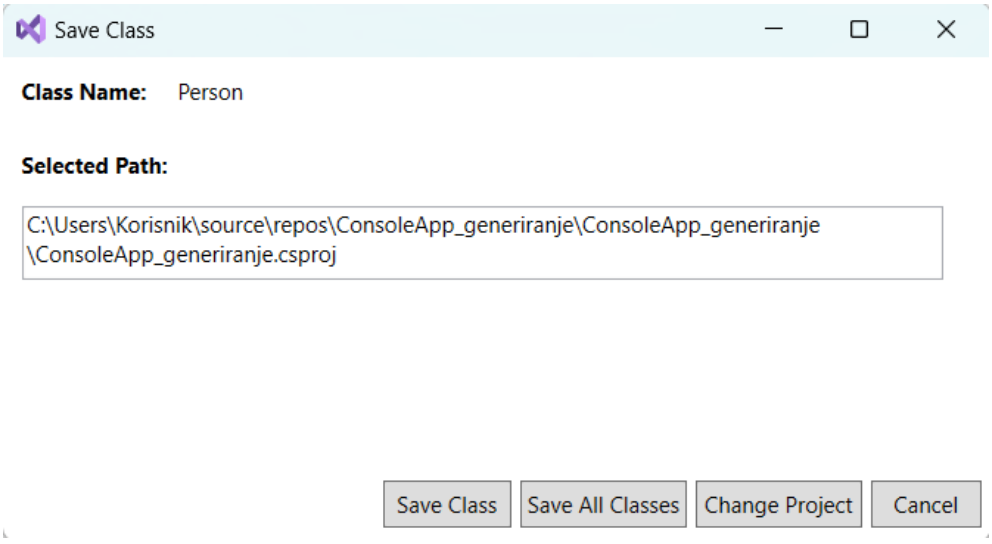
Slika 43: Broj dodanih svojstava

Za potrebe primjera, dodat će se i svojstvo *Name*.



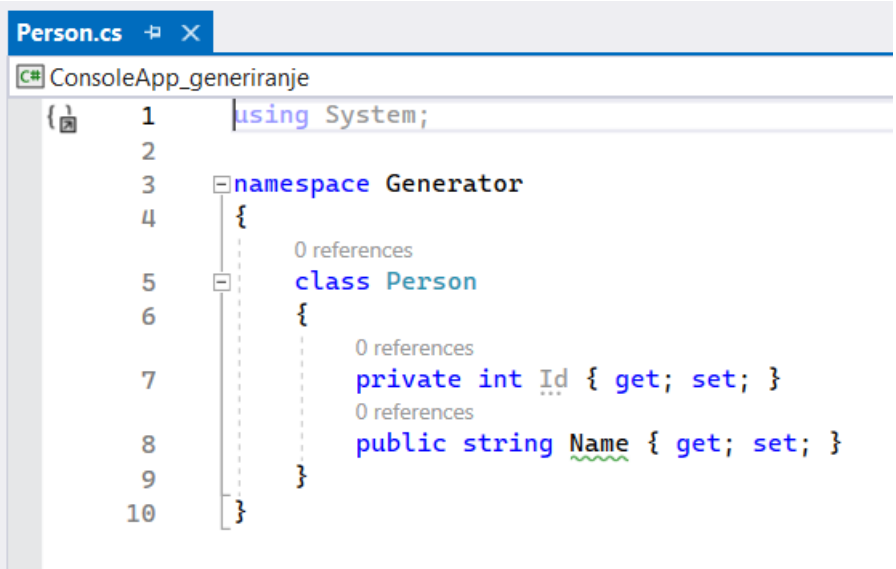
Slika 44: Svojstvo *Name*

Add Class gumb sprema klasu zajedno sa svojim svojstvima. Pritiskom na gumb *Generate* ponovno se otvara dijalog za odabir projektne datoteke, ali sad je, naravno, unutar *SaveClassWindow* prozora naziv klase koja je ručno dodana.



Slika 45: Prozor *Save Class* za klasu *Person*

U projekt je uspješno dodana generirana klasa *Person*.



Slika 46: Klasa *Person*

6. Zaključak

Kroz razradu teme dokazano je da je Roslyn kompilator uistinu moćna platforma koja pruža mnogo mogućnosti te kontrole prilikom generiranja programskog kôda. Osim što se njime mogu izraditi samostalni alati za generiranje, njegova integracija u razvojnu okolinu Visual Studio pruža značajne prilike programerima koje do sad nisu imali.

Način na koji je kompilator implementiran, ostvarena je mogućnost pregleda i analize vlastitog kôda, refaktoriranja i unaprjeđenja kôda. Moguće je uključiti se u sami proces kompilacije, uvidjeti sve korake postupka i osigurati da rješenje radi točno kako je zamišljeno u svakom pogledu. Uz ponuđene mogućnosti, otvara se mogućnost izrade još mnogo korisnih alata pomoću ove platforme.

Razrada teme i alata omogućena je velikom količinom materijala koji se mogu pronaći u sadržajnim bibliotekama akademskih članaka poput Google Scholar alata i dokumentacije razvojnih timova Roslyn kompilatora koje pruža tvrtka Microsoft. Za izradu alata korištena je razvojna okolina Visual Studio i testna baza locirana na SQL Serveru, a čitavo verzioniranje kôda odrađeno je kroz GitHub platformu.

Dokazano je da generatori kôda uistinu smanjuju vrijeme, povećavaju produktivnost, daju uvid u proces i korake kompilacije, daju mnogo mogućnosti za generiranje i analizu jer mogu otkriti greške u kôdu kontinuiranom kompilacijom što predstavlja ogromnu prednost. Nedostatak je i dalje da alati za generiranje mogu brzo postati kompleksni i problem za održavanje ukoliko se funkcionalnosti generatora žele promijeniti.

Korištenje Roslyna u izradi alata za generiranje programskog kôda bilo je ugodno iskustvo. Sintaksa korištenih klase i metoda za izradu alata, *SyntaxTree* i *SyntaxFactory*, bila je lako razumljiva i intuitivna. Ostvarivanje komunikacije s bazom također je bilo jednostavno kroz uporabu pristupnih podataka za povezivanje. Najveći trošak vremena bio je problem ostvarivanja manipulacije nad projektima. Budući da je zahtjev alata bio spremanje generiranih klasa u željeni projekt, trebalo je tražiti od strane korisnika putanju do projekta nakon čega je trebalo mijenjati projektnu datoteku, *.csproj*, kako bi projekt prepoznao novododanu klasu. Kreiranje datoteke i zapis generiranog kôda nije bio problematičan dio zbog korištenja klasa *FileStream* i *StreamWriter* koje su znatno olakšale rad s datotekama.

Popis literature

[1] Kelner Maia *i ostali*, .„NET Compiler Platform (‚Roslyn') extensibility", *Microsoft Learn*, 10. ožujak 2023. https://learn.microsoft.com/en-us/visualstudio/extensibility/dotnet-compiler-platform-roslyn-extensibility?view=vs-2022 (pristupljeno 25. srpanj 2023.).

[2] „Code Generation", *Teach Computer Science*. https://teachcomputerscience.com/code-generation/#What_is_Code_Generation (pristupljeno 25. srpanj 2023.).

[3] E. Syriani, L. Luhunu, i H. Sahraoui, „Systematic mapping study of template-based code generation", u *Computer Languages, Systems & Structures*, Pergamon, lip. 2018, str. 43–62. doi: 10.1016/J.CL.2017.11.003.

[4] J. Cabot, „The Last One – A code generator for BASIC from 1981", 21. siječanj 2015. https://modeling-languages.com/last-one-code-generator-basic-1981/ (pristupljeno 25. srpanj 2023.).

[5] J. Herrington, *Code Generation In Action*. 2003.

[6] A. Mehmood i D. N. A. Jawawi, „Aspect-oriented model-driven code generation: A systematic mapping study", u *Information and Software Technology*, Elsevier, velj. 2013, str. 395–411. doi: 10.1016/J.INFSOF.2012.09.003.

[7] S. Jörges, *Construction and Evolution of Code Generators*. 2013. Pristupljeno: 29. srpanj 2023. [Na internetu]. Dostupno na: http://ndl.ethernet.edu.et/bitstream/123456789/34487/1/144.pdf

[8] I. Rončević, „awesome roslyn". https://github.com/ironcev/awesome-roslyn#source-generators (pristupljeno 31. srpanj 2023.).

[9] „Roslyn GitHub repozitorij". https://github.com/dotnet/roslyn (pristupljeno 31. srpanj 2023.).

[10] M. Torgersen, „How Microsoft rewrote its C# compiler in C# and made it open source", 26. rujan 2018. https://medium.com/microsoft-open-source-stories/how-microsoft-rewrote-its-c-compiler-in-c-and-made-it-open-source-4ebed5646f98 (pristupljeno 31. srpanj 2023.).

[11]. „NET Compiler Platform (‚Roslyn') Overview". https://github.com/dotnet/roslyn/blob/main/docs/wiki/Roslyn-Overview.md (pristupljeno 31. srpanj 2023.).

[12] N. Harrison, *Code Generation with Roslyn*. Apress, 2017. doi: 10.1007/978-1-4842-2211-9.

[13] B. Wagner *i ostali*, „Source Generators", *Microsoft Learn*, 25. lipanj 2023. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview (pristupljeno 30. srpanj 2023.).

[14] „Source Generators Cookbook". https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md (pristupljeno 02. kolovoz 2023.).

[15] „Incremental Generators". https://github.com/dotnet/roslyn/blob/main/docs/features/incremental-generators.md (pristupljeno 02. kolovoz 2023.).

[16] „Syntax Visualizer Overview". https://github.com/dotnet/roslyn/blob/main/docs/wiki/Syntax-Visualizer.md (pristupljeno 03. kolovoz 2023.).

[17] „SyntaxFactory Class". https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntaxfactory?view=roslyn-dotnet-4.6.0 (pristupljeno 02. kolovoz 2023.).

[18] „Overview of source code analysis". https://learn.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022 (pristupljeno 05. kolovoz 2023.).

[19] „Overview of .NET source code analysis“. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview?tabs=net-7#code-style-analysis (pristupljeno 06. kolovoz 2023.).
[20] A. Del Sole, *Roslyn Succinctly*. Syncfusion, 2016. Pristupljeno: 30. srpanj 2023. [Na internetu]. Dostupno na: https://www.syncfusion.com/succinctly-free-ebooks/roslyn/project-roslyn-the-net-compiler-platform
[21] Gordon Steve, „GETTING STARTED WITH THE ROSLYN APIS: WRITING CODE WITH CODE“, *Pluralsight*, 22. ožujak 2021. https://www.stevejgordon.co.uk/getting-started-with-the-roslyn-apis-writing-code-with-code (pristupljeno 25. srpanj 2023.).

Popis slika