Pregled mogućnosti prevoditeljske platforme .NET Roslyn

Trifunović, Goran

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Goran Trifunović

Pregled mogućnosti prevoditeljske platforme .NET Roslyn

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Goran Trifunović

Matični broj: 44058/5–IZV

Studij: Poslovni sustavi

Pregled mogućnosti prevoditeljske platforme .NET Roslyn

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Robert Kudelić

Varaždin, lipanj 2020.

Goran Trifunović

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski 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

U ovom radu obradio sam prevoditeljsku platformu .Net (engl. .NET Compiler Platform), poznatu i pod nazivom Roslyn. U prvom dijelu rada opisuje se teoretska osnova na kojoj se Roslyn zasniva. Tu se opisuje struktura Roslyna, opisuju se faze prevoditelja i utvrđuju temeljni principi na kojima se zasniva. Nakon toga se prelazi u primjere primjene nekih od API-ja Roslyna, a zatim se prelazi na glavnu značajku Roslyna, pisanje analizatora s ispravkom koda i objavom istoga na trgovinu Visual Studio Marketplace. U radu sam osim primarnog doprinosa pokušao dati i doprinos hrvatskom govornom području – primarno kroz agregiranje poznatih prijevoda za strane termine na jednom mjestu, a sekundarno i predlaganjem novih hrvatskih termina za koje trenutno ili nema uvriježenog prijevoda ili ih tijekom izrade rada nisam pronašao.

Ključne riječi: prevoditelj, stablo sintakse, semantički model, analizator s ispravkom koda, čvorovi sintakse, cjevovod prevoditelja, nepromjenjivost

Sadržaj

1. Uvod

Prevoditeljska platforma .NET ili Roslyn je jedan jak niz API-ja koji nam dozvoljava da razumijemo proces prevođenja programa i upravljamo s dijagnostikom uređivača, točnije da naređujemo što se smatra greškom, krivom praksom ili lošom navikom.

Za obradu ove teme nema baš puno izvora, naprotiv, malo je izvora, a najkorisniji su ispali video izvori za učenje pisanja samog koda u Roslynu i službeni izvori s Githuba, sa stranice projekta i „Microsoft Docs“ stranica. Knjige koje obrađuju Roslyn su malobrojne i po mom mišljenju, nisu ni dobre ili su precijenjene za sadržaj koji obuhvaćaju. Glavni izvor motivacije za ovaj rad je bio raditi na nečemu što je meni bilo potpuno nepoznato prije početka rada.

U ovom radu, prije nego što li krenemo sa praktičnom primjenom Roslyna i pisanjem samog analizatora s ispravkom koda, prvo će biti predstavljeni teoretski koncepti Roslyna,. Pokušati ću prikazati mogućnosti Roslyna i koliko nam Roslyn može olakšati posao, ako ga pametno koristimo.

2. Teorija i koncepti Roslyna

Tradicionalno, prevoditelji su crne kutije, izvorni kod uđe na ulaz prevoditelja, „magija“ se dogodi negdje u prevoditelju, a sklopovi i objektne datoteke su izlaz prevoditelja. Kako prevoditelji rade, tako oni razvijaju i razumijevanje o kodu koji procesiraju, no to razumijevanje je dostupno samo „čarobnjacima“ za implementaciju prevoditelja. Nakon što je prevedeni izlaz prevoditelja proizveden, sve informacije su zaboravljene. [1]

Desetljećima, ovaj pogled na prevoditelje dobro nas je služio, ali on nije više dovoljan. Sve više i više se oslanjamo na mogućnosti integriranog razvojnog okruženja (engl. Integrated development enviroment, IDE) kao što su InteliSense, refaktoriranje, pametno preimenovanje, „Find all references“, „Go To Definition“, sve da bismo povećali produktivnost. Oslanjamo se na alate za analizu koda da bismo poboljšali kvalitetu koda i na generatore koda da bismo si pomogli pri izgradnji aplikacija. Kako se ti alati razvijaju i postaju sve pametniji, tako im je i sve više potrebno dublje znanje koda koje samo prevoditelji posjeduju. Upravo to je misija Roslyna – otvaranje crnih kutija i dijeljenje informacija o kodu, koje posjeduju samo prevoditelji, sa alatima i korisnicima. Umjesto da budu neprozirni prevoditelji za ulaz izvornog koda i izlaz objektnog koda, prevoditelji preko Roslyna postaju platforme - API-ji koje možete koristiti za zadatke povezane s kodom u svojim alatima i aplikacijama. [1]

Prijelaz na prevoditelje kao platforme dramatično spušta prepreku ulasku u stvaranje alata i aplikacija usmjerenih na kod. To stvara brojne mogućnosti za inovacije u područjima kao što su metaprogramiranje, stvaranje i transformacija koda, interaktivna uporaba jezika C # i VB i ugradnja C# i VB u jezike specifične za domenu. [1]

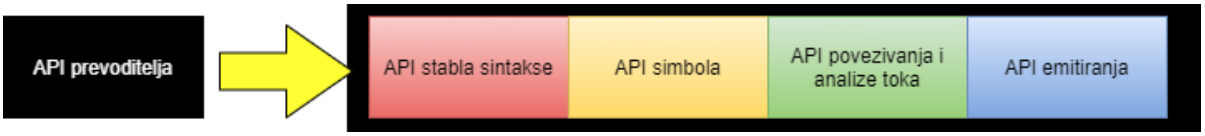
2.1. Izlaganje API-ja prevoditelja

Roslyn izlaže analizu koda prevoditelja C# tako što pruža sloj API-ja što zrcali tradicionalni cjevovod prevoditelja.

Slika 1: Cjevovod prevoditelja (Compiler pipeline, bez dat.)

Svaka faza ovog cjevovoda je sada zasebna komponenta. Cjevovod prevoditelja se sast oji od četiri faza:

1. Faza raščlanjivanja - izvor se žetonizira i raščlanjuje u sintaksu koja slijedi jezičnu gramatiku.
2. Faza deklaracije – deklaracije iz izvornih i uvezenih metapodataka analiziraju se u obliku imenovanih simbola
3. Faza povezivanja – identifikatori u kodu podudaraju se sa simbolima
4. Faza emitiranja – sve informacije koje je sastavio prevoditelj emitiraju se kao sklop



Slika 2: API prevoditelja (Compiler pipeline API, bez dat.)

U skladu sa svakom od ovih faza pojavljuje se objektni model, koji omogućava pristup informacijama u toj fazi. Faza raščlanjivanja izložena je kao stablo sintakse, faza deklaracije izložena je kao hijerarhijska tablica simbola, faza povezivanja kao model koji izlaže rezultate semantičke analize prevoditelja i faza emitiranja kao API koji proizvodi IL bajtne kodove. Svaki prevoditelj kombinira te komponente zajedno kao jedinstvenu cjelovitu cjelinu. U Visual Studiou, stabla sintakse koriste značajke za izdvajanje i oblikovanje koda, hijerarhijsku tablicu simbola koriste preglednik objekata (engl. Object Browser) i navigacijske značajke, semantički model koristi se za refaktoriranje i za Idi na definiciju (engl. Go To Definition), a Uredi i nastavi (engl. Edit and Continue) koristi sve objektne modele, uključujući i API emitiranja. [1]

2.2. Slojevi API-ja

Roslyn se sastoji od dva glavna sloja API-ja, API-ji prevoditelja (engl. Compiler APIs) i API-ji radnih prostora (engl. Workspaces APIs). API-ji prevoditelja još sadržavaju API za skripte (engl. Scripting API) i dijagnostički API. [1]

2.2.1. API prevoditelja

Sloj prevoditelja sadrži semantičke i sintaktičke objektne modele, koji odgovaraju informacijama izloženim u svakoj fazi cjevovoda prevoditelja. Sloj prevoditelja također sadrži i nepromjenjivi snimak pojedinačnog poziva prevoditelja, uključujući reference za sastavljanje, opcije prevoditelja i datoteke izvornog koda. Postoje dva različita API-ja, za C# i Visual Basic, svaki podešen za svoj jezik. Ovaj sloj nema nikakvih ovisnosti na komponente Visual Studia. [1]

API-ji za skripte

Kao dio sloja prevoditelja, tim koji radi na Roslynu je stvorio API-je za hosting / skriptiranje za izvršavanje isječaka koda i skupljanje konteksta izvršavanja. REPL koristi ove API-je. [1]

Dijagnostički API

Kao dio svoje analize prevoditelj može stvoriti skup dijagnostike koji pokriva sve, od sintaksne, semantičke i određene pogreške u dodjeljivanju do različitih upozorenja i informacijske dijagnostike. Sloj API-ja prevoditelja otkriva dijagnostiku pomoću proširivog API-ja, koji omogućava da se analizirani elementi definirani od strane korisnika uključe u prevođenje i dijagnostiku definiranu od strane korisnika, poput one proizvedene pomoću alata kao što su *StyleCop* ili *FxCop*, a koje se proizvode uz dijagnostiku koju definira prevoditelj. Izrada dijagnostike na ovaj način ima korist od prirodne integracije s alatima kao što su MSBuild i Visual Studio koji ovise o dijagnostici za iskustva poput zaustavljanja izrade zasnovane na politici i prikazivanja stvarnih kretnji u uređivaču i predlaganja ispravaka koda. [1]

2.2.2. API-ji radnih prostora

Sloj radnih prostora sadrži API-je radnih prostora, koji služe kao početna točka za refaktoriranje i analizu koda preko cijelog rješenja. Pomaže vam u organiziranju svih informacija o projektima u rješenju u model jednog objekta, nudeći vam izravan pristup objektnim modelima prevoditelja, bez potrebe za analizom datoteka, konfiguriranjem opcija ili upravljanja projektom, ovisno o projektu. [1]

Pored toga, sloj radnih prostora obrađuje skup najčešće korištenih API-ja koji se koriste pri implementaciji alata za analizu koda i alata za refaktoriranje koji djeluju unutar okruženja domaćina poput IDE-ja Visual Studio, poput API-ja za pronalaženje svih referenci, oblikovanja i generiranja koda. Također kao i sloj prevoditelja, sloj radnih prostora ne ovisi o komponentama Visual Studia. [1]

2.3. Rad sa sintaksom

Najosnovnija struktura podataka koju izlažu API-ji prevoditelja je stablo sintakse. Ova stabla predstavljaju leksičku i sintaktičku strukturu izvornog koda. Služe u dvije važne svrhe: [1]

1. Da bi dopustili alatima, kao što su IDE, dodaci, alati za analizu koda i refaktoriranje, da vide i obrade sintaktičku strukturu izvornog koda korisničkog projekta

2. Da bi omogućili alatima, kao što su refaktoriranja i IDE, da stvaraju, izmjenjuju i preuređuju izvorni kod na prirodan način, bez korištenja izravnih izmjena teksta. Stvaranjem i manipuliranjem stablima sintakse, alati mogu lako stvoriti i preurediti izvorni kod.

2.3.1. Stablo sintakse

Stablo sintakse je osnovna struktura korištena za prevođenje, analizu koda, povezivanje, refaktoriranje, značajke IDE-a i generiranje koda. Nije razumljiv nijedan dio izvornog koda bez njegovog prvobitnog identificiranja i razvrstavanja u jedan od mnogih poznatih strukturnih jezičnih elemenata. [1]

Stabla sintakse imaju tri ključna atributa:

- Prvi atribut je taj da stablo sintakse sadrži sve izvorne informacije u potpunoj vjernosti, što znači da stablo sintakse sadrži svaki podatak koji se nalazi u izvornom kodu, svaku gramatičku konstrukciju, svaki leksički znak i sve ostalo između, uključujući bijeli prostor, komentare i smjernice pretprocesora. Na primjer, svaki doslovni izraz koji se spominje u izvoru predstavljen je točno onako kako je i upisan. Stabla sintakse predstavljaju i pogreške u izvornom kodu kada je program nepotpun ili neispravan, predstavljajući preskočene ili nedostajuće žetone u stablu sintakse.

- Drugi atribut stabla sintakse omogućen je prvim atributom; stablo sintakse dobiveno iz raščlanjivača (Eng. Parser) potpuno se može zaokružiti natrag u izvorni kod iz kojeg je raščlanjen, što znači da je moguće iz bilo kojeg čvora sintakse dobiti tekstualni prikaz

pod-stabla ukorijenjenog na tom čvoru. Ovaj atribut omogućava da se stabla sintakse koriste kao način za izgradnju i uređivanje izvornog koda. Stvaranjem stabla stvara se ekvivalentni tekst, a uređivanjem sintakse stabla, izrađivanjem novog stabla iz promjena postojećeg stabla, uređuje se izvorni tekst.

- Treći atribut stabla sintakse je taj da je ono nepromjenjivo i sigurno u dretvama. To znači da nakon što smo dobili stablo, ono je snimak trenutnog stanja koda i nikada se ne mijenja. To omogućava da više korisnika u isto vrijeme rade na istom stablu sintakse, u različitim dretvama bez zaključavanja i dupliciranja. Kako su stabla nepromjenjiva i ne mogu se izvršiti nikakve izmjene izravno na stablu, tvorničke metode pomažu u stvaranju i izmjeni stabala, stvaranjem dodatnih snimki stabla. Stabla su učinkovita u upotrebi temeljnih čvorova, tako da se nova verzija može obnoviti brzo i s malo dodatne memorije.

Stablo sintakse je doslovno struktura podataka stabla, gdje ne-krajnji strukturni elementi nadređuju ostale elemente. Svako stablo sintakse sastoji se od čvorova, žetona i sitnica. [1]

2.3.2.Čvorovi sintakse

Čvorovi sintakse su jedan od osnovnih elemenata stabla sintakse. Ovi čvorovi predstavljaju sintaktičke konstrukcije poput deklaracija, izjava, članova i izraza. Svaka kategorija čvorova sintakse predstavljena je zasebnom klasom izvedenom iz *SyntaxNode*. Skup klasa čvorova nije proširiv. [1]

Svi čvorovi sintakse nisu krajnji čvorovi u stablu sintakse, što znači da uvijek imaju i druge čvorove i žetone kao djecu. Kao dijete drugog čvora, svaki čvor ima nadređeni čvor kojem se može pristupiti preko roditeljskog svojstva. Zato što su čvorovi i stabla nepromjenjivi, roditelj čvora se nikad ne mijenja. Korijen stabla ima roditelj *null*. [1]

Svaki čvor ima metodu *ChildNodes*, koja vraća listu čvorova djece redoslijedno u odnosu na njihovu poziciju u izvornom tekstu. Ova lista ne sadrži žetone. Svaki čvor također ima i zbirku *Descendant* metoda, kao što su *DescendantNodes*, *DescendantTokens* ili *DescendantTrivia*, koji predstavljaju listu svih čvorova, žetona i sitnica koje postoje u pod-stablu ukorijenjenom u tom čvoru. [1]

Pored toga, svaki podrazred čvora sintakse izlaže svu djecu snažno tipiziranim svojstvima. Na primjer, klasa čvora *BinaryExpressionSyntax* ima tri dodatna svojstva specifična za binarne operatore: *Left*, *OperatorToken* i *Right*. Vrsta *Left* i *Right* je *ExpressionSyntax*, a vrsta

OperatorToken je *SyntaxToken*. Neki čvorovi sintakse imaju neobaveznu djecu. Na primjer, *IfStatementSyntax* ima neobavezni *ElseClauseSyntax*. Ako djeteta nema, svojstvo vraća null. [1]

2.3.3.Žetoni sintakse

Žetoni sintakse su terminali jezične gramatike, koji predstavljaju najmanje sintaktičke fragmente koda. Oni nikad nisu roditelji drugih čvorova ili žetona. Žetoni sintakse sačinjeni su od ključnih riječi, identifikatora, doslovnih izraza i interpunkcije. U svrhu učinkovitosti, vrsta SyntaxToken je CLR (engl. Common Language Runtime) vrijednost. Stoga, za razliku od čvorova sintakse, postoji samo jedna struktura za sve vrste žetona s mješavinom svojstava koja imaju značenje ovisno o vrsti žetona koji se predstavlja. Na primjer, žeton doslovnog izraza tipa cjelobrojne vrijednosti predstavlja brojčanu vrijednost. Pored izvornog teksta kojeg žeton obuhvaća, žeton doslovnog izraza ima svojstvo *Value* koje govori o točno dekodiranoj vrijednosti cjelobrojnog broja. Ovo svojstvo upisuje se kao objekt, jer može biti jedno od mnogobrojnih primitivnih vrsta. *ValueText* svojstvo prikazuje iste informacije kao i svojstvo *Value*, međutim, ovo svojstvo je uvijek tipa *string*. Identifikator u izvornom tekstu C# može sadržavati znakove iz tablice Unicode koji služe za preskakanje slijeda, ali se sintaksa samog slijeda ne smatra dijelom imena identifikatora. Dakle, iako neobrađeni tekst razapet žetonom uključuje sekvencu za preskakanje slijeda, isto ne vrijedi za svojstvo *ValueText*. Umjesto toga, *ValueText* svojstvo uključuje znakove Unicodea identificirane preskočenim slijedom. [1]

2.3.4.Sitnice sintakse

Sitnice sintakse predstavljaju dijelove izvornog teksta koji su većinom nebitni za normalno razumijevanje koda. Sitnicama se smatraju razmaci, komentari i direktive pretprocesora. Zato što sitnice nisu dio normalnog jezika sintakse i mogu se pojaviti bilo gdje između bilo koja dva žetona, one nisu uključene u stablo sintakse kao dijete čvora, ali pošto su nam one važne u implementaciji značajki poput refaktoriranja i za održavanje potpune vjernosti izvornom tekstu, one postoje kao dio stabla sintakse. Sitnicama možemo pristupiti pregledom kolekcija *LeadingTrivia* ili *TrailingTrivia*, koje svaki žeton ima. Kada je raščlanjen izvorni tekst, nizovi sitnica povezani su s žetonima. Općenito, žeton posjeduje bilo koje sitnice nakon njega na istoj liniji, sve do sljedećeg žetona. Svaka sitnica nakon te linije, povezana je sa sljedećim žetonom. Prvi žeton u izvornoj datoteci dobiva sve početne sitnice, a posljednji slijed sitnica nalijepljen je na žeton na kraju datoteke, koji inače ima nultu širinu. [1]

Za razliku od čvorova sintakse i žetona, sitnice nemaju roditelja, no zato što su one dio stabla i svaka je udružena s jednim žetonom, možemo pristupiti žetonu kojem pripada koristeći se svojstvom *Token*. Kao i žetoni sintakse, sitnice imaju vrstu vrijednosti, kojoj je naziv *SyntaxTrivia* i koristi se za opis svih vrsta sitnica. [1]

2.3.5.Rasponi

Svaki čvor, žeton ili sitnica zna svoju poziciju u izvornom tekstu i broj slova koje sadrži. Položaj teksta predstavljen je kao 32-bitni cijeli broj, što je nulti indeks znaka Unicode. . Objekt *TextSpan* početna je pozicija i broj znakova, oba predstavljena kao cijeli brojevi. Ako *TextSpan* ima nultu duljinu, odnosi se na mjesto između dva znaka. Svaki čvor ima dva *TextSpan* svojstva: *Span* i *FullSpan*. Svojstvo *Span* je raspon teksta od početka prvog žetona u pod-stablu čvora do kraja posljednjeg žetona. Ovaj raspon ne uključuje nikakve vodeće ili prateće sitnice. Svojstvo FullSpan je tekstualni raspon koji uključuje normalni raspon čvora, plus raspon svih vodećih ili pratećih sitnica. [1]

Svojstvo Span je raspon teksta od početka prvog žetona u pod-stablu čvora do kraja posljednjeg žetona.

Svojstvo FullSpan je tekstualni raspon koji uključuje normalni raspon čvora, plus raspon svih vodećih ili pratećih sitnica.

Na primjer:

```
if (x > 5)
{
    ||  //ovo nevalja
    |throw new Exception("Nije u redu.");| //bolja iznimka?
}
```

Čvor unutar bloka *if* ima raspon zadan sa | znakom, što obuhvaća izraz „throw new Exception(„Not right“);“, to svojstvo je svojstvo *Span*, a puni raspon ili *FullSpan* svojstvo označeno je sa duplim || znakom i to obuhvaća raspon *Span* svojstva, uz vodeće i prateće sitnice, koje su ovdje razmaci i znakovi novog reda. [1]

2.3.6.Vrste

Svaki čvor, žeton ili sitnica ima svojstvo *RawKind* tipa *System.Int32* koje identificira točno

predstavljeni element sintakse. Svaki jezik, C# ili VB, ima jedno nabrajanje *SyntaxKind* koje navodi sve moguće čvorove, žetone i elemente sitnica u gramatici. Ova se pretvorba može izvršiti automatski pristupom metodama proširenja *CSharpSyntaxKind* () ili *VisualBasicSyntaxKind* (). Svojstvo *RawKind* omogućuje jednostavno razlučivanje tipova čvora sintakse koji dijele istu klasu čvorova. Za žetone i sitnice ovo je svojstvo jedini način razlikovanja jedne vrste elemenata od druge. [1]

Na primjer, klasa *BinaryExpressionSyntax* ima djecu *Left*, *OperatorToken* i *Right*. Svojstvo *Kind* razlikuje je li riječ o čvoru sintakse *AddExpression*, *SubtractExpression* ili *MultiplyExpression*. [1]

2.3.7. Pogreške

Čak i kad izvorni tekst sadrži pogreške u sintaksi, izloženo nam je puno zaokruženo stablo sintakse. Kada raščlanjivač naiđe na kod koji ne podliježe definiranoj sintaksi jezika, ono koristi jednu od dviju tehnika da napravi stablo sintakse: [1]

1. Ako raščlanjivač očekuje određenu vrstu žetona, ali ga ne pronađe, on će umetnuti žeton koji nedostaje u stablo sintakse na mjesto u stablu gdje je taj žeton bio očekivan. Žeton koji nedostaje predstavlja pravi žeton koji se očekivao, ali njegov raspon je prazan i svojstvo *IsMissing* vraća *true*.

2. Raščlanjivač može preskočiti žetone dok ne nađe žetone od kojih može nastaviti sa raščlanjivanjem. U tom slučaju, žetoni koji su preskočeni priloženi su kao čvor sitnice vrste *SkippedTokens*.

2.4. Rad sa semantikom

Stabla sintakse predstavljaju leksičku i sintaktičku strukturu izvornog koda. Iako te informacije same po sebi su dovoljne da opišu sve deklaracije i logiku u izvornom kodu, one nisu dovoljne za prepoznavanje onoga na što se referencira. Na primjer, mnoge vrste, polja, metode i lokalne varijable sa istim imenom mogu biti prisutni u izvornom kodu i iako svaki od njih je jedinstveno različit, određivanje točno onoga na kojega se zapravo identifikator odnosi često zahtjeva dublje razumijevanje pravila jezika. [1]

Postoje programski elementi i programi predstavljeni u izvornom kodu koji se također odnose na prethodno prevedene biblioteke, pakirane u datotečnim sklopovima. Iako nije dostupan izvorni kod za sklopove i samim tim nema čvorova ili stabala sintakse, programi se i dalje mogu odnositi na elemente unutar njih. Pored sintaktičkog modela izvornog koda, semantički model obuhvaća i jezična pravila, omogućujući vam da na taj način lakše uočite te razlike. [1]

2.4.1.Kompilacija

Kompilacija je prikaz svega potrebnog za sastavljanje programa C# ili Visual Basic, koji uključuje sve reference za sastavljanje, opcije prevoditelja i izvorne datoteke. [1]

Zbog toga što su sve informacije na jednom mjestu, elementi sadržani u izvornom kodu mogu se opisati u više detalja. Kompilacija predstavlja svaku deklariranu vrstu, član ili varijablu kao simbol. Kompilacija sadrži razne metode koje pomažu u pronalaženju i povezivanju simbola koji su deklarirani u izvornom kodu ili uvezeni kao metapodaci iz skupa. [1]

Slično stablima sintakse, kompilacije su nepromjenjive. Nakon što je kompilacija stvorena, ona se ne može mijenjati od strane korisnika ili bilo koga drugoga s kim se dijeli. Međutim, nova kompilacija se može napraviti od postojeće, sve dok se specificiraju promjene koje se rade. Na primjer, moguće je stvoriti kompilaciju koja je ista u svim pogledima, osim što ima dodatnu izvornu datoteku. [1]

2.4.2.Simboli

Simbol predstavlja jedinstveni element deklariran u izvornom kodu ili uvezen iz skupa kao metapodatak. Simbolima su predstavljeni svaki prostor imena, tip, metoda, svojstvo, polje, događaj, parametar i lokalna varijabla. [1]

Razne metode i svojstva na *Compilation* vrsti pomažu u pronalaženju simbola. Na primjer, možete pronaći simbol deklariranog tipa prema njegovom uobičajenom imenu metapodatka. Također se može pristupiti cijeloj tablici simbola kao stablu simbola ukorijenjenom na globalnom prostoru imena. [1]

Simboli također sadrže dodatne informacije koje prevoditelj određuje prema izvoru ili metapodacima, kao što su drugi referencirani simboli. Svaka vrsta simbola predstavljena je zasebnim sučeljem izvedenim iz sučelja *ISymbol*, svaka ima svoje metode i svojstva koja opisuju podatke koje je prevoditelj prikupio. Mnoga od tih svojstava direktno se referenciraju

na druge simbole. Na primjer, svojstvo *ReturnType* klase *IMethodSymbol* govori o stvarnom tipu simbola na koji se odnosi deklaracija metode. [1]

Simboli predstavljaju uobičajeni prikaz prostora imena, vrsti i članova između metapodataka i izvornog koda. Na primjer, metoda koja je deklarirana u izvornom kodu i metoda koja je uvezena iz metapodataka su predstavljene sa istim svojstvima preko *IMethodSymbol* sučelja. [1]

Simboli su u konceptu slični sustavu CLR kao što ga predstavlja *System.Reflection* API, ali su bogatiji po tome što modeliraju više nego samo tipove. Prostori imena, lokalne varijable i oznake su simboli. Pored toga, simboli su reprezentacija jezičnih koncepata, a ne koncepata CLR-a. Mnogo je preklapanja, ali u mnogo čemu se i razlikuju. Na primjer, metoda iteratora u C# ili Visual Basic je simbol. Međutim, kad se metoda iteratora prevodi u metapodatke CLR-a, to je vrsta i više metoda. [1]

2.4.3. Semantički model

Semantički model predstavlja sve semantičke informacije za pojedini izvorni dokument. Njime možemo otkriti sljedeće [1]:

- simbole na koje se referencira u izvoru na specifičnoj lokaciji
- vrstu rezultata bilo kojeg izraza
- svu dijagnostiku, odnosno pogreške i upozorenja
- tok varijabli u i izvan područja izvora
- odgovore na više spekulativna pitanja.

2.5. Rad sa radnim prostorima

Sloj radnih prostora je početna točka iz koje vršimo analizu koda i refaktoriranje preko cijelog rješenja. Unutar ovog sloja, API radnog prostora (engl. Workspace API) pomaže u organiziranju svih informacija vezanih uz projekt u rješenju u pojedinačni objektni model, pružajući izravan pristup objektnim modelima sloja prevoditelja, kao što su izvorni tekst, stablo sintakse, semantički modeli i kompilacije bez potrebe raščlanjivanja datoteka, konfiguriranja opcija ili upravljanja među-projektnim ovisnostima. [1]

Okružje domaćina (engl. Host Enviroment), poput IDE-a, pruža nam radni prostor koji odgovara otvorenom rješenju. Ovaj je model moguće koristiti i izvan IDE-a jednostavnim učitavanjem datoteke rješenja. [1]

2.5.1.Radni prostori (engl. Workspaces)

Radni prostor je aktivni prikaz rješenja kao zbirke projekata, svaka sa zbirkom dokumenata. Radni prostor jest tipično vezan za okružje domaćina koja se konstantno mijenja kako korisnik unosi ili manipulira svojstvima. [1]

Radni prostor pruža pristup do trenutnog model rješenja. Kada se dogode promjene u okružju domaćina, radni prostor pokreće odgovarajuće događaje (engl. Events) i svojstvo *CurrentSolution* se ažurira. Na primjer, kada korisnik unosi u uređivač teksta koji odgovara jednom od izvornih dokumenata, radni prostor koristi događaj koji signalizira da se cjelokupni model rješenja promijenio i pokazuje koji se točno dokument promijenio. Na te promjene možete reagirati analizirajući ispravnost novog modela, istaknuti važna područja ili podnijeti prijedloge za promjenu koda. [1]

Također možete stvoriti samostalne radne prostore koji nisu vezani za okruženje domaćina ili se koriste u aplikaciji koja nema okruženje domaćina. [1]

2.5.2.Rješenja, projekti i dokumenti

Iako se radni prostor mijenja svakim pritiskom tipke, moguće je raditi sa modelom rješenja u izolaciji. Rješenje je nepromjenjivi model projekata i dokumenata. [1]

To znači da model može biti dijeljen bez zaključavanja ili dupliciranja. Nakon što se dobije instanca rješenja, iz *CurrentSolution*, ta instanca se nikad neće promijeniti. Međutim, kao i sa stablima sintakse i kompilacijama, rješenja se mogu mijenjati izradom novih instanci temeljenih na postojećim rješenjima i specifičnim promjenama. Da bi radni prostor zrcalio promjene na rješenju, potrebno je primijeniti promijenjeno rješenje eksplicitno natrag na radni prostor. [1]

Projekt je dio cjelokupnog nepromjenjivog modela rješenja. Predstavlja sav izvorni kod dokumenata, raščlambe i kompilacijske opcije te reference za sastavljanje i reference za projekt. Iz projekta se može pristupiti odgovarajućim kompilacijama bez potrebe za određivanjem ovisnosti projekta ili raščlanjivanja bilo kojih izvornih datoteka. [1]

Dokument je također dio cjelokupnog modela nepromjenjivog rješenja. Dokument predstavlja jedinstvenu izvornu datoteku iz koje možete pristupiti tekstu datoteke, stablu sintakse i semantičkom modelu. [1]

3. Mogućnosti Roslyna

Sada kada smo se upoznali sa teorijom koja se krije iza Roslyna, vrijeme je za isprobati neke od mogućnosti Roslyna. Pošto je slovo na papiru samo po sebi nedovoljno, kroz C# i Visual Studio 2019 ćemo dalje razjasniti značajke Roslyna, i detaljnije opisati kako funkcionira i što je sve moguće sa Roslynom.

No, prije nego što krenemo, u ovom radu sam koristio Visual Studio 2019, a za korištenje Roslyna, potrebno je instalirati prevoditeljsku platformu .NET, koja nam pruža sve alate potrebne za isprobavanje Roslyna. Instaliranjem prevoditeljske platforme .NET u Visual Studio, dobijemo predloške za rad za Roslynom koji imaju dosta prožimajućeg koda koji nam nije uvijek potreban. Za sljedeće primjere koristio sam Stand Alone Code Analysis Tool koji dođe sa Rosylnom, no to su sve aplikacije konzole, tako da se može i instalirati Roslyn preko NuGet managera tako da odemo na Tools > NuGet Package Manager > Package Manager Console i zalijepimo Install-Package Microsoft.CodeAnalysis.

3.1. Analiza sintakse

API sintakse izlaže stabla sintakse koja koriste prevoditelji da bi razumjeli C# i Visual Basic programe. Proizvodi ih isti raščlanjivač koji se pokreće kad je projekt izgrađen ili kada developer stisne F5, ili preko Visual Stuidia stisnu „Start Debugging“. Stabla sintakse, kao što je prije spomenuto, imaju punu vjernost sa jezikom, svaki dio datoteke koda je predstavljen u stablu sintakse, uključujući stvari kao što su prazni prostori (razmaci, tabulatori, novi red itd.) i komentari. Stabla sintakse su nepromjenjiva, što osigurava da više developera u više dretvi može analizirati stablo, bez zaključavanja ili drugih mjera istodobnosti, uz sigurnost da neće doći do promjena podataka. Također, kao i prethodno spomenuto, stablo sintakse može se pretvoriti natrag u isti izvorni tekst iz kojega je raščlanjeno. [2]

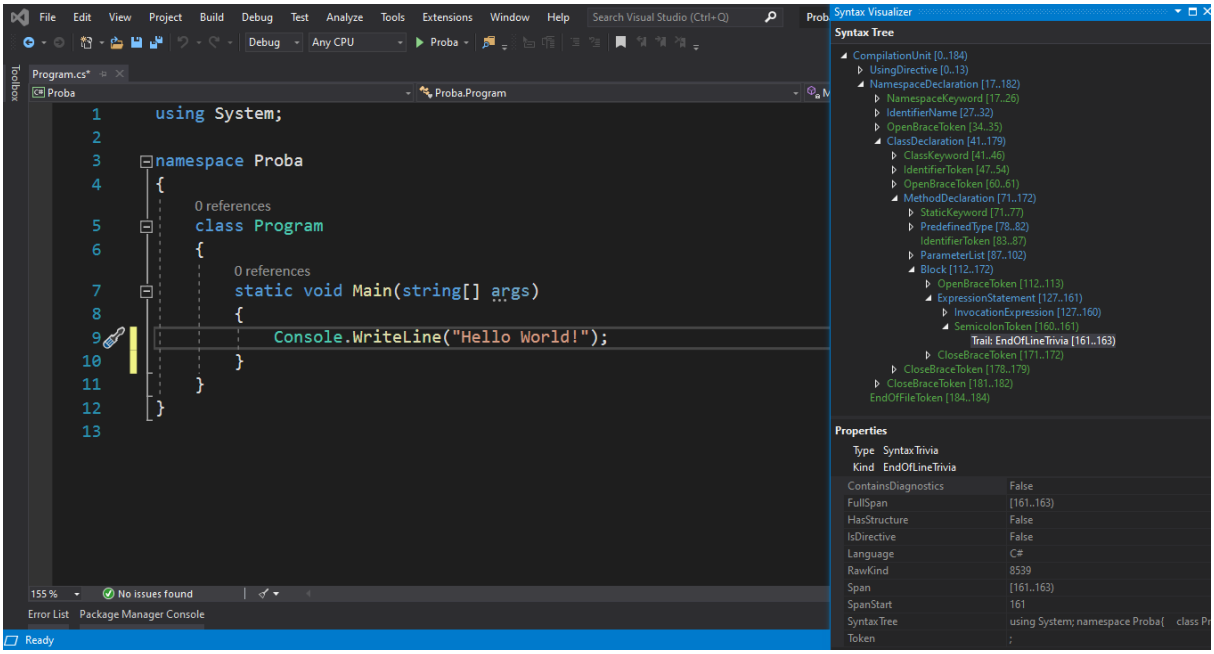
Stablo sintakse zasnivaju se na sljedećim komponentama:

1. Klasa *SyntaxTree*, čija instanca predstavlja cijelo raščlanjeno stablo sintakse. To je apstraktna klasa koja ima derivate specifične za jezik. Da bi raščlanili sintaksu u

određenom jeziku, potrebno je koristiti metode raščlanjivanja koje se nalaze u *CSharpSyntaxTree* ili *VisualBasicSyntaxTree* klasi.

2. Klasa *SyntaxNode*, čije instance predstavljaju sintaktičke konstrukcije poput deklaracija, izjava, rečenica i izraza.
3. Struktura *SyntaxToken*, koja predstavlja pojedine ključne riječi, identifikatore, operatore ili interpunkcije.
4. Struktura *SyntaxTrivia*, koja predstavlja sintaktički nebitne dijelove informacija kao što su praznine između žetona, direktive pretprocesora i komentare.

Sitnice, žetoni i čvorovi poslagani su hijerarhijski da bi stvorili stablo koje predstavlja sve što se nalazi u fragmentu C# ili Visual Basic koda. Sve to može se vidjeti sa alatom prikazivač sintakse (engl. Syntax Visualizer), koji dolazi sa prevoditeljskom platformom .NETi kojega otvaramo kroz ***View -> Other Windows -> Syntax Visualizer***.



Slika 3: Prikazivač sintakse

Čvorovi sintakse u prikazivaču sintakse su označeni plavom bojom, žetoni sintakse zelenom, a sitnice su označene crvenom bojom. Osim što nam daje lijepi pregled stabla sintakse za isječak koda, prikazivač nam pokazuje i svojstva za svaki element stabla, koja se također koriste u pisanju analizatora koda. [2]

Također, prikazivač sintakse nam može pokazati rezultat izraza, na primjer da imamo isječak: var x = 3 + 4.5;

Na stablu sintakse pojavit će se *AddExpression* čvor, koji se odnosi na dio koda 3 + 4.5, ako stisnemo desni klik, i izaberemo *View Converted Type Symbol*, u svojstvima se može vidjeti kojeg će tipa podataka biti rezultat matematičkog izraz. Ovaj alat je koristan u slučaju kada

nismo sigurni i želimo provjeriti rezultat izraza bez postavljanja točke prekida i prevođenja cijelog rješenja. Vješta navigacija kroz stablo sintakse može uštedjeti dosta vremena kod pisanja koda i pronalaženja problema u postojećem kodu, također ovo svojstvo može pomoći u pisanju analizatora koda koji bi uklanjao *var* izraze.

Sada možemo pokazati na jednom malom primjeru kako je sastavljeno stablo sintakse i kako dohvaćamo pojedine elemente sa stabla.

Sljedeći program se koristi svojstvima izvedenih iz *SyntaxNode* klasa, da bi dohvatio *args* parametar iz deklaracije *Main* metode jednostavnog *HelloWorld* programa.

```
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace PrimjeriSintaktickeAnalize
{
    class Program
    {
        static async Task Main(string[] args)
        {
            SyntaxTree stablo = CSharpSyntaxTree.ParseText(
            @"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
            int dob = 39;
            Console.Writeline(""Ja sam {0} godina star."", dob);
        }
    }
}");

            var korijen = (CompilationUnitSyntax)stablo.GetRoot();
            var prviClan = korijen.Members[0];
            var helloWorldDeklaracija = (NamespaceDeclarationSyntax)prviClan;
            var deklaracijaPrograma =
(ClassDeclarationSyntax)helloWorldDeklaracija.Members[0];
            var deklaracijaMain =
(MethodDeclarationSyntax)deklaracijaPrograma.Members[0];
            var argsParametar = deklaracijaMain.ParameterList.Parameters[0];
        }
    }
}
```

Primjer 1: Analiza sintakse HelloWorld programa

Analizator, da bi vršio analizu koda, mora imati izvor koda koji analizira. U ovom primjeru to je samo mali *HelloWorld* program. Varijabla korijen se dobije iz raščlanjenog stabla sintakse programa kojeg analiziramo, a ona je tipa *CompilationUnitSyntax* (primijetimo da na prikazivaču sintakse, kompilacijska jedinica (engl. Compilation Unit) je čvor bez roditelja u programu), te ona sadrži u sebi popis *Usings* direktiva i deklaracije prostora imena pod kolekcijom *Members*. Korijen predstavlja prvi čvor stabla sintakse, zato i naziv korijen. Korijen u ovom slučaju sadrži samo jedan *Members* element, spremimo ga u *prviClan* varijablu. Ako pregledamo varijablu *prviClan*, vidjeti ćemo da je ona tip podatka *NamespaceDeclarationSyntax*, točnije *HelloWorldNamespaceDeclaration*, pa prebacimo *prviClan* u varijablu toga tipa. U *helloWorldDeklaracija* varijabli, u *Members* kolekciji se nalaze sve deklaracije klasa koje postoje u tom prostoru imena. Uzmimo prvi i jedini član i stavimo ga u *deklaracijaPrograma* varijablu, tipa *ClassDeclaratioSyntax* pošto se ona odnosi na klasu program raščlanjenog programa. Sada u *deklaracijaPrograma* varijabli pod *Members* kolekciji su sadržane sve deklaracije metoda koje se pojavljuju u raščlanjenom programu, spremimo to u varijablu *deklaracijaMain* varijablu, tipa *MethodDeclarationSyntax*. Iz *deklaracijaMain* varijable možemo dobiti popis svih parametara u toj funkciji, pa izvadimo *argsParametar* koji je prvi i jedini član u kolekciji parametara. [2]

Sve ovo prethodno možemo također odraditi upitima, koje možemo koristiti sa LINQ-om da bi brže pronašli stvari u stablu. Ove metode definirane su na *SyntaxNode* klasi.

```
var prviParametri = from deklaracijaMain in korijen.DescendantNodes()
                                              .OfType<MethodDeclarationSyntax>()
                    where deklaracijaMain.Identifier.ValueText == "Main"
                    select deklaracijaMain.ParameterList.Parameters.First();

var argsParametar2 = prviParametri.Single();
```

A za kraj ovog primjera, dodajmo još ove linije koda:

```
            var sadrzajMain =
deklaracijaMain.DescendantNodes().OfType<LiteralExpressionSyntax>().ToList();

            foreach (var item in sadrzajMain)
            {
                Console.WriteLine(item);
            }
```

Rezultat ovog koda su svi doslovni izrazi nađeni u raščlanjenom stablu sintakse *HelloWorld* programa:

"Hello, World!"

39

"Ja sam {0} godina star."

3.2. Semantička analiza

API sintakse izlaže raščlanjivače, sama stabla sintakse i usluge za njihovo razumijevanje i izgradnju. On nam dozvoljava da gledamo strukturu programa, no često se nađemo u situaciji gdje trebamo više informacija o semantici ili značenju programa. Dok se datoteka ili isječak VB ili C# koda može sintaktički analizirati u izolaciji, nema previše smisla u vakuumu postaviti pitanja poput „Koja je vrsta ove varijable?“. Značenje vrste može ovisiti o referentnim zbirkama, uvozima iz prostora imena ili drugim datotekama koda. Tu nam ulazi klasa *Compilation*.

Kompilacija je analogna jednom projektu po viđenju prevoditelja i predstavlja sve što je potrebno za prevođenje C# ili VB programa, kao što su reference za sastavljanje, opcije prevoditelja i skup izvornih datoteka za prevođenje. S ovim kontekstom, možemo razmatrati značenje koda. Preko kompilacija možemo naći simbole, entitete kao što su tipovi, prostori imena, članovi i varijable na koja upućuju imena i drugi izrazi. Proces asociranja izraza i imena sa simbolima se zove povezivanje. [3]

Kao i klasa *SyntaxTree*, klasa *Compilation* je apstraktna klasa sa derivatima specifičnim za jezik. Kada stvaramo instancu klase *Compilation*, potrebno je prizvati tvorničku metodu *CSharpCompilation* klase. [3]

U sljedećem primjeru, vidjeti ćemo kako stvoriti kompilaciju dodavanjem referenci za sastavljanje i izvornih datoteka. Kao i sa stablima sintakse, sve u API-ju simbola i API-ju povezivanja je nepromjenjivo.

Isto kao i u prethodnom primjeru, koristiti ćemo jednostavni *HelloWorld* program za raščlambu i potrebno je dohvatiti korijen stabla sintakse. Nakon toga ćemo napraviti objekt kompilacije za *HelloWorld* program.

Kod za sljedeći primjer je preuzet sa [9].

```
using System;
using System.Linq;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SemanticQuickStart
{
    class Program
    {
        static void Main(string[] args)
        {
            SyntaxTree tree = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}");

            var root = (CompilationUnitSyntax)tree.GetRoot();
```

Primjer 2: Semantička analiza

Iz kompilacije, odnosno *CSharpCompilation* objekta izvlačimo semantički model, pa je potrebno prvo napraviti objekt kompilacije.

```
            var compilation = CSharpCompilation.Create("HelloWorld")
                .AddReferences(MetadataReference.CreateFromFile
                (typeof(object).Assembly.Location))
                .AddSyntaxTrees(tree);
```

Sada kada imam kompilaciju, možemo zatražiti i semantički model za bilo koje stablo sintakse sadržano u kompilaciji. Upiti na *SemanticModels* mogu odgovoriti na pitanja „Koja su imena u opsega na ovoj lokaciji?“, „Koji su članovi dostupni ovom metodom?“, Koje se varijable koriste u ovom bloku teksta?“ i „Na što se odnosi ovo ime/izraz?“. [3]

Semantički modeli rade sa simbolima. Programi u C# sastavljeni su od metoda, tipova, svojstava, a simboli predstavljaju sve što prevoditelj zna o svakom od tih jedinstvenih elemenata. Svaki simbol sadrži informacije o: [4]

- Tome je li element deklariran u izvoru ili metapodacima, ili je došao iz nekog vanjskog sklopa.
- Tome u kojem se prostoru imena i tipu taj simbol nalazi
- Raznim istinama o simbolu, je li *abstract*, *static*, *sealed* itd.
- Drugim informacijama sadržanim u *ISymbol* sučelju

Kada radimo sa metodama, koristimo *IMethodSymbol* sučelje, koje nam dozvoljava da odredimo: [4]

- Je li metoda skriva baznu metodu
- Simbol koji predstavlja vrstu povratka metode
- Metoda proširenja s koje je smanjen ovaj simbol

Sada kada smo to prošli, vratimo se na primjer. Potrebno je nakon vađenja objekta kompilacije uzeti semantički model, pa dodajmo sljedeće linije koda.

```
SemanticModel model = compilation.GetSemanticModel(tree);
SymbolInfo nameInfo = model.GetSymbolInfo(root.Usings[0].Name);
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Ovaj isječak koda će listati sve simbole prostora imena sadržane u *Using System* simbolu. Prvo vezujemo ime „using Sytem“ na varijablu *nameInfo*, primijetimo da ako pregledamo *nameInfo* u modu za uklanjanje pogrešaka, vidjeti ćemo da je *Symbol*.*Kind* svojstvo vrijednosti *SymbolKind*.*Namespace*. Svojstvo *Symbol* inače vraća simbol na koji se izraz odnosi, ali za izraze koji se ni na što ne odnose kao što su numerički doslovni izrazi, ono vraća vrijednost *null*. Prebacimo simbol u *NamespaceSymbol* i iterirajmo po imenima svih prostora imena u simbolu uz *GetNamespaceMembers*() funkciju. Također postoji i *GetTypeMembers*() funkcija koja nam daje sve tipove koje sadrži ovaj simbol. [3]

Rezultat:

```
Collections
Configuration
Deployment
Diagnostics
Globalization
IO
Reflection
Resources
Runtime
Security
StubHelpers
Text
Threading
```

U ovom primjeru vidjeli smo kako povezati ime da bi pronašli simbol, no postoje i drugi izrazi u C# jeziku koje možemo povezivati koji nisu imena. Sada ćemo vidjeti kako radi povezivanje kod drugih izraza, u ovom slučaju doslovnog izraza tipa *string*. [3]

Prvo moramo pronaći *string* u pitanju u stablu sintakse i spremiti ga u varijablu. Pošto je „Hello World“ *string* jedini takav u našem stablu, možemo koristiti ovaj način da ga nađemo, ali u

praksi treba biti precizniji kada tražimo podatke. Također tražimo i *TypeInfo* od pronađenog *stringa*, koji kada ga pregledamo, vidimo da mu je *Type = INamedTypeSymbol* za *System.String* tip. Možemo nastaviti pisati odakle smo stali u primjeru 2.[3]

```
var helloWorldString = root.DescendantNodes()
                .OfType<LiteralExpressionSyntax>()
                .First();

var literalInfo = model.GetTypeInfo(helloWorldString);
```

A sada dodajmo sljedeći kod da bi se enumeriralo po javnim metodama *System.String* klasa koje vraćaju *string* i ispisalo ih u konzolu.

```
Console.Clear();
foreach (var name in (from method in stringTypeSymbol.GetMembers()
                                                    .OfType<IMethodSymbol>()
                      where method.ReturnType.Equals(stringTypeSymbol) &&
                            method.DeclaredAccessibility ==
                                        Accessibility.Public
                      select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Rezultat:

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
```

3.3. Transformacija sintakse

Već smo prije mnogo puta spomenuli kako je jedan od glavnih koncepata Roslyna nepromjenjivost.

Sve transformacija na stablu sintakse se vrše tako da se stvori novo stablo sintakse, temeljeno na razlikama od postojećeg, tako da nikad ne mijenjamo postojeće stablo sintakse, već uvijek stvaramo novo stablo sintakse, koje je više ili manje različito od početnog stabla. [5]
Da bi dodavali čvorove na stablo, moramo se koristiti *SyntaxFactory* metodama. Za svaku vrstu čvora, žetona ili sitnice postoji tvornička metoda koja se može koristiti za stvaranje instance tog tipa. Sastavljanjem čvorova hijerarhijski, odozdo prema gore, možemo napraviti stablo sintakse. Na primjeru će se pokazati izrada čvorova sintakse *SyntaxNode* koriste tvorničke metode. [5]
Ali prije toga treba razjasniti *NameSyntax*. *NameSyntax* je bazna klasa četiri vrste imena koja se pojavljuju u C#-u: [5]

- *IdentifierNameSyntax* koja predstavlja jednostavna, jednodijelna imena kao što su *System* i *Microsoft*
- *GenericNameSyntax* koje predstavlja opći tip ili ime metode kao što je *List*, na primjer.
- *QualifiedNameSyntax* koje predstavlja kvalificirano ime obrasca <lijevo-ime>.<desni-identifikator-ili-opći-naziv> kao što je to npr. *Sytem.IO*
- *AliasQualifiedNameSyntax* koje predstavlja ime koristeći pseudonim vanjske biblioteke kao što je to *LibraryV2*::*Foo*. Sastavljanjem ovih imena moguće je stvoriti bilo koje ime koje se pojavljuje u jeziku C#

Kod za sljedeći primjer preuzet je sa [5].
Otvorite novi *Stand Alone Code Analysis Tool* projekt i dodajte sljedeću *Using* direktivu na vrh datoteke za uvoz tvorničkih metoda klase *SyntaxFactory* kako bismo ih kasnije mogli koristiti bez kvalificiranja:

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
```

Zatim treba deklarirati varijablu tipa *NameSyntax* koja će biti tipa *IdentifierNameSyntax* i predstavljati ime „System“, ova varijabla mora biti tipa *NameSyntax* jer će se još ponovno koristiti dalje u kodu.

```
NameSyntax ime = IdentifierName("Sytem");
```

Sada na to ime dodajemo dva *QualifiedName* imena, koja su *Collections* i *Generic*.

```
ime = QualifiedName(ime, IdentifierName("Collections"));
ime = QualifiedName(ime, IdentifierName("Generic"));
```

Nakon ove dvije operacije, ako bi koristili *Immediate Window* u Visual Studiu da upišemo *ime.ToString*(), vidjeli bismo da je ono „System.Collections.Generic“, sada možemo koristiti to ime da zamijenimo *using* direktivu u kodu.

```
            SyntaxTree stablo = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}");

            var korjen = (CompilationUnitSyntax)stablo.GetRoot();

            var stariUsing = korjen.Usings[1];
            var noviUsing = stariUsing.WithName(ime);

            korjen = korjen.ReplaceNode(stariUsing, noviUsing);
```

Primjer 3: Transformacija sintakse

I ovdje koristimo *HelloWorld* program, te na njemu mijenjamo „using System.Collections“ u „using System.Collections.Generic“, kod je sam po sebi dosta trivijalan.

4. Analizator sa ispravkom koda

Roslyn nam dozvoljava da napišemo analizatore koda i ispravke koda koji traže i ispravljaju pogreške u kodu. Te pogreške u kodu su prikazane u uređivaču kao valovite crtice pod kodom. Analizatori razumiju strukturu i sintaksu koda i traže pogreške u kodu ili neke loše navike za ispraviti. Ispravak koda pruža jedno ili više rješenja za ispravak pogreški pronađenih analizatorom koda. Praksa je da se analizator i ispravak koda upakiraju zajedno u zaseban projekt. [6]

Da bi razumjeli kod, analizatori i ispravke koda se koriste statičkom analizom. Oni služe da bi istaknuli prakse koje vode do lošeg, neodrživog koda ili do kršenja dogovorenih smjernica. Zbog toga što nam Roslyn pruža niz API-ja za razumijevanje koda, možemo se lakše fokusirati na samo pisanje ispravke koda i analizatora koda. Analizatori i ispravke koda manji su i koriste manje memorije kada su učitani u Visual Studio nego da bi pisali cijelu bazu kodova za razumijevanje projekta. [6]

Pisanje analizatora s ispravkom koda je korisno za:

1. Pružanje smjernica za kodiranje – Mnogi timovi imaju standarde za pisanje koda koji su primorani sa recenzijama koda sa drugim članovima u timu. Pregled koda se dogodi

kada developer podjeli svoj koda sa svojim timom. Developer je uložio vrijeme za razvoj neke značajke i tjedni mogu proći prije nego što dobije ikakve komentare na svoj kod. Analizatori s ispravkom koda mogu pomoći u zadavanju smjernica i primoravanju članova tima da ih se drže i da kod bude što manje „aljkav“ kod pregleda koda.

2. Pružanje smjernica za pakete biblioteke – Postoji mnoštvo različitih *NuGet* biblioteka dostupnih za .NET developere i nijedna nije ista. Analizatori s ispravkom koda mogu pomoći u određivanju pravilnog korištenja biblioteke i u pomaganju drugim korisnicima biblioteke da brže primijete pogreške u kodu.
3. Pružanje općih smjernica – Zajednica programera .NET otkrila je kroz iskustvo obrasce koji su dobri u praksi i obrasce koje je najbolje izbjegavati. Nekoliko članova zajednice stvorilo je analizatore koji primjenjuju preporučene obrasce. Kako saznajemo više, uvijek ima mjesta za nove ideje. Ovi analizatori se mogu pronaći na Visual Studio Marketplace-u.

4.1. Pisanje analizatora s ispravkom koda

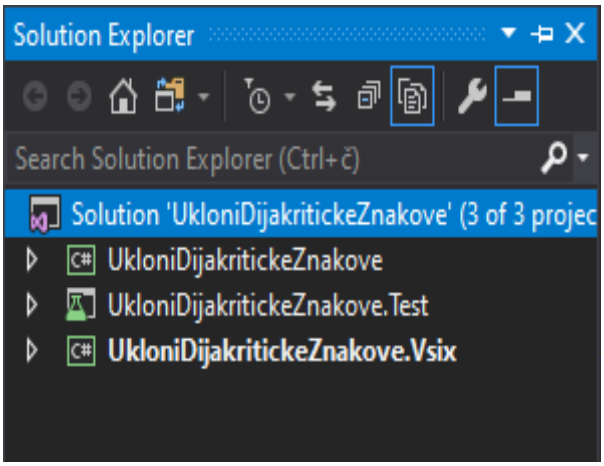
Sada kada smo ustanovili za što nam služe analizatori s ispravkom koda, bilo bi vrijeme i za pisanje jednog. U sljedećem primjeru napisati ćemo analizator s ispravkom koda koji će pronaći dijakritičke znakove u deklaracijama varijabli, metoda, klasa, enumeracija, polja, sučelja i svojstava.

Sa instalacijom prevoditeljske platforme .NET dani su i predlošci za pisanje analizatora i analizatora s ispravkom koda. U Visual Studio-u otvorimo *File -> New -> Project -> Analyzer with code fix*

Imenujmo projekt *UkloniDijakritickeZnakove*, pošto će upravo to i raditi.

Nakon što se otvori predložak u Visual Studio-u, vidjeti ćemo da su se stvorila tri projekta, a to su:

1. Prijenosna biblioteka klasa (engl. Portable class library): tu se nalazi kod za analizator i za ispravak koda te za druge pomoćne klase. Ovaj projekt je postavljen tako da stvara *NuGet* paket pri izgradnji.



2. Testni projekt (engl. Test project): projekt koji služi za testiranje analizatora i ispravke koda, korisno za izbjegavanje grešaka u projektu
3. VSIX projekt: projekt koji stvara VSIX datoteku koja nam služi za instalaciju analizatora kao proširenje Visual Studia, također koristi u otklanjanju neispravnosti na analizatoru.

Slika 4: Projekti u predlošku Analizaora s ispravkom koda

Sada otvorimo prvi projekt i *UkloniDijakritickeZnakoveAnalyzer.cs*. U njemu ima dosta koda koji nam ni ne treba, već je rađen samo kao predložak za kako koristiti analizator i koji nam neće trebati u ovom primjeru, pa nakon što izbrišemo nepotrebne isječke, kod bi trebao izgledati ovako:

```
namespace UkloniDijakritickeZnakove

{

    [DiagnosticAnalyzer(LanguageNames.CSharp)]

    public class UkloniDijakritickeZnakoveAnalyzer : DiagnosticAnalyzer

    {

        public const string DiagnosticId = "UkloniDijakritickeZnakove";

        private static readonly LocalizableString Title = new
LocalizableResourceString(nameof(Resources.AnalyzerTitle),
Resources.ResourceManager, typeof(Resources));

        private static readonly LocalizableString MessageFormat = new
LocalizableResourceString(nameof(Resources.AnalyzerMessageFormat),
Resources.ResourceManager, typeof(Resources));

        private static readonly LocalizableString Description = new
LocalizableResourceString(nameof(Resources.AnalyzerDescription),
Resources.ResourceManager, typeof(Resources));

        private const string Category = "Imenovanje";

        private static DiagnosticDescriptor Rule = new
DiagnosticDescriptor(DiagnosticId, Title, MessageFormat, Category,
DiagnosticSeverity.Warning, isEnabledByDefault: true, description:
Description);

        public override ImmutableArray<DiagnosticDescriptor>
SupportedDiagnostics { get { return ImmutableArray.Create(Rule); } }


        public override void Initialize(AnalysisContext context)

        {

        }

    }

}
```

Primjer 4: Sadržaj analizatora prije pisanja koda

Prije nego što se krene sa pisanjem analizatora, preimenujmo *Category* u „Imenovanje“, i u *Resources.resx* datoteci promijenimo opise za *Title*, *MessageFormat* i *Description*. Trebalo bi ovako izgledati:

	Name	Value	Comment
	AnalyzerDescription	Trebalo bi ukloniti dijakritiku	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Tip imena: '{0}' sadrzi dijakritiku	The format-able message the diagnostic displays.
	AnalyzerTitle	Sadrzi dijakritike	The title of the diagnostic.
*			

Slika 5: Sadržaj datoteke Resources.resx

Sada vratimo se na analizator i na funkciju *Initialize*. Vidimo da funkcija ima ulaz *context* koji je tipa *AnalysisContext*. Preko njega možemo registrirati delegat kojim ćemo vršiti analizu

koda. Tako da prvo što trebamo odrediti je što ćemo točno gledati u kodu da analiziramo. Pa dodajmo u kod sljedeći isječak:

```
context.RegisterSyntaxNodeAction(c =>
{

}, SyntaxKind.ClassDeclaration, SyntaxKind.InterfaceDeclaration,
                                SyntaxKind.VariableDeclarator,
                SyntaxKind.EnumDeclaration, SyntaxKind.FieldDeclaration,
                SyntaxKind.MethodDeclaration
    , SyntaxKind.PropertyDeclaration);
```

Za *context* koji promatramo, rekli smo mu da gledamo samo radnje koje se odnose na čvorove sintakse koje dalje točnije određujemo sa *SyntaxKind* tipom podataka, koji govori analizatoru da treba provjeravati kod za pogreške u deklaracijama sučelja, varijabli, enumeracija, polja, metoda i svojstava.

Prebacimo se sada u funkciju delegata c => i dodajmo sljedeći kod unutar zagrada:

```
        var tokenIdentifikator = c.Node.DescendantTokens().FirstOrDefault(x =>
x.IsKind(SyntaxKind.IdentifierToken));
        if (tokenIdentifikator == null) return;
        if (tokenIdentifikator.Text.SadrziDijakritiku()) return;
        var diag = Diagnostic.Create(Rule, tokenIdentifikator.GetLocation(),
tokenIdentifikator.Text);
        c.ReportDiagnostic(diag);
```

U čvorovima sintakse se nalaze žetoni. Od tih žetona ima jedan koji nam daje ime čvora, u ovom slučaju tražimo ime bilo čega što se deklarira u kodu, a da je metoda, varijabla, enumeracija itd. kako je navedeno u drugom argumentu metode *RegisterSyntaxNodeAction* sa *SyntaxKind* tipovima podataka. Žeton koji nam daje ime deklariranog čvora jest žeton identifikator (engl. Identifier Token). Njega vadimo iz danog čvora sintakse. Ne prijavljuje se greška, ako žeton ne postoji ili ne sadrži dijakritiku. Ako pak se nađe dijakritika u žetonu, onda se prijavljuje greška i dižemo dijagnostiku.

Također napravimo i novu klasu *Pomagalo.cs* koja će nam koristiti za pronalaženje i uklanjanje dijakritičkih znakova. Ona će imati dvije funkcije: *SadrziDijakritiku*() koja će provjeriti da li je dani *string* normaliziran po *FormD* formi ili ne, i *UkloniDijakritiku*() koja će normalizirati dani *string* u *FormD*.

```
static class Pomagalo
{
    internal static string UkloniDijakritike(this string stIn)
    {
        string stFormD = stIn.Normalize(NormalizationForm.FormD);
        StringBuilder sb = new StringBuilder();

        for (int ich = 0; ich < stFormD.Length; ich++)
        {
            UnicodeCategory uc = CharUnicodeInfo.GetUnicodeCategory(stFormD[ich]);
            if (uc != UnicodeCategory.NonSpacingMark)
            {
                sb.Append(stFormD[ich]);
            }
        }

        return sb.ToString().Normalize(NormalizationForm.FormC);
    }
    internal static bool SadrziDijakritiku(this string stIn)
    {
        return stIn.IsNormalized(NormalizationForm.FormD);
    }
}
```

Primjer 5: Sadržaj klase Pomagalo.cs

Sada možemo prijeći na ispravak koda, koji također ima dosta nepotrebnog koda iz predloška, pa i njega treba ukloniti. Funkcija pri dnu klase nije potrebna, a u funkciji *RegisterCodeFixesAsync* ostaviti ćemo sljedeće linije koda koje su korisne i skoro za sve ispravke koda se koriste. Također *string Title* preimenujemo u „Ukloni dijakritiku“.

```
        public sealed override async Task RegisterCodeFixesAsync(CodeFixContext
context)
        {
            var root = await
context.Document.GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
            var diagnostic = context.Diagnostics.First();
            var diagnosticSpan = diagnostic.Location.SourceSpan;

         }
```

Primjer 6: Početni sadržaj ispravka koda

Metoda ima jedan ulaz, *context*, koji sadrži informacije koje se također nalaze i u analizatoru na razini dijagnostike. Ovaj kod nam praktički govori koji dokument javlja problem i lokaciju problema, odnosno dijagnostike. Iz ovoga možemo početi raditi ispravak koda za korisnika.

```
            var node = root.FindNode(diagnosticSpan);
```

U varijablu *node*, iz korijena stabla sintakse, *root* varijable, vadimo čvor koji je uzrokovao dijagnostiku, *diagnosticSpan* je mjesto u kojem se pojavila dijagnostika.

```
context.RegisterCodeFix(
     CodeAction.Create(
         title: title,
         createChangedSolution: async c =>
         {
         }
         ,equivalenceKey: title),
     diagnostic);
```

Primjer 7: Registriranje ispravka koda

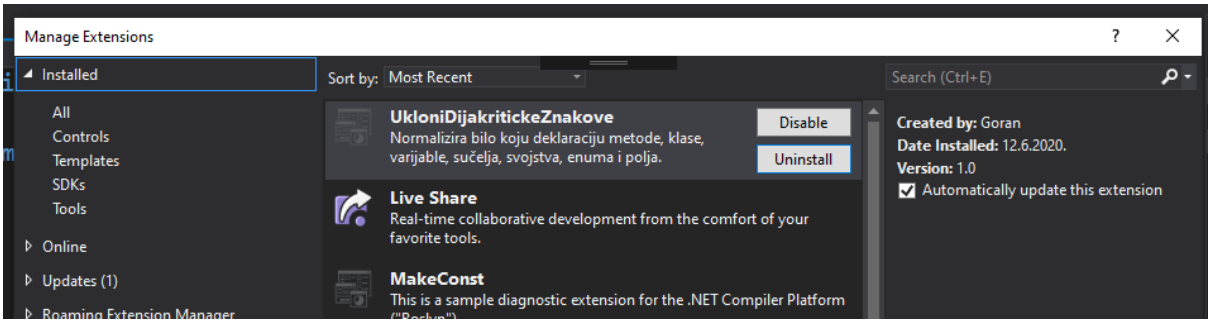
Isječak u primjeru 7 nam služi za registriranje ispravka koda. U zagrade treba napisati točno što će se napraviti sa kodom, točnije kako će ispravak izgledati, pa napišimo sljedeće:

```
var semanticModel = await context.Document.GetSemanticModelAsync();
var simbol = semanticModel.GetDeclaredSymbol(node);
var novoIme = simbol.Name.UkloniDijakritike();
var novoRjesenje = await Renamer.RenameSymbolAsync(context.Document.Project.Solution,
simbol, novoIme, null);
return novoRjesenje;
```

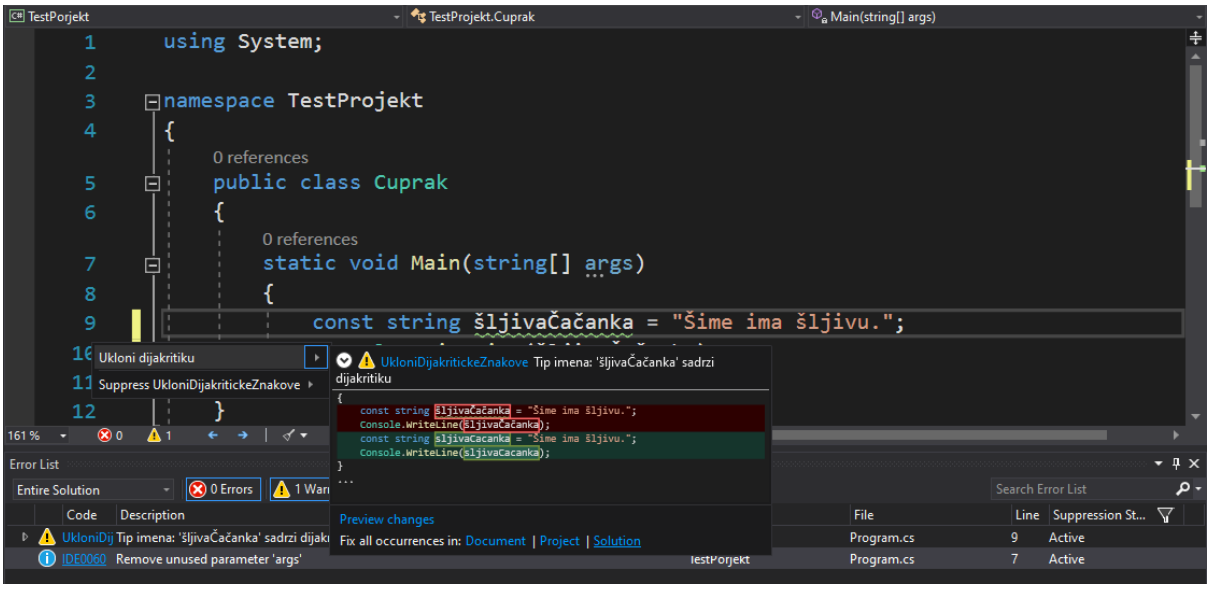
Primjer 8: Ispravak koda i vraćanje novog rješenja

Potrebno je prvo dohvatiti semantički model iz dokumenta. Potrebno je također pronaći simbol koji odgovara čvoru sintakse nad kojim vršimo ispravak koda. Iz simbola možemo izvršiti funkciju iz klase Pomagalo koja normalizira *string* tipove podataka. I to je to što se tiče rješenja, sve što je preostalo je da se vrati novo rješenje ili novi dokument, pa zato pomoću *Renamer* klase možemo vratiti rješenje koje mijenja ime na simbolu čvora koji sadržava dijakritičke znakove.

Sada možemo isprobati da li analizator i ispravak koda rade kako smo i planirali. Izgradnjom i pokretanjem koda otvara se nova instanca Visual Studia koja pod proširenja ima instaliran analizator koji smo napisali.



Slika 6: Proširenja pri pokretanju analizatora



Slika 7: Analizator predlaže ispravak koda.

Napravio sam jedan testni projekt u kojem sam deklarirao varijablu u kojoj ima dijakritičkih znakova. Kao što se vidi na slici iznad, analizator je pronašao pogrešku u kodu, tj. nije pogreška već je postavljeno kao upozorenje, i predlaže ispravak koda. Također, ako se ta pogreška pojavljuje u drugim mjestima u rješenju, možemo ih ispraviti sve jednim klikom miša, odnosno pritiskom na „Fix all occurences in: Solution“. U našem jeziku dijakritički znakovi su, „Č, Ć, Š, č, ć, š“. Jedino „ž, đ“ se smatraju posebnim znakovima koji ne spadaju u kategoriju dijakritika, pa se oni ne normaliziraju, no ako je potrebno, uvijek se može napraviti dodatak na klasi *Pomagalo* koji traži baš te znakove.

I naposljetku, pogledajmo testni projekt i VSIX projekt.

U testnom projektu imaju dvije metode koje služe kao primjer za testiranje projekta koji je bio u predlošku. Prva ispituje sa praznim kodom, kod kojeg nema prevođenja, nema ničega osim praznog *stringa*, a druga ispituje sa kodom na kojemu treba naći grešku i dati ispravak koda. Prema njima sam napisao test i za ovaj analizator. Kod je malo opširan pa u svrhu uštede prostora, ukratko ću ga opisati.

Prvi test:

```
var test = @"";
VerifyCSharpDiagnostic(test);
```

Primjer 9: Testni projekt, prvi test.

Prazni test na kojem se ne javlja nikakva dijagnostika.

U drugom testu, u testu u kojem se treba pojaviti dijagnostika i ispravak koda, potrebno je postaviti program koji se testira.

```
            var test = @"
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using System.Diagnostics;

    namespace ConsoleApplication1
    {
        class TypeName
        {
            int x = 3;
            var sšestiČlan = 6;
        }
    }";
```

Primjer 10: Postavljanje drugog testa za projekt

Zatim je potrebno napisati na kojem mjestu u kodu i kakve jačine treba biti (pogreška, upozorenje, informacija, skriveno). Jedini problem sa ovime je da treba za lokaciju pogreške u kodu pružiti broj linije koda uz broj stupca u kodu, koji nije baš vidljiv u Visual Studio-u. *VerifyCSharpDiagnostic* provjerava da li se pogreška nalazi na mjestu na kojemu smo pokazali i da li je to ta pogreška koju tražimo.

```
            var ocekivano = new DiagnosticResult
            {
                Id = "UkloniDijakritickeZnakove",
                Message = String.Format("Tip imena: '{0}' sadrzi dijakritiku",
"sšestiČlan"),
                Severity = DiagnosticSeverity.Warning,
                Locations =
                    new[] {
                            new DiagnosticResultLocation("Test0.cs", 14, 17)
                        }
            };

            VerifyCSharpDiagnostic(test, ocekivano);
```

Primjer 11: Postavljanje očekivane dijagnostike

Nakon što je postavljena očekivana dijagnostika, može se pružiti očekivani kod nakon ispravka pogreške i provjeru da li je ispravak uspješan.

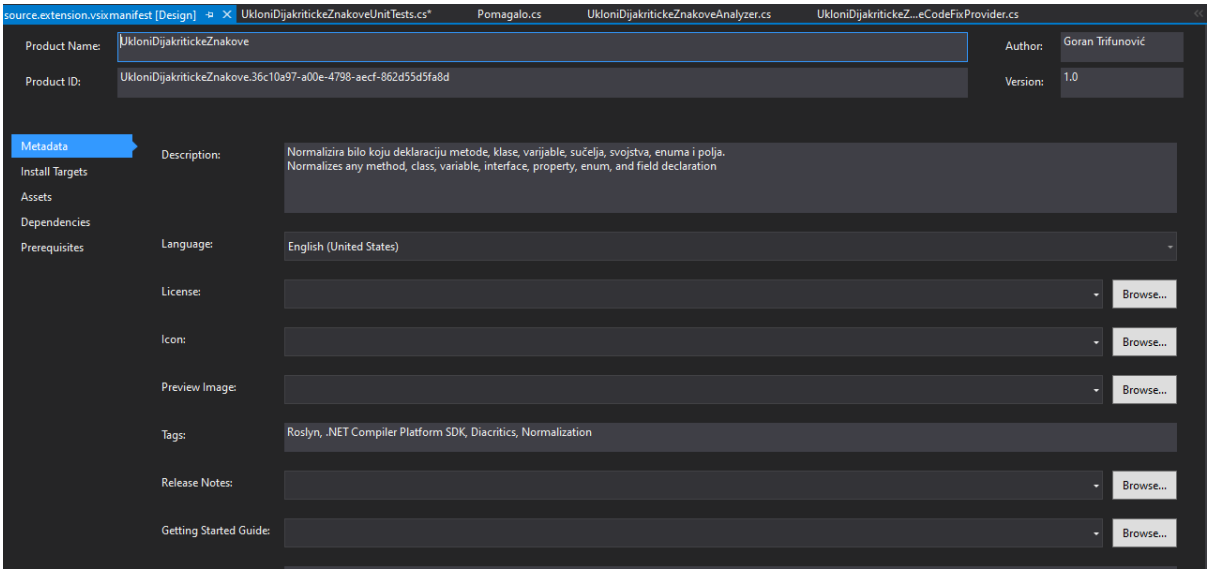
```
        var fixtest = @"
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class TypeName
    {
        int x = 3;
        var ssestiClan = 6;
    }
}";
        VerifyCSharpFix(test, fixtest);
```

Primjer 12: Očekivani ispravak pogreške

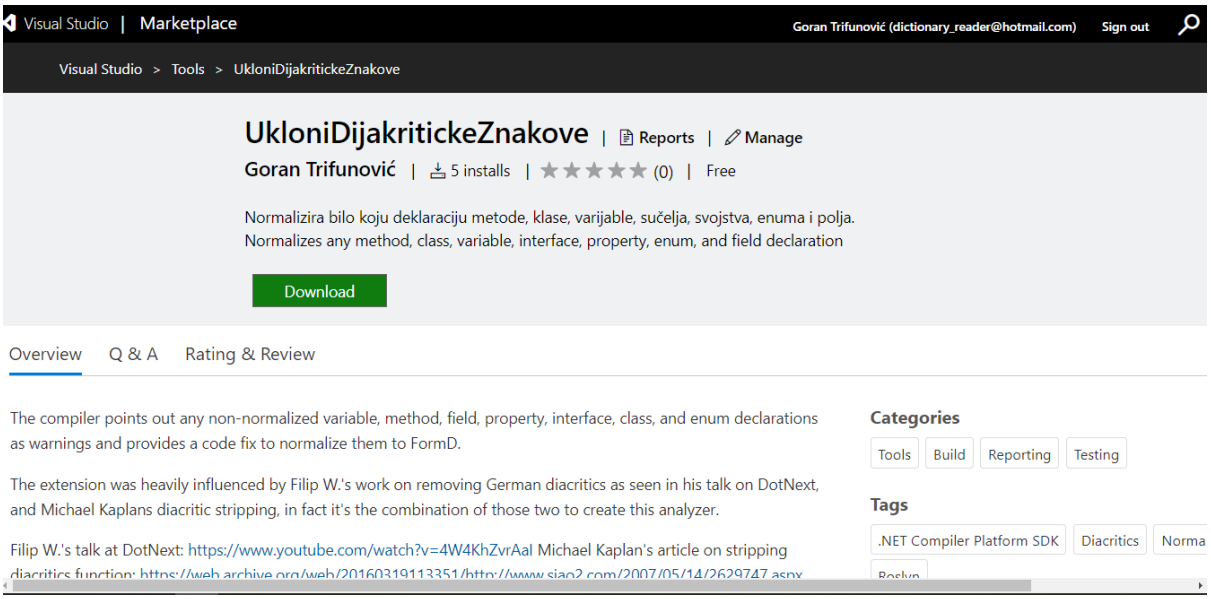
Kao i prethodno, imamo *string* sa očekivanim izgledom koda nakon popravka, i metodu koja provjerava da li je ispravak uspješan.

VSIX projekt dozvoljava nam da ovaj analizator sa ispravkom koda objavimo online preko Visual Studio Marketplacea. Otvorimo manifest i ispunimo podatke.



Slika 8: Manifest VSIX projekta

Također pri postavljanju VSIX manifesta, bitno je provjeriti „Prerequisites“, jer su tamo postavljeni „.NET Core“ i „.NET Compiler Platform SDK“ kao preduvjeti za instalaciju proširenja. Te preduvjete možemo slobodno promijeniti. Nakon ispunjenja manifesta, možemo ga objaviti na Visual Studio Marketplace.



Slika 9: Analizator s ispravkom koda objavljen na Visual Studio Marketplaceu

Napisali smo analizator s ispravkom koda koji provjerava dijakritičke znakove u deklaracijama i objavili ga na VSM-u.

5. Zaključak

Roslyn predstavlja niz API-ja koji nam mogu mnogo pomoći u svakodnevnom programiranju. Prava moć Roslyna je u tome što je nešto što je inače vrlo komplicirano, pojednostavljeno sa specijaliziranim API-jima. Sami procesi prevoditelja, raščlambe koda i dohvaćanje pojedinih čvorova sintakse u svrhu analize i ispravljanja koda pojednostavljeni su i može se reći i trivijalizirani.

U ovom radu prošli smo kroz teoretsku osnovu koja čini sam temelj Roslyna, a uz pomoć koje nam je lakše predočiti kako Roslyn u svojoj implementaciji funkcionira. Nakon teorije, prošli smo kroz neke demonstracijske primjere gdje smo pokazali kako izvući stablo sintakse i semantički model te što dalje raditi s njima. Na posljetku, obradili smo i glavnu značajku Roslyna i najmoćniji alat koji ima, a to je analizator s ispravljanjem koda. Primjer u ovom radu vrlo je trivijalan, ali svrsishodno demonstrira koliko je zapravo s Roslynom jednostavno baratati.

Ovo je tema koja možda u određenim krugovima i nije tako popularna i za nju sam čuo tek kod traženja teme za završni rad. A što nije ni čudno, gledajući kako je Microsoft sam Roslyn izdao kao projekt otvorenog koda, 2014. godine, a tada još nisu niti bile gotove sve značajke Roslyna – već su se u to vrijeme spremali za C# 6, a kasnije opet za C# 7.

Ovo je tema koja jako dobro može služiti u obrazovanju, a imajući na umu kako nam sam Roslyn daje mnogo jednostavniju sliku prevoditelja i cijelog procesa prevođenja koda u programskim jezicima C# i Visual Basic.

Popis literature

[1] „Roslyn Overview“ na Github. Dostupno: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview#wiki-pages-box [Pristupano: 03.06.2020.]

[2] Anthony D. Green, „Getting Started C# Syntax Analysis“ na Github. Dostupno: https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Syntax-Analysis [Pristupano: 04.06.2020.]

[3] Bill Wagner, „Getting Started C# Semantic Analysis“ na Github. Dostupno: https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Semantic-Analysis [Pristupano: 04.06.2020.]

[4] Josh Varty, „Learn Roslyn Now: Part 7 Introducing the Semantic Model“, 2014. [Na Internetu]. Dostupno: https://joshvarty.com/2014/10/30/learn-roslyn-now-part-7-introducing-the-semantic-model/ [Pristupano: 05.06.2020.]

[5] Anthony D. Green, „Getting Started C# Syntax Transformation“ na Github. Dostupno: https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Syntax-Transformation [Pristupano: 04.06.2020.]

[6] Microsoft, „The .NET Compiler Platform SDK“, 2017. [Na Internetu]. Dostupno: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/ [Pristupano: 14.06.2020.]

[7] Compiler pipeline [Slika] (bez. dat.) Preuzeto 03.06.2020. sa https://raw.githubusercontent.com/wiki/dotnet/roslyn/images/compiler-pipeline.png

[8] Compiler pipeline API [Slika] (bez. dat.) Preuzeto 03.06.2020. sa https://raw.githubusercontent.com/wiki/dotnet/roslyn/images/compiler-pipeline-api.png

[9] Bill Wagner (2018), „Getting Started C# Semantic Analysis“ [Izvorni kod] Preuzeto sa: https://github.com/dotnet/docs/pull/4331/commits/577b4404e1c547b3f4b9d00ef4bfe2e7cd087161

Popis slika

Popis primjera