C# AST fix

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Ana Petrović, 1073/2020, pana.petrovic@gmail.com,
Aleksandra Dotlić, 1077/2020, alexandra.nerandzic@gmail.com,
Branko Đaković, 1083/2019, brankodjakovic08@gmail.com,
Jovana Pejkić, 1089/2020, jovanadpejkic@gmail.com

13. septembar 2021

Sažetak

Tokom razvoja softvera neretko dolazi do propusta u arhitekturi, pisanju koda, testiranju, dokumentovanju i ostalim fazama razvoja. Upravo zbog toga treba voditi računa o svim fazama i posebno se posvetiti svakoj od njih. Da bi kod bio kvalitetniji, greške i propuste treba uočiti na vreme i ispraviti. Od velikog značaja za kvalitet softvera jeste analiza izvornog koda i zamena nepodobnih sintaksnih formi odgovarajućim, čime se ovaj rad bavi, a uz pomoć čega se može postići značajna memorijska i vremenska ušteda, kao i razni drugi benefiti.

Sadržaj

1 Opis problema

Zadatak koji je rešavan u okviru ovog projekta je analiza koda u jeziku *C#*, detektovanje i popravka neispravnih ili nedovoljno dobrih sintaksnih konstrukata. Nakon konstantovanja problema, korisniku neće biti (samo) prijavljeno upozorenje i sugestija kako se kod može popraviti, već će se vršiti uklanjanje ili zamena nepoželjnih konstrukata onima koji su poželjniji. Uz sve ovo, funkcionalnost koda ne sme biti promenjena.

Cilj ovog rada je kreiranje kvalitetnijeg koda, bez suvišnih delova, kompleksnih fragmenata i nečitljive sintakse. To je način da se izvorni kod "očisti"i tako smanji mogućnost za pojavu grešaka.

U nastavku, u okviru ove sekcije je detaljnije opisano **refaktorisanje** - proces koji predstavlja navedene transformacije (podsekcija 1.1). Dalje, u podsekciji 1.2 su opisane tehnologije koje su korišćene.

1.1 Refaktorisanje

Refaktorisanje je proces (postupak) kojim se postojeći kod menja tako da bude čitljiviji i manje kompleksan. Ključna stvar je da izmene koda ne utiču na funkcionalnost samog koda. Cilj je poboljšanje dizajna, strukture i/ili implementacije softvera, uz očuvanje njegove funkcionalnosti. Ovo može poboljšati održavanje izvornog koda i stvoriti jednostavniju, čistiju i izražajniju unutrašnju arhitekturu. Osim toga, proces refaktorisanja može uticati i na poboljšanje performansi.

1.2 Korišćene tehnologije

Sve transformacije su pisane u jeziku *C#* uz korišćenje .NET platforme za kompajliranje (eng. *.NET Compiler Platform*) pod nazivom **Rozlin** (eng. *Roslyn*), opisane u podsekciji 1.2.1. Za testiranje rezultata ovih transformacija pisani su testovi koji predstavljaju jednostavne programe, takođe pisane u jeziku *C#*. Projekat je rađen u okruženju *Visual Studio*.

1.2.1 Rozlin

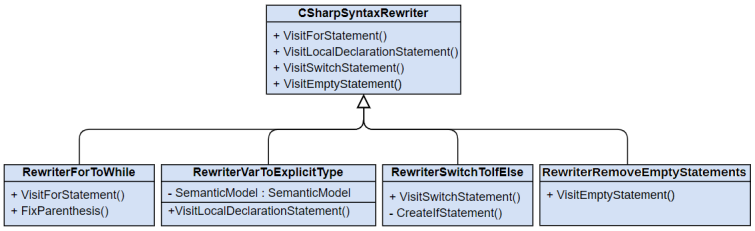
Rozlin je platforma koju čini skup kompajlera otvorenog koda (eng. *open-source compilers*) i interfejsa za analizu koda (eng. *code analysis APIs*) za *C#* i *Visual Basic* (*VB.NET*) Majkrosoftove jezike. Rozlin je dizajniran tako da olakša razvoj alata za analizu izvornog koda. Ova .NET platforma ima tri vrste APIja: *feature APIs*, *work-space APIs* i *compiler APIs*, koji su detaljnije opisani u tabeli 1.2.1.

Feature APIs	dozvoljavaju *source code tool* programerima da refaktorišu i isprave kod
Work-space APIs	dozvoljavaju korišćenje plugina za, na primer, pronalaženje referenci promenljive ili oblikovanja koda (eng. *code formatting*)
Compiler APIs	dopuštaju još elegantniju analizu izvornog koda, izlaganjem (odnosno upućivanjem) direktnih poziva za izvođenje sintaksnog stabla i analizom toka vezivanja (eng. *binding flow analysis*)

2 Opis arhitekture sistema

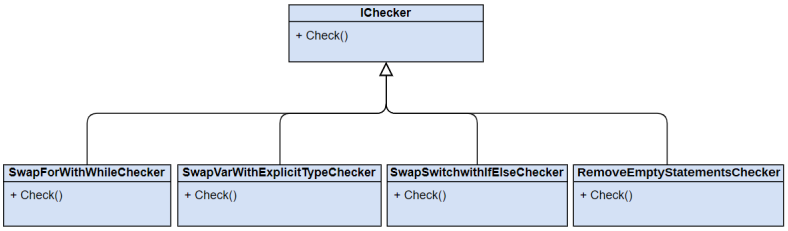
U ovoj sekciji je opisana struktura programa. Naime, sam program se sastoji iz glavnog *.cs* fajla - *Program.cs*, interfejsa *IChecker.cs* i osam klasa - četiri *čekera* i četiri *prepisivača*. U nastavku su dati odgovarajući grafički prikazi koji ovo vizualizuju.

Na slici 1 je prikazana hijerarhija klasa koje su korišćene za izvršavanje transformacija. Ono što je za sve implementirane klase zajedničko, jeste da sve one nasleđuju klasu *CSharpSyntaxRewriter* i implementiraju metode u skladu sa transformacijom koju vrše.



Slika 1: Nasleđivanje klase *CSharpSyntaxRewriter*

Na slici 2 su prikazani različiti čekeri. Ove klase naseđuju *IChecker* klasu i implementiraju metod *Check*.



Slika 2: Nasleđivanje klase *IChecker*

3 Opis rešenja problema

Osnovna ideja je da se u *.cs* fajlu, čija se putanja dobija od strane korisnika, izvrši pronalaženje i zamena nepodobnih programskih konstrukata odgovarajućim (bez promene semantike koda) i tako izmenjen kod upiše u *.cs* fajl, čiju putanju, opet, zadaje korisnik. Stoga, rad programa je zasnovan na analizi izvornog koda i adekvatnim transformacijama. Preciznije, najpre se određeni konstrukti pretražuju u kodu a zatim i zamenjuju onima koji su poželjniji. Primer transformacije može biti zamena *for* petlje u *while* petlju, zamena ključne reči *var* u eksplicitni tip promenljive i slično. Namera je da se korisniku ukaže na nepravilne, nečitljive konstrukte, ali i da se prikaže bolje rešenje za konkretan slučaj. Ideja je da ove transformacije vrše različiti "prepisivači"(eng. *Rewriters*) - klase u kojima se redefinišu metode nadklase *CSharpSyntaxRewriter*. U projektu su implementirani sledeći "prepisivači":

- RewriterForToWhile - zamenjuje for petlju u while
- RewriterVarToExplicitType - zamenjuje ključnu rec var sa eksplicitnim tipom
- RewriterSwitchToIfElse - zamenjuje switch naredbu if-else naredbom
- RewriterRemoveEmptyStatements - eliminiše prazne naredbe

Sam program se sastoji od nekoliko čekera (eng. *Chekers*) i "prepisivača", gde čekeri pozivaju metod *Visit* nad instancom neke od *Rewriter* klasa. Klase *SwapForWithWhileChecker* i *RewriterForToWhile* su opisane u podsekciji 3.1, klase *SwapVarWithExplicitTypeChecker* i *RewriterVarToExplicitType* u podsekciji 3.2, klase *SwapSwitchWithIfElseChecker* i *RewriterSwitchToIfElse* u podsekciji 3.3, a klase *RemoveEmptyStatementsChecker* i *RewriterRemoveEmptyStatements* u podsekciji 3.4. Kod je organizovan u nabrojanim klasama kako bi bio čitljiviji i tako da svaka klasa predstavlja jednu celinu.

Za kreiranje i transformaciju stabla Rozlin nudi dve mogućnosti: *Factory* metodi su najbolji izbor kada je potrebno zameniti specifičan čvor, ili ukoliko postoji specifična lokacija na kojoj je potrebno dodati nov čvor. *Rewriters* su najbolja opcija kada treba skenirati ceo projekat kako bi u kodu bili pronađeni (prepoznati) šabloni (eng. *code patterns*) koje je potrebno zameniti.

U nastavku ove sekcije je prikazan osnovni algoritam, tako što je detaljno opisana svaka od implementiranih klasa. U podsekciji 3.1 je opisana zamena *for* petlje u *while* petlju, u podsekciji 3.2 je opisana zamena ključne reči *var* u eksplicitni tip promenljive, zatim je u podsekciji 3.3 opisana zamena naredbe *switch* u *if-else* naredbu, i na kraju u podsekciji 3.4 je opisana eliminacija praznih naredbi.

3.1 Zamena for petlje u while petlju

Iako se i *for* i *while* petljom postižu slični rezultati, odluka o tome koja će biti iskorišćena u mnogim slučajevima zavisi od prioriteta samog programera. Ipak, demonstracije radi, u ovoj sekciji će biti prikazana zamena *for* petlje *while* petljom. Ukoliko se u datom *.cs* fajlu naiđe na konstrukciju oblika kao u kodu 1, to će biti zamenjeno *while* petljom kao što je prikazano u kodu 2.

```
1000    for (int i = 0; i < 5; i++) {
        ...
1002    }
```

Listing 1: *For* petlja

```
1000    int i = 0;

1002    while (i < 5) {
        ...
1004    i++;
        }
1006
```

Listing 2: *While* petlja

Kao i za svaku transformaciju, i ovde se koriste klase *Checker* i *Rewriter*. Klasa *SwapForWithWhileChecker* je prikazana na slici 3. Ova klasa nasleđuje klasu *IChecker* i u njoj je implementiran metod *Check* koji kao argumente prima sintaksno stablo i semantički model. Unutar ovog metoda kreira se instanca klase *RewriterForToWhile* nad kojom se poziva metod *Visit*, koji vrši obilazak stabla krenuvši od korena koji je prosleđen kao argument.

Dodatno, na početku svakog čekera, navodi se opcija za argument komandne linije koju je potrebno uključiti za konkretnu transformaciju. U ovom slučaju to je "*-forToWhile*".

```
namespace ConsoleApp.Checkers
{
    [CommandLineArgument("-forToWhile")]
    0 references
    public class SwapForWithWhileChecker : IChecker
    {
        2 references
        public SyntaxNode Check(SyntaxTree tree, SemanticModel semanticModel)
        {
            var rewritten = new RewriterForToWhile().Visit(tree.GetRoot());
            return rewritten;
        }
    }
}
```

Slika 3: Klasa *SwapForWithWhileChecker*

Dalje, klasa *RewriterForToWhile* (prikazana na slici 4) nasleđuje klasu *CSharpSyntaxRewriter* i redefiniše metod *VisitForStatement*. Ovaj metod kao argument prima čvor i, krenuvši od njega, posećuje svaku *for* naredbu. Kada naiđe na takvu naredbu, proverava da li u njoj postoji deklaracija (na primer *int i=0*) i, ukoliko postoji, ona biva sačuvana u promenljivu *declarationNode* kako bi kasnije bila postavljena iznad *while* petlje. Samo telo petlje se nalazi unutar *syntaxNode.Statement*. Na osnovu ovoga, može se napraviti novi čvor koji predstavlja *while* petlju. Najpre se proverava da li je for petlja imala neki uslov i, ukoliko jeste, onda se on postavlja za uslov *while* petlje, a ukoliko nije, onda se za uslov *while* petlje postavlja *true*. Zatim se parsira telo *for* petlje u *while* (dodaje se inkrementiranje i slično) i postavljaju se *;* tamo gde je to potrebno. Na kraju je napravljen novi čvor koji se sastoji od pomenute deklaracije promenjive i *while* petlje - to je 'spojeno' i vraćeno kao jedan čvor. Implementirana je i pomoćna funkcija *FixParenthesis* koja 'popravlja' zagrade pre kreiranja novog čvora.

Slika 4: Klasa *RewriterForToWhile*

3.2 Zamena ključne reči var u eksplicitni tip

Pri lokalnoj deklaraciji promenljivih u jeziku C#, promenljivoj se može dodeliti implicitni tip navođenjem ključne reči *var*. U nekim slučajevima, takav kod je preporučljivo refaktorisati promenom deklaracije takvih promenljivih u eksplicitan tip - npr. kada nije neophodno inicijalizovati promenljivu u samoj deklaraciji, ili jednostavno da bi se dobio čitljiviji kod. U nastavku (kod 3 i 4) su prikazani oblici sintaksnog konstrukta pre i nakon zamene.

```
1000    var i = 5;
        if(i<10)
1002    var a = i + 1.3;
```

Listing 3: Var promenljiva

```
1000    int i = 5;
        if(i<10)
1002    double a = i + 1.3;
```

Listing 4: Eksplicitan tip

Slično kao u prethodnom primeru, postoji klasa koja implementira čeker - *SwapVarWithExplicitTypeChecker*, i klasa koja implementira metod koji posećuje čvorove - *RewriterVarToExplicitType* (Slika 5). U ovoj transformaciji, posećuju se čvorovi koji su tipa *LocalDeclarationStatement*, odnosno deklaracija promenljive. Kako bi se dobila informacija o tipu čvora, kod je neophodno analizirati na semantičkom nivou, pa je potrebno uključiti i semantički model. Metod *VisitLocalDeclarationStatement* najpre iz semantičkog modela pokupi tip implicitne naredbe. Sa dobijenim tipom i ostatkom naredbe koji ostaje isti kao i ranije kreira se nova *LocalDeclarationStatement*, pritom zadržavajući formatiranje koje je korišćeno. Na kraju, vratća se kreirana deklaracija.

```
class RewriterVarToExplicitType: CSharpSyntaxRewriter
{
    private readonly SemanticModel SemanticModel;

    1 reference
    public RewriterVarToExplicitType(SemanticModel model)
    {
        SemanticModel = model;
    }

    0 references
    public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
    {
        var symbolInfo = SemanticModel.GetSymbolInfo(node.Declaration.Type);
        var typeSymbol = symbolInfo.Symbol;
        var type = typeSymbol.ToDisplayString(
          SymbolDisplayFormat.MinimallyQualifiedFormat);

        var declaration = SyntaxFactory
            .LocalDeclarationStatement(
                SyntaxFactory
                    .VariableDeclaration(SyntaxFactory.IdentifierName(
                      SyntaxFactory.Identifier(type)))
                        .WithVariables(node.Declaration.Variables)
                        .NormalizeWhitespace()
            )
            .WithTriviaFrom(node);
        return declaration;
```

Slika 5: Klasa *RewriterVarToExplicitType*

3.3 Zamena switch naredbe u if-else

Naredbe *switch* i *if-else* imaju sličnu ulogu, ali je u nekim situacijama ispravnije iskoristiti jednu odnosno drugu. Na primer, ukoliko je potrebno ispitati samo jedan uslov (i u skladu sa njegovom vrednošću izvršiti odgovarajuće naredbe), preporučuje se korišćenje *if-else* naredbe. Ukoliko se u kodu naiđe na konstrukte poput prikazanog u kodu 5, ova transformacija to konvertuje u oblik prikazan u kodu 6.

```
1000    int k = 2;

1002    switch (k) {

1004    case 1:
        ...
1006    break;

1008    case 2:
        ...
1010    break;

1012    default:
        break;
1014    }
```

Listing 5: *Switch* naredba

```
1000    int k = 2;

1002    if (k == 1) {
        ...
1004    }
        else if (k == 2) {
1006    ...
        }
1008    else {
        ...
```

```
1010        }
```

Listing 6: *If-else* naredba

Slično kao u prethodnim podsekcijama, ovde se najpre metodom *VisitSwitchStatement* obilaze sve *switch* naredbe, gde se za svaku izdvaja izraz (eng. *expression*) koji figuriše u *switch* naredbi, kao i obeležja (eng. *labels*) i naredbe (eng. *statements*) koje su različite od *break* naredbe. Nakon toga, kreira se novi čvor čija se vrednost postavlja na *null*. Ukoliko unutar *switch* naredbe postoji samo jedno obeležje i to obeležje je *default*, onda se kreira *if* naredba sa uslovom *true* i blokom naredbi koji sadrži sve naredbe iz default obeležja i to postaje vrednost novog čvora. Inače, ukoliko postoji više od jednog obeležja, poziva se privatna funkcija *CreateIfStatement()* kojoj se prosleđuju izdvojeni izraz, obeležja i naredbe. U ovoj funkciji najpre se izdvaja prvo obeležje, zatim sva obeležja osim prvog (ukoliko postoje) i njihove naredbe (ukoliko postoje). Nakon toga, kreira se *if* naredba u zavisnosti od toga da li je prvo obeležje ustvari grupa više obeležja sa zajedničkim naredbama ili ne. Što se tiče *else* grane, proverava se da li je osim prvog obeležja ostalo još neko obeležje (ispod njega) i ono ima bar jednu naredbu (neračunajući naredbu *break*). Ukoliko je ovaj uslov ispunjen i pritom se tu nalazi i obeležje *default*, kreiraće se *else* grana sa blokom naredbi iz *default* obeležja. Ukoliko je uslov ispunjen, ali se tu ne nalazi obeležje *default*, onda se kreira *else-if*, odnosno *else* grana s tim da se ponovo poziva funkcija *CreateIfStatement*. Ukoliko uslov nije ispunjen, znači da je to bilo poslednje obeležje i u tom slučaju će se vratiti *null*. Opisani metod *VisitSwitchStatement()* je dat na slici 6.

```
class RewriterSwitchToIf : CSharpSyntaxRewriter
{
    public override SyntaxNode VisitSwitchStatement(SwitchStatementSyntax node)
    {
        var syntaxNode = (SwitchStatementSyntax)base.VisitSwitchStatement(node);

        var expression = syntaxNode.Expression.ToString();
        var labels = syntaxNode.Sections.Select(s => s.Labels.ToArray());

        var statements = syntaxNode.Sections.Select(s => s.Statements.Where(statement => statement.Kind() != SyntaxKind.BreakStatement));
        StatementSyntax newNode = null;
        if (labels.Count() == 1 && labels.ElementAt(0).Select(l => l.Kind()).Contains(SyntaxKind.DefaultSwitchLabel))
        {
            newNode = SyntaxFactory.ParseStatement(SyntaxFactory.IfStatement(SyntaxFactory.ParseExpression("true"),
                SyntaxFactory.Block(SyntaxFactory.ParseStatement(statements.ElementAt(0).Aggregate("", (x, y) =>
                                                        x.ToString() + " " + y.ToString()))))).ToString());
        }
        else
        {
            newNode = SyntaxFactory.ParseStatement(CreateIfStatement(expression, labels, statements));
        }

        return newNode;
    }
```

Slika 6: Klasa *RewriterSwitchToIf*, metod *VisitSwitchStatement()*

3.4 Uklanjanje praznih naredbi

Poslednji primer transformacije jeste ukljanjanje praznih naredbi. Prazna naredba, tj. tačka-zarez (**;**), obično nastaje greškom, na primer kada se ostavi da bi se kasnije zamenila konkretnom naredbom, ali se to ne desi, ili greškom u kucanju (npr. ;;). Ponekad, programeri koriste praznu naredbu kao telo petlje. To nije dobra praksa, te se preporučuje izbegavanje ovakvog koda. U kodovima 7 i 8 dat je primer ovakvih konstrukata i kako oni treba da izgledaju nakon tranformacije. Kad god se u kodu naiđe na

Slika 7: Klasa *RewriterSwitchToIf*, funkcija *CreateIfStatement()*

praznu naredbu upotrebljenu u kontekstu u kom nije neophodna, ista se jednostavno eliminiše.

```
1000    void DoSomething()
        {
1002    ;
        }

1004
        void DoSomethingElse()
1006    {
        Console.WriteLine("Hello, world!");;

1008
        for (int i = 0; i < 3; Console.WriteLine(i), i++);
1010    }
```

Listing 7: Prazna naredba

```
1000    void DoSomething()
        {
1002    }

1004    void DoSomethingElse()
        {
1006    Console.WriteLine("Hello, world!");

1008    for (int i = 0; i < 3; Console.WriteLine(i), i++)
        {
1010    }
        }
1012
```

Listing 8: Uklonjena prazna naredba

Klase *RemoveEmptyStatementsChecker* i *RewriterRemoveEmptyStatements* implementirane su na sličan način kao i u svim prethodnim primerima. Na slici 8 može se videti implementacija prepisivača. Metod *VisitEmptyStatement* obilazi sve čvorove u stablu koji predstavljajju prazne naredbe. Vrši se provera vrste roditelja čvora, tj. provera da li se naredba

nalazi unutar petlje, *if* ili *else* naredbe. Ako je to slučaj, vraća prazan blok (otvorena i zatvorena vitičasta zagrada { }). Inače, jednostavno vrati *null* - uklanja se ; .

```
class RewriterRemoveEmptyStatements : CSharpSyntaxRewriter
{

    0 references
    public override SyntaxNode VisitEmptyStatement(EmptyStatementSyntax node)
    {
        if (node.Parent.Kind() == SyntaxKind.WhileStatement
            || node.Parent.Kind() == SyntaxKind.IfStatement
            || node.Parent.Kind() == SyntaxKind.ForStatement
            || node.Parent.Kind() == SyntaxKind.ElseClause)
        {
            return SyntaxFactory.Block();
        }
        return null;
    }
```

Slika 8: Klasa *RewriterRemoveEmptyStatements*