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

12. 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.

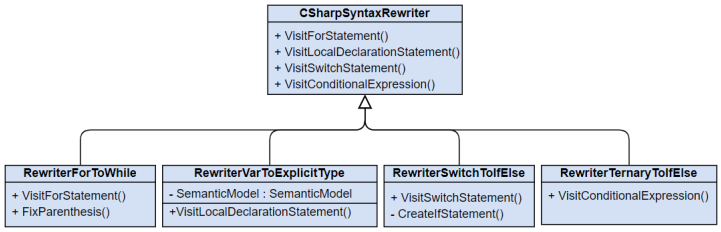
Feature APIs	dozvoljavaju *source code tool* programerima da refaktorišu i isprave kod
Work-space APIs	dozvoljavaju korišćenje plugina za, na primer, pronalaženje referenci promenljive ili oblikovanja koda (eng. *code formatting*)
Compiler APIs	dopuštaju još elegantniju analizu izvornog koda, izlaganjem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

2 Opis arhitekture sistema

U ovoj sekciji bice opisano ..... **TODO**

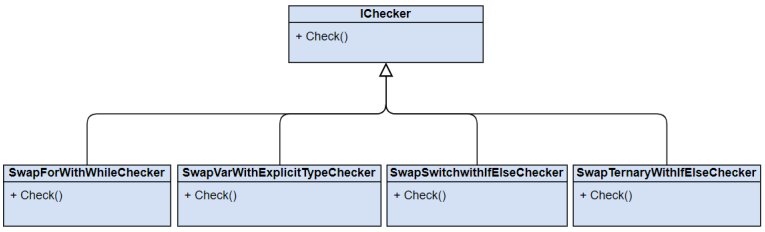
- **TODO** Opis osnovnih modula implementacije - **TODO** Reprezentacija sistema na visokom nivou - **TODO** Način na koji je program razložen na komponente

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 listingu 1, to će biti zamenjeno *while* petljom kao što je prikazano u listingu 2.

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

Listing 1: For petlja

```
1000 int i = 0;

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

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* kome se prosleđuje koren datog stabla. Ovaj metod radi ???????? i kao povratnu vrednost ovaj metod vraća ??????

```
public class SwapForWithWhileChecker : IChecker
{
    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.

```
class RewriterForToWhile : CSharpSyntaxRewriter
{
    public override SyntaxNode VisitForStatement(ForStatementSyntax node)
    {
        var syntaxNode = (ForStatementSyntax)base.VisitForStatement(node);
        var declarationNode = SyntaxFactory.ParseStatement($"{(syntaxNode.Declaration != null ?
                                                               syntaxNode.Declaration.ToString() + ";" : "")}");
        var statement = syntaxNode.Statement.ToString();

        var whileNode = SyntaxFactory.WhileStatement(
            syntaxNode.Condition ?? SyntaxFactory.ParseExpression("true"),
            SyntaxFactory.ParseStatement(
                $"{{{FixParenthesis(statement)}{(syntaxNode.Incrementors.Count != 0 ?
                                                 syntaxNode.Incrementors.ToString().Replace(',', ';') + ";" : "")}\n}}")
            );
        var newNode = SyntaxFactory.ParseStatement($"{{{declarationNode}\n{whileNode}}}");
        return newNode;
    }


    public string FixParenthesis(string statement) {
        if (statement[0] == '{')
        {
            statement = statement.Substring(1, statement.Length-2);
        }
        return statement;
    }
```

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 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,

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 ispiati 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 na listingu 5, ova transformacija to konvertuje u oblik prikazan na listingu 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) {
```

```
    ...
}
else {
    ...
}
```

Listing 6: If-else naredba

Slično kao u prethodnim podsekcijama, ovde se najpre metodom *VisitSwitchStatement* obilaze sve *switch* naredbe, a zatim se izdvaja izraz (eng. *expression*) koji figuriše u *switch* naredbi, kao i naredbe koje su različite od *break* naredbe.

```
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 5: Klasa *RewriterSwitchToIf*

```
    private string CreateIfStatement(string expression, IEnumerable<IEnumerable<SwitchLabelSyntax>> labels,
                                                        IEnumerable<IEnumerable<StatementSyntax>> statements)
    {
        var labelValues = labels.Select(l => l.Select(lb => lb.Kind() == SyntaxKind.DefaultSwitchLabel ?
                                                            "true" : ((CaseSwitchLabelSyntax)lb).Value.ToString()));

        var labelsWithoutCurrent = labels.Skip(1);
        var statementsWithoutCurrent = statements.Skip(1);
        var wat = statementsWithoutCurrent.FirstOrDefault().FirstOrDefault();
        IfStatementSyntax ifStatement = SyntaxFactory.IfStatement(
            SyntaxFactory.ParseExpression(
                labelValues.ElementAt(0).Count() != 1
                    ? labelValues.ElementAt(0).Aggregate((x, y) =>  expression + "==" + x + " || " + expression + "==" + y)
                    : labelValues.ElementAt(0).Select(l => expression + "==" + l.ToString()).First()),
            SyntaxFactory.Block(SyntaxFactory.ParseStatement(statements.ElementAt(0).Aggregate("", (x, y) =>
                                                                    x.ToString() + " " + y.ToString()))),
            labelsWithoutCurrent.Count() != 0 && statementsWithoutCurrent.FirstOrDefault().FirstOrDefault() != null
                ? labelsWithoutCurrent.ElementAt(0).Select(l => l.Kind()).Contains(SyntaxKind.DefaultSwitchLabel)
                    ? SyntaxFactory.ElseClause(SyntaxFactory.ParseStatement("{" + statementsWithoutCurrent.
                                                                            ElementAt(0).
                                                                            Aggregate("", (x, y) =>
                                                                            x.ToString() + " " + y.ToString()) + "}"))
                    : SyntaxFactory.ElseClause(SyntaxFactory.ParseStatement(" " +
                    CreateIfStatement(expression, labelsWithoutCurrent, statementsWithoutCurrent)))
                : null
            );

        return ifStatement.ToString();
    }
```

Slika 6: Klasa *RewriterSwitchToIf* - nastavak

3.4 Uklanjanje praznih naredbi

Poslednji primer prikazuje ukljanjanje praznih naredbi. Kad god se u kodu naiđe na praznu naredbu, ista se jednostavno eliminiše.

```
for (int j = 0; j <= 10; j++) ;
if (true) ;
```

Listing 7: Prazna naredba

```
for (int j = 0; j <= 10; j++)
{
}
if (true)
{
}
```

Listing 8: Uklonjena prazna naredba