RoseLib: A Library for Simplifying .NET Compiler Platform Usage

Nenad Todorović, Aleksandar Lukić, Bojana Zoranović, Renata Vaderna, Željko Vuković, Sebastijan Stoja
Faculty of Technical Sciences, Novi Sad, Serbia
{nenadtod, lukic.aleksandar, bojana.zoranovic, vrenata, zeljkov, sebastijan.stoja}@uns.ac.rs

***Abstract*—Integrating hand-written with generated code can sometimes be challenging part of implementing the MDSE approach, especially if the clean separation is not possible. To overcome this problem, we based our solution on Microsoft's *Roslyn* project. During our research, we found that *Roslyn*'s Syntax Tree API is difficult to use due inherent properties of its implementation. In this paper, we present our *RoseLib* library that abstracts large part of this implementation, which liberates developers from remembering unnecessary details and makes development process much more efficient.**

***Keywords*—MDSE, code generation, Roslyn**

I. Introduction

Model-driven software engineering (MDSE) [1] is a methodology for applying the advantages of modeling for optimization of software engineering activities. One of the application scenarios for MDSE is software development automation, where model-driven techniques are used with the goal of creating a running system. Our research is centered around the development of a code generator that automates a part of the development process of an already existing solution, a large-scale software product line (SPL) [2] developed in C#.NET. A code generation was to be used for the creation of new products and also for maintenance of existing ones. This set limitations on how we could integrate generated code because such use case requires both a modification of existing handwritten code and generation of new code. Introducing changes in architecture in order to enable clean separation was not allowed. In order to enable safe modification of the existing handwritten code, we based our development on .NET Compiler platform [3], also known as *Roslyn*. This platform provides an SDK that adds an API layer on top of the C# and Visual Basic compilers to provide control over compilation process and insight in wealth of available information. Throughout our research, we have experienced benefits and drawbacks of using the *Roslyn* API. It became evident that its inherent properties combined with the complexity of these programming languages cause a steep learning curve and make development difficult. We tried to find solutions to mitigate this problem and optimize our work. Majority of solutions that we have found were focused on creating cases for automated testing [4], refactoring [5], and source code analysis. A tool described in [6] offers substantial support for general code generation. However, it is not compatible with our problem since it demands separate project for generated code and imposes rules about project structure. This paper presents a solution that simplifies work with *Roslyn* by adding a layer that abstracts details of its implementation. With our library, programmers can focus only on concepts they use in their everyday work.

II. *Roslyn* Platform and its Limitations

The *Roslyn* API mirrors the compiler's traditional pipeline, where each phase is treated as a separate component that exposes its information through an object model, as depicted in Fig. 1. In the first, parse phase, compiler tokenizes and parses source code into a syntax tree that follows the language grammar. In the second, declaration phase, compiler analyses declarations from source code and imported meta-data to form named symbols. In the next, binding phase compiler matches identifiers with symbols and exposes the information from the compiler's semantic analysis. And in the final, emit phase all the information built up by the compiler is emitted as an intermediate language(IL) bytecode assembly.

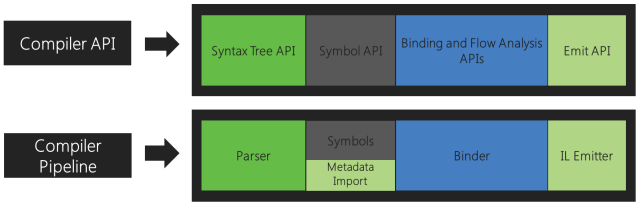

Fig. 1: Compilation phases and their corresponding APIs [3]

This platform is most commonly used for development of tools, that on the one side increase the productivity of programmers, such as IntelliSense [7], code refactorings, finding all references to components, scaffolding, etc. On the other, some tools help improve software quality, enabling automated testing, source code analysis, and custom compile time error checkers.

A. Architecture

The architecture of the *Roslyn* platform consists of two primary and one secondary layer. These are the Compiler API, Workspace API, and Feature API respectively (Fig. 2). Each layer higher in the stack is relying upon the functionality of the previous ones.

The compiler layer contains object models that hold information exposed at each phase of the compiler's pipeline, as described above. These object models are a part of the

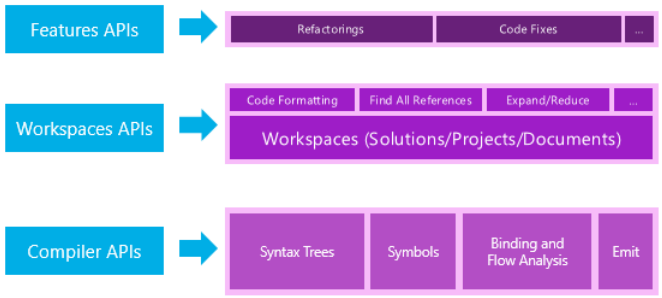

Fig. 2: *Roslyn* platform architecture [3]

immutable snapshot created by a single invocation of the compiler. This snapshot contains assembly references, compiler options, and source code files. On top of the compiler layer, is the workspace layer. This layer provides Workspace API that allows users to do code analysis and refactoring over an entire solution. It organizes all the information about projects in a solution into a single object model that offers direct access to the compiler layer object models without needing to parse files, configure options or manage project-to-project dependencies. The workspace layer also provides commonly used APIs for implementing code analysis and refactoring tools like finding all references, and formatting. On top of this layer is the Feature API which adds code fix and code refactoring functionality. In our work, we have only relied upon the compiler layer, more specifically on the results of the previously mentioned parsing phase.

B. Syntax Trees

Syntax trees represent the lexical and syntactic structure of the source code. A syntax tree, as a data structure, consists of syntax nodes, tokens, and trivia. Each constituent part of the tree has its type and information about its position in the source text.

In the *Roslyn* platform, syntax trees have few fundamental properties. First, syntax trees hold all the source information in full fidelity, which means that a syntax tree obtained from the parser can produce the same text it was parsed from. This means that syntax trees can be used as a way to construct and edit source text i.e. generate code. Second, they are immutable, so direct modification is not possible. After a tree is obtained, it is a snapshot of the current state of the code, and never changes. Because of that, creation and modification of trees is enabled through creation of additional snapshots of the same tree. The trees are efficient in the way they reuse underlying nodes, so the new version can be rebuilt fast and with little extra memory [3].

Immutability ensures thread-safety, which allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. But, this property also makes them very complex and hard to work with.

The syntax trees can also contain errors that are present in source code when the program is incomplete or malformed, in the form of skipped or missing tokens.

1) Syntax Nodes: Syntax nodes are the main building blocks of syntax trees. These nodes represent higher-level syntactical constructs like declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from *SyntaxNode* class.

All syntax nodes are non-terminal nodes with other nodes and tokens as children. All nodes except the root node have a parent node that can be accessed through the Parent property. Each node has a *ChildNodes* method, which returns a list of child nodes in sequential order based on its position in the source text. Each node also has *DescendantNodes*, *DescendantTokens*, or *DescendantTrivia* methods - that represent a list of all the nodes, tokens, or trivia that exist in its sub-tree.

Also, each syntax node subclass exposes all the same children through strongly typed properties. For example, a *BinaryExpressionSyntax* node class has three additional properties specific to binary operators: *Left, OperatorToken*, and *Right*. The type of *Left* and *Right* is *ExpressionSyntax*, and the type of *OperatorToken* is *SyntaxToken*.

Some syntax nodes have optional children. For example, an *IfStatementSyntax* has an optional *ElseClauseSyntax*. If the child is not present, the property returns null.

2) Syntax Tokens: Syntax tokens are the terminals of the language grammar, and they consist of keywords, identifiers, literals, and punctuation. They are never parents of other nodes or tokens.

For efficiency purposes, the *SyntaxToken* type is a value type [3]. Therefore, unlike syntax nodes, there is only one structure for all kinds of tokens with a mix of properties that have meaning depending on the kind of token that is being represented.

3) Syntax Trivia: Syntax trivia represent the parts of the source text that is not relevant to its understanding, such as comments, whitespace and preprocessor directives. Because trivia is not part of the normal language syntax, they are not included in the syntax tree as a child of a node. Instead, they can be accessed by the token's *LeadingTrivia* or *TrailingTrivia* properties. This means that when a source text is parsed, sequences of trivia are associated with tokens.

Like syntax tokens, trivia are value types. The single *SyntaxTrivia* type is used to describe all kinds of trivia.

C. Limitations

Limitations of the *Roslyn* platform are mainly caused by its previously mentioned inherent properties: the immutability and high complexity of the syntax trees. The high complexity can be shown through an example of a simple invocation of *WriteLine* method contained by the *Console* class, shown in Listing 1. An object representation of syntax tree of such invocation is depicted in Figure 3: syntax nodes are colored blue, tokens are colored green, and trivia is colored white and gray.

```
Console.WriteLine("Hello World");
```

Listing 1: WriteLine method invocation

To create a node higher-up in the tree (such as the node of the statement of method invocation, shown previously), all of

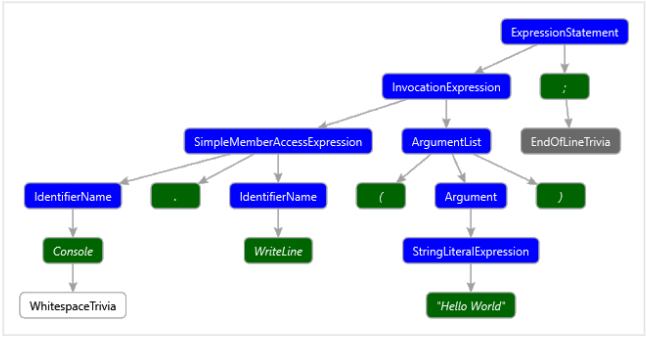

Fig. 3: Resulting syntax tree of a method invocation.

its descendant nodes first must be created and appropriately assembled. This means that a user first has to instantiate suitable tokens (such as tokens for identifiers of *Console* class and *WriteLine* method) and trivia, use them to create a needed syntax node near the bottom of the syntax tree, and then repeat the process to create ancestor nodes until the root node is made. Given this example, it is clear that creation and modification of even the simplest language constructions require in-depth knowledge of the API.

The *Roslyn* API includes a *SyntaxFactory* class that somewhat eases this problem by providing methods that automate some of the work. But, the main problem with this approach is that a user needs to think about the structure of syntax trees. It would be much more convenient if he could express himself using concepts of a programming language itself.

The problem that is caused by immutability can be depicted by an example of renaming a class and its constructors. The action that renames the class creates a new snapshot of the tree, so to rename the first constructor we must first select that constructor in that new tree, and then rename it, which again creates a new snapshot and so on. To mitigate this problem *Roslyn* platform provides an additional DocumentEditor API. This API is a part of the Workspace API, and can be used to make multiple modifications to the tree, and then return a single snapshot. This API has a few drawbacks. First, because it is a part of the Workspace API, it adds a lot of additional code - if we want to work with a new document, we first must create a new workspace, a project in that workspace and only then can we create a new document in that project. Second, all the methods used for modification of the tree are a part of a single object and are not context aware. This means that a user must be fully aware what node can be added as a child of another. For example, to add a new statement into an existing method, a user must know that adding a statement directly into a method node results in an exception and that it should be added to its body node instead. We are going to discuss this API in more detail and compare it to ours in the next section.

Another problem that arises when working with immutable trees is tracking of nodes across snapshots. To enable such functionality, the *Roslyn* APIs offer *TrackNode* and *TrackNodes* methods. These methods accept nodes from the current snapshot of a tree as their parameters, and return a new snapshot. By calling the *GetCurrentNode* or *GetCurrentNodes* methods, we can get tracked nodes from the future snapshots that match the ones from the snapshot that preceded the returned one. Furthermore, when a user wants to add an entirely new node into a tree, the actual node that is added to the tree has a different reference than the passed node. Tracking new nodes is only possible with annotations, that are instances of *SyntaxAnnotation* class. We can create custom annotations, attach them to nodes, and find annotated nodes in the new snapshots.

So, basically, there are three different mechanism that aim to mitigate different problems that arise from working with immutable trees. Complexity of working with these three mechanisms could be reduced by unifying them under a single API that hides the immutability problem and tracks nodes automatically.

To summarize, the library that could make manipulation of *Roslyn* syntax trees easier, by mitigating mentioned limitations, should provide:

- A mechanism that liberates a user from thinking about the immutable nature of syntax trees and keeping track of tree nodes;
- Methods that help find needed tree nodes of various C# language elements;
- Creational methods, that serve to simplify creation of various C# language elements;
- Update methods that serve to alter already existing nodes of syntax trees;
- Insertion methods, which insert nodes into the syntax tree;
- Methods that ease the problem with indentation of generated code.

III. Solution

Aforementioned categories of mechanisms and methods that *RoseLib* provides are discussed in more detail in this section. A simplified class diagram of our solution can be seen in Fig. 4. To preserve diagram's clarity, mapping of some language's properties and elements to *RoseLib* classes, such as nesting, operator overriding and structs (which are, from *RoseLib*'s point of view, similar to classes), are not fully shown in it. The API can be split into two halves: classes that implement searches of the syntax trees (selectors) are depicted on the upper half, and the classes that can alter the syntax trees (composers) are depicted on the bottom half.

All selector classes inherit the *BaseSelector* class, and all the composer classes implement the *IComposer* interface. We aimed to create a fluent API [8], and as a result, all the composer classes inherit corresponding selector classes. For example, *ClassComposer* class inherits the *ClassStructSelector* class. Also, the *BaseSelector* has a reference to an *IComposer*, so that the composer can be the return value of selecors' methods.

Composers are organized into a hierarchy, which follows the structure of a C# language document. Each composer has the *ParentComposer* reference as its field, and methods that can create child composers, when suitable. This hierarchy is going to be explained in detail in the following sections.

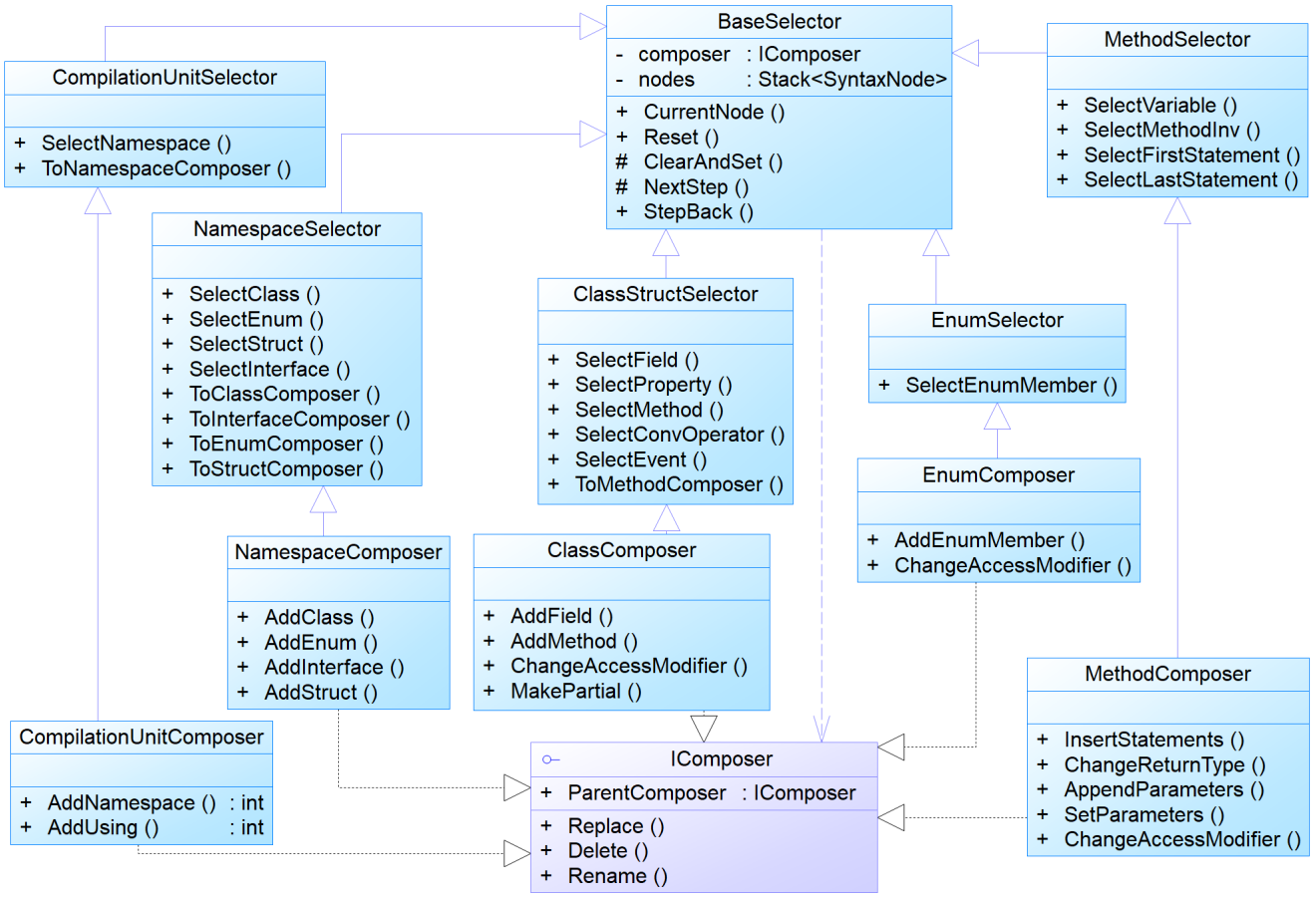

Fig. 4: A simplified class diagram of *RoseLib*.

A. *Tracking nodes between syntax tree versions*

To tackle the problem of tracking nodes between versions of syntax trees, we have created a mechanism mainly guided by the previously mentioned *BaseSelector* class and *IComposer* interface. *BaseSelector* defines a stack, which serves to track selection of the nodes. Pushing nodes into the stack can only be done by classes that inherit *BaseSelector*, and is usually done by methods that select various language elements. After the selection, a user can manually return to a previously selected node by calling the *StepBack* method. All the API methods that search or alter the syntax trees are dependant on the currently selected node. Because the alteration of a syntax tree results in a new tree, selected nodes need to be tracked between versions, and references to them have to be renewed. *IComposer* declares a *Replace* method, which every composer class has to implement, and which should ensure that this renewing passes as expected. Replace methods are implemented using mentioned *Roslyn*'s mechanism for tracking nodes. *Replace* method is available to users so that they could combine both *Roslyn's* and *RoseLib*'s functionality for altering and adding syntax nodes.

B. *Altering a syntax tree*

Creational methods that are implemented in concrete composers are centered around C# language concepts and hide the structure of syntax trees as much as possible. They ease the generation of structural components like namespaces, classes, interfaces, and enumerations, but also fields, properties, constructors, and methods. To enable altering of dynamic properties of code, they also allow the creation of local variables, and method and constructor invocations. Implementation of these creational methods is mainly based on the *Roslyn* API and T4 [9] templates, which are utilized for creation of syntax trees. Creational methods may or may not be dependent on the current selection. Those that are dependent can insert newly created nodes before or after the selected node. These methods will automatically select the inserted node.

Besides the explained methods, there are ones to enable ease altering of nodes already existing in a syntax tree, such as methods for changing types of fields, properties, and variables. Some use cases required modification of existing classes by changing their name, making them partial, alter inheritance, so we provided methods that support those requirements as well. Every composer has a *Rename* and *Delete* methods: rename will change the name of the selected element of the composer, while delete method will remove it from the tree. Deletion cannot be performed on a first element of the stack - that must be done through the parent composer.

C. Composer architecture overview

To better illustrate the architecture of our API, we will describe it through following the usual C# language document structure. As the document represents an instance of a compilation unit, starting point of our API is the *CompilationUnitComposer* class. This composer can be instantiated with or without a file path parameter - passing a file path parameter causes loading of an existing C# source document. It contains methods that search and create elements at the compilation unit level, such as using directives and namespaces. When a specific namespace is selected, a new namespace composer can be created with currently selected node on the top of composer's stack, and the compilation unit composer as its parent composer. Namespace composer contains methods that search and create new classes, interfaces, structs, enums and delegates. Each of these sub-elements map to their own composers. Class composer expands the functionality of the namespace composer with methods that work with fields, properties, methods and events. Also, it can be used to change access modifiers of previously mentioned elements and make the class partial, static, sealed etc.

Method composer exposes methods for search and creation of statements, parameters, and can be used to change the return type or name of a method.

IV. API Comparison

In this section, we will demonstrate how differently these APIs create a simple language structure from the beginning. A listing of resulting structure is shown in Listing 2.

```
using System;
using System.IO;

namespace RoseLib.Example
{
    class ExampleClass
    {
        public string ExampleProp
        {
            get;
            set;
        }

        public void ExampleMethod()
        {
            Console.WriteLine(ExampleProp);
        }
    }
}
```

Listing 2: Code generated by the APIs.

As it has been said previously, creation starts with a compilation unit composer (Listing 3). For the code to be successfully compiled, two using directives are added, and then the example namespace. To add a class to the namespace, we first have to create a namespace composer, which is then used to create the example class. Passed object contains information about the properties of the example class, such as name of the class and access modifiers. And finally, the class is added to the namespace.

Next, we have to create a class composer, which we can then use to add a method and property to the example class. Creation of the example property and method is also done by passing objects that contain the information about their respective properties - their names, access modifiers, are they static, etc.

```
var unitComposer = new CompilationUnitComposer();

NamespaceComposer namespaceComposer = unitComposer
    .AddUsing("System")
    .AddUsing("System.IO")
    .AddNamespace("RoseLib.Example")
    .ToNamespaceComposer();

ClassOptions options = new ClassOptions("ExampleClass");

ClassComposer classComposer = namespaceComposer
    .AddClass(options)
    .ToClassComposer();

PropertyOptions propertyOptions =
    new PropertyOptions("ExampleProp", "string",
    AccessModifierTypes.PUBLIC);

MethodOptions methodOptions =
    new MethodOptions("ExampleMethod", "string",
    AccessModifierTypes.PUBLIC);

MethodComposer methodComposer = classComposer
    .AddProperty(propertyOptions)
    .AddMethod(methodOptions)
    .ToMethodComposer();

methodComposer
    .InsertStatements("Console.WriteLine(ExampleProp);");
```

Listing 3: Code generation with *RoseLib* API.

Unlike our API, *Roslyn* API requires building a tree in a bottom up manner. For this example, *Roslyn's* code is given in 4 and 5. These listings show the order in which nodes must be created. To create a property we have to:

- Create an access modifier token (if needed);
- Create a list of modifier tokens;
- Create a type of the property;
- Create get and set declarations;

Only then we can create a property using suitable methods of the *SyntaxFactory* class.

To create a method we have to:

- Create a return type;
- Create access modifiers, as described previously;
- Create statements (if needed);
- Create the body of a method, using the previously created statements;

Next, to create a class, we only need to provide its identifier. Then, methods, properties and other class members can be added to a class. Namespaces are created in a similar way. Finally, we have to create using directives, and then assemble a compilation unit.

```
var publicKeyword = SyntaxFactory
    .Token(SyntaxKind.PublicKeyword);

SyntaxTokenList modifiers = new SyntaxTokenList();
modifiers = modifiers.Add(publicKeyword);

var propertyType = SyntaxFactory.ParseTypeName("string");

var property = SyntaxFactory
    .PropertyDeclaration(propertyType, "ExampleProp")
    .WithModifiers(modifiers);

var semicolon = SyntaxFactory
    .Token(SyntaxKind.SemicolonToken);

var getter = SyntaxFactory
    .AccessorDeclaration(SyntaxKind.GetAccessorDeclaration)
    .WithSemicolonToken(semicolon);

var setter = SyntaxFactory
    .AccessorDeclaration(SyntaxKind.SetAccessorDeclaration)
    .WithSemicolonToken(semicolon);

property = property.AddAccessorListAccessors(getter, setter);

TypeSyntax returnType = SyntaxFactory.ParseTypeName("void");
var method = SyntaxFactory
    .MethodDeclaration(returnType, "ExampleMethod")
    .WithModifiers(modifiers);

var statement = SyntaxFactory
    .ParseStatement("Console.WriteLine(ExampleProp);");

var block = SyntaxFactory.Block(statement);
method = method.WithBody(block);
```

Listing 4: Creation of the example property and method with *Roslyn* API

```
var @class = SyntaxFactory.ClassDeclaration("ExampleClass");
@class = @class.AddMembers(property, method);

var namespaceId = SyntaxFactory.IdentifierName("RoseLib.Example");
var @namespace = SyntaxFactory.NamespaceDeclaration(namespaceId);

@namespace = @namespace.AddMembers(@class);

var systemId = SyntaxFactory.IdentifierName("System");
var usingSystem = SyntaxFactory.UsingDirective(systemId);

var systemIOId = SyntaxFactory.IdentifierName("System.IO");
var usingSystemIO = SyntaxFactory.UsingDirective(systemIOId);

var compilationUnit = SyntaxFactory.CompilationUnit();

SyntaxList<SyntaxNode> usings = new SyntaxList<SyntaxNode>();

var usingList = new List<SyntaxNode> { usingSystem, usingSystemIO };
usings = usings.AddRange(usingList);

compilationUnit = compilationUnit.WithUsings(usings);

SyntaxList<SyntaxNode> members = new SyntaxList<SyntaxNode>();
members = members.AddRange(new List<SyntaxNode> { @namespace });

compilationUnit = compilationUnit.WithMembers(members);
```

Listing 5: Creation of a class and a namespace with *Roslyn* API.

V. Conclusion

RoseLib currently tackles major part of problems related to code generation in C# including selection, tracking, creation, insertion, alteration and deletion of C# language elements. In addition, generated code is seamlessly integrated with existing hand-written code which leaves development process unaffected.

RoseLib has been proven through implementation and usage in development of industrial SPL applications ref, which confirmed our assumptions – it has improved efficiency of developers by abstracting implementation details of *Roslyn*. Developers with no previous experience with the *Roslyn* platform were able to effortlessly complete numerous tasks related to code generation (such as creating, altering or removing structural components) in an ongoing commercial project using *RoseLib* only.

In near future, we plan to introduce minor changes and extensions to *RoseLib*. For example, error handling process should be improved in order to provide more informative and precise error detection and resolution. In addition, selector methods should be transformed to C# extension method to reduce redundant code.

References

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[2] Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, et al. A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/productlines/index. html*, 2007.

[3] .net compiler platform (”roslyn”). https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/. Accessed: 27-Apr-2018.

[4] Mehrdad Saadatmand. Towards automating integration testing of. net applications using roslyn. In *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on*, pages 573–574. IEEE, 2017.

[5] Bc Matúš Pietrzyk. Automatic refactoring of large codebases.

[6] Roslyn-based code generation. https://github.com/AArnott/CodeGeneration.Roslyn. Accessed: 27-Apr-2018.

[7] Using intellisense. https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx. Accessed: 27-Apr-2018.

[8] Martin Fowler. Fluent interface.

[9] Code generation and t4 text templates. https://msdn.microsoft.com/en-us/library/bb126445.aspx. Accessed: 27-Apr-2018.