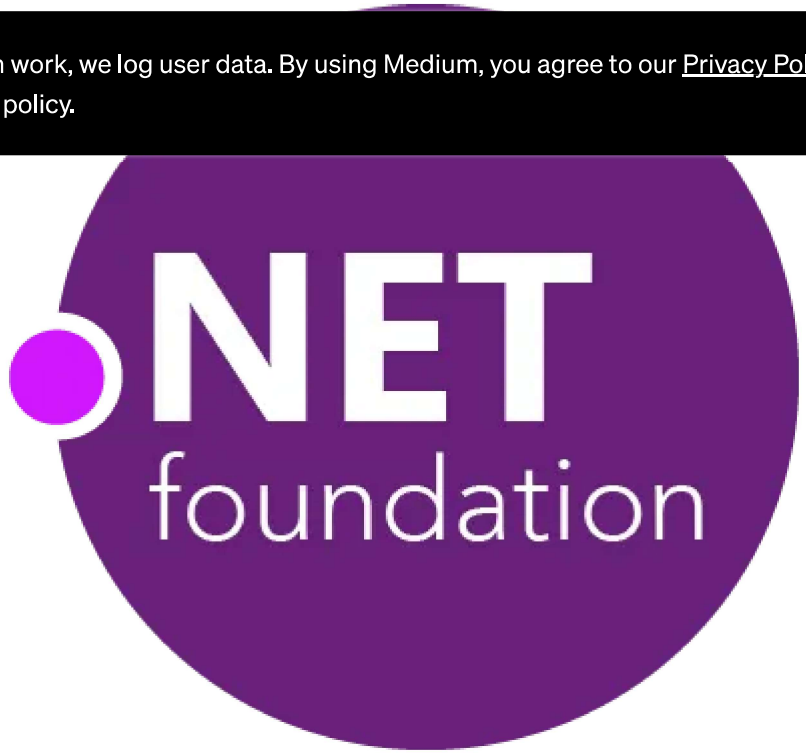
Introduction to Roslyn and its use in program development

Unicorn Developer Follow 31 min read · May 19, 2016

*Author: **Sergey Vasiliev***

Roslyn is a platform which provides the developer with powerful tools to parse and analyze code. It's not enough just to have these tools, you should also understand what they are needed for. This article is intended to answer these questions. Besides this, you will find details about the static analyzer development which uses Roslyn API.



Introduction

The knowledge given in this article was gained during the course of working with PVS-Studio static analyzer, the C# part of which was written using the Roslyn API.

The article can be divided into 2 logical parts:

- General information about Roslyn. An overview of tools provided by Roslyn for parsing and analyzing the code. We provide a description of

entities and interfaces, as well as the point of view of a static analyzer

- Peculiarities that should be taken into account during the development of static analyzers. Description of how to use Roslyn to develop products of this class; what should be considered when developing diagnostic rules; how to write them; an example of a diagnostic.

If we split the article into more detailed parts, we can see the following sections:

- Roslyn; what is it, and why do we need it?
- Preparations for the parsing of projects and analysis of files.
- Syntax tree and semantic model as two major components required for static analysis.
- Syntax Visualizer-extension for Visual Studio, and our helper in the parsing of the code.
- Features that must be taken into account when developing a static code analyzer.
- An example of a diagnostic rule.

Note. Additionally, I suggest reading a similar article “Manual on de

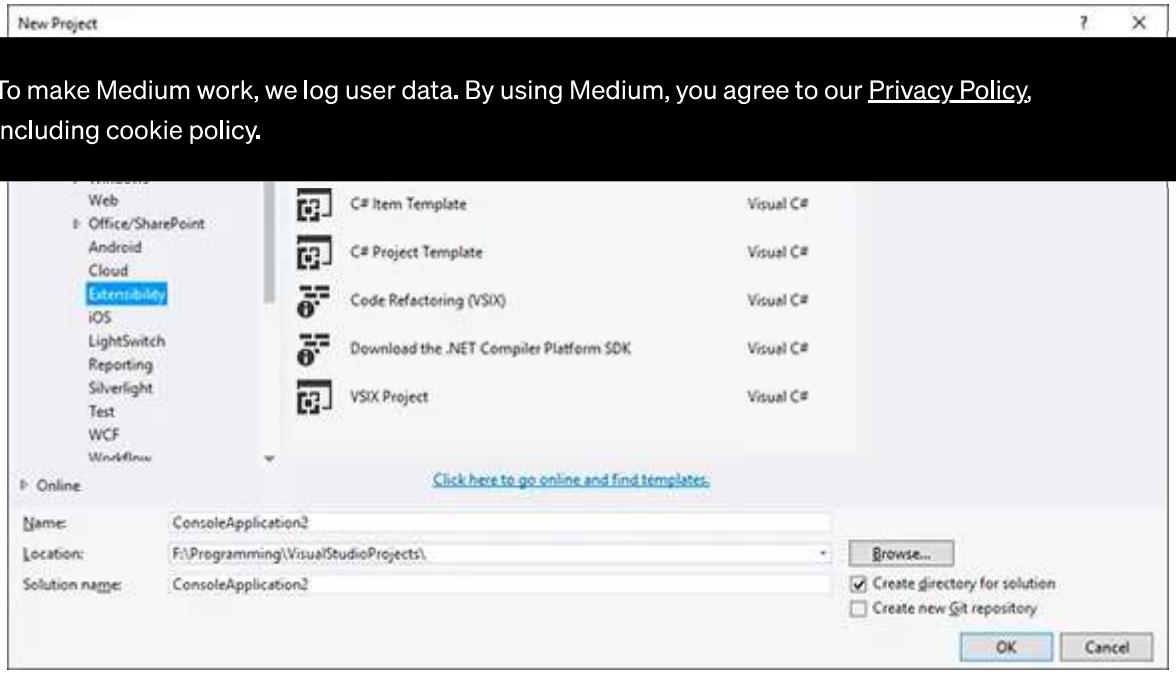
Roslyn

Roslyn is an open source platform, developed by Microsoft, containing compilers and tools for parsing and analysis of code written in C# and Visual Basic.

Roslyn is used in the Microsoft Visual Studio 2015 environment. Various innovations such as code fixes are implemented by means of the Roslyn platform.

Using the analysis tools provided by Roslyn, you can do a complete code parsing, analyzing all the supported language constructs.

The Visual Studio environment enables the creation of tools embedded in the IDE itself (Visual Studio extensions), as well as independent applications (standalone tools).



The source code of Roslyn is available via a repository at GitHub. This allows you to see the way it works and in case of an error — report it to the developers.

The following way of creating a static analyzer and it's diagnostic rules isn't the only one. There is also the possibility of creating diagnostics based on the use of a standard class *DiagnosticAnalyzer.* Built-in Roslyn diagnostics use this solution. This enables, for instance, integration with a standard list of Visual Studio errors, the ability to highlight errors in a text editor, and so on. But we should remember that if these processes are inside the *devenv.exe* process, which is 32-bit, there will be strong limitations on the usage of

memory. In some cases it is critical, and will not allow the in-depth analysis of

th

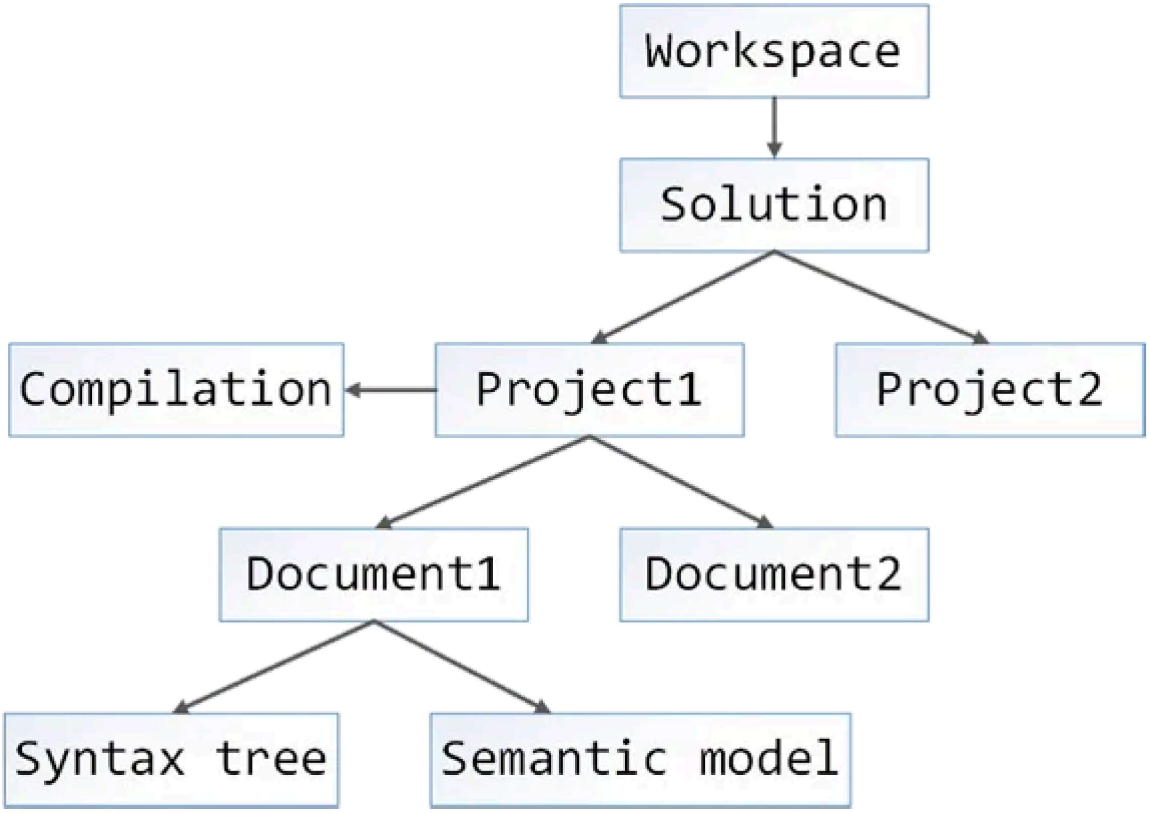
does the paralleling of this process itself.

C# PVS-Studio analyzer is a standalone application, which solves the problem with the restrictions on memory use. On top of this, we get more control over the traversing of a tree; do the paralleling as is necessary, controlling the process of parsing and analyzing the code. As we already had experience in creating an analyzer that works according to this principle, (PVS-Studio C++), we decided to use it when creating the C# analyzer. The integration with Visual Studio environment is similar to the C++ analyzer — we did that also by means of a plugin, calling this standalone-application. Thus, using our groundwork, we managed to create a new analyzer for a new language, bound with the solutions we already had, and embed it in a fully-fledged product — PVS-Studio.

Preparation for the file analysis

Before doing the analysis itself, we have to get a list of files, whose source code is going to be checked, and also get the entities required for correct analysis. We can think of several steps that should be taken to get the data necessary for the analysis:

- Creating the workspace;
- Getting the solution;
- Getting the projects;
- Parsing the project: getting the compilation and the list of files;
- Parsing the file: getting the syntax tree and the semantic model.



Let's discuss each point in detail.

Cr

Creating the workspace is essential in getting the solutions or the projects. To create the workspace you should call a static method *Create* of the *MSBuildWorkspace* class, which returns the object of an*MSBuildWorkspace* type.

Getting the solution

It's necessary to get the solution when we have to analyze several projects of a solution, or all of them. Then, if you have a solution, it's easy to get the list of all the projects included in it.

To get the solution we use the *OpenSolutionAsync* of the *MSBuildWorkspace* object. Finally we get a collection containing the list of projects (i.e. object *IEnumerable<Project>*).

Getting the projects

If there is no necessity to analyze all of the projects, you can get a separate pr
M

Parsing the project: getting the compilation and the list of files

Once we have a list of projects ready for analysis, we can start parsing them. The result of parsing the project should be a list of files for analysis and compilation.

It's simple to get the list of files for the analysis — we use the property *Documents* of the *Project* class.

To get the compilation, we use method *TryGetCompilation* or *GetCompilationAsync*.

Getting the compilation is one of the key points, as it is used to get the semantic model (more details on this will be given later), needed for a thorough and complex analysis of the source code.

To get the correct compilation, the project must be compiled — there should not be any compilation errors, and all dependencies should be located

correctly

An

Below is code that demonstrates different ways to obtain project files using the *MSBuildWorkspace*class:

```
void GetProjects(String solutionPath, String projectPath)
{
  MSBuildWorkspace workspace = MSBuildWorkspace.Create();
  Solution currSolution = workspace.OpenSolutionAsync(solutionPath)
                                   .Result;
  IEnumerable<Project> projects = currSolution.Projects;
  Project currProject = workspace.OpenProjectAsync(projectPath)
                                 .Result;
}
```

These actions shouldn't cause any questions, as we have described them earlier.

Parsing the file: getting a syntax tree and a semantic model

The next step is parsing the file. Now we need to get the two entities which the full analysis is based on — a syntax tree and a semantic model. A syntax

tree is built on the source code of the program, and is used for the analysis of va
th

To get a syntax tree (an object of *SyntaxTree*type) we use the instance method *TryGetSyntaxTree*, or method *TryGetSyntaxTree* of *GetSyntaxTreeAsync* of *Document* class.

A semantic model (an object of *SemanticModel* type) is obtained from the compilation using the syntax tree, which was obtained earlier. To do that we use *GetSemanticModel* method of *Compilation* class, taking an object of *SyntaxTree* type as a required parameter.

The class that will traverse the syntax tree and do the analysis should be inherited from the*CSharpSyntaxWalker*, which will enable to override the traverse methods of various nodes. By calling the *Visit* method that takes the root of the tree as a parameter (we use the *GetRoot* method of the object of *SyntaxTree*) we start a recursive traverse of the nodes of the syntax tree.

Here is the code, showing the way it can be done:

```
  {
    SyntaxTree tree = file.GetSyntaxTreeAsync().Result;
    SemanticModel model = compilation.GetSemanticModel(tree);
    Visit(tree.GetRoot());
  }
}
```

Overridden methods of traversing the nodes

Nodes are defined for every language construct. In turn, for every node type there is a method traversing the nodes of a similar type. Thus, adding the handlers (diagnostic rules) to the traverse methods of the nodes, we can analyze only those language constructs that are of interest to us.

An example of an overridden method of node traversing, corresponding to the *if* statement.

```
public override void VisitIfStatement(IfStatementSyntax node)
{
  base.VisitIfStatement(node);
}
```

By adding the necessary rules to the body of the method, we will analyze all *if s*

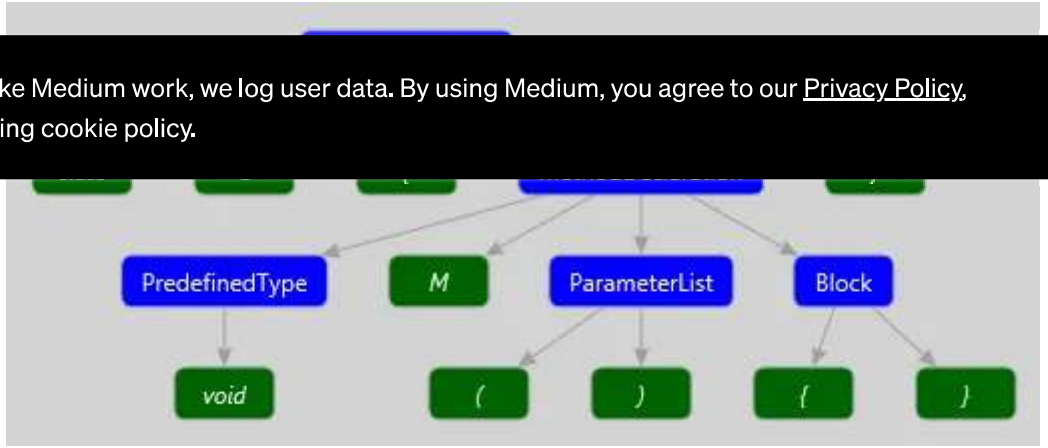
A syntax tree

A syntax tree is a basic element, essential for code analysis. It is the syntax tree that we move along during the analysis. The tree is built on the code, given in the file, which suggests that each file has its own syntax tree. Besides that it should be noted that a syntax tree is unalterable. Well, technically we can change it by calling an appropriate method, but the result of this work will be a new syntax tree, not an edited version of an old one.

For example, for the following code:

```
class C
{
  void M()
  { }
}
```

The syntax tree will be like this:



Nodes of the tree (*Syntax nodes*) are marked in blue, tokens (*Syntax tokens*) — in green.

We can see three elements of a syntax tree that is built by Roslyn on the base of the program code:

- Syntax nodes;
- Syntax tokens;
- Syntax trivia.

Let's have a closer look at these elements, as all of them in one way or another, are used during the static analysis. Some of them are used regularly, and the others — much less often.

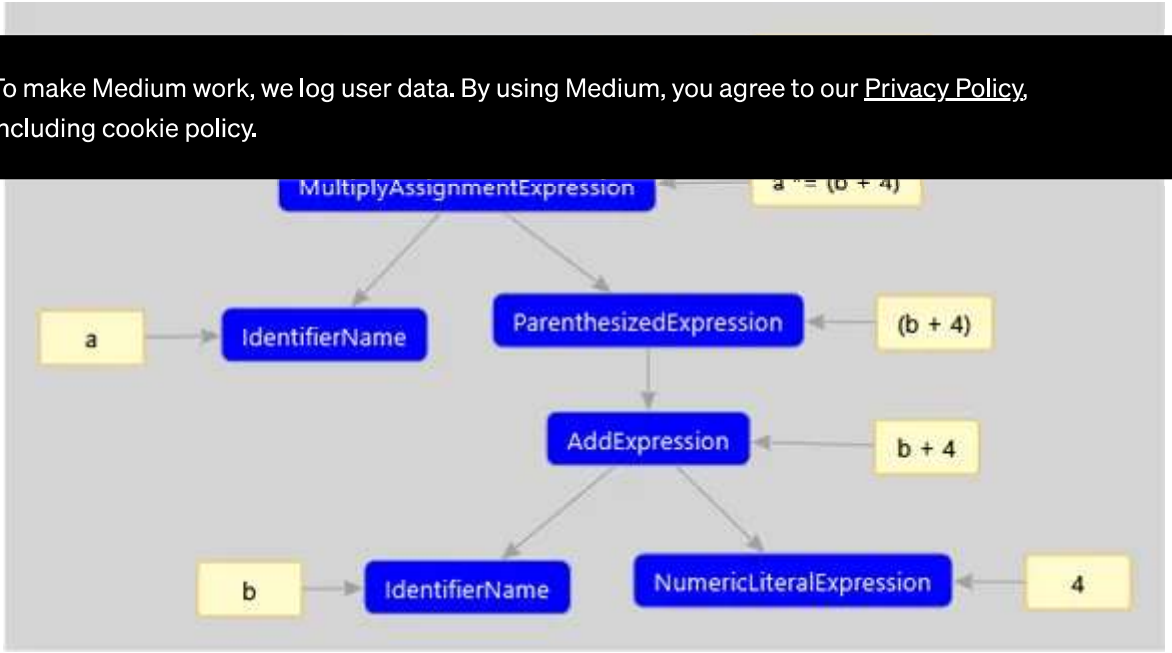
Syntax nodes

Sy declarations, statements, expressions, etc. The main workload of an analyzer is related to the handling of the nodes. These are the nodes that we move along, and the diagnostic rules are based on the traverses of the nodes.

Let's have a look at an example of a tree, equal to the expression

```
a *= (b + 4);
```

In contrast to the previous picture, we have the nodes and commentaries that help us to see which node corresponds to which construction.



A base type

A base node type is an abstract class *SyntaxNode*. This class provides a developer with methods, common for all nodes. Let's enumerate some of the most often used (if something is unclear to you — like *SyntaxKind* or something like that — no worries, we'll speak about it later)

- *ChildNodes* — gets a list of nodes which are child nodes of the current one. It returns an object of*IEnumerable<SyntaxNode>* type;
- *DescendantNodes* — gets a list of all the nodes that are below the current one in the tree. It also returns an object of *IEnumerable<SyntaxNode>* type;

- *Contains* — checks whether the current node includes another node

- *GetLeadingTrivia*-allows you to get elements of the syntax trivia which are prior to the current node, if any;
- *GetTrailingTrivia* — allows you to get elements of the syntax trivia, following this node, if any;
- *Kind* — returns an enumeration *SyntaxKind*, which specifies this node;
- *IsKind* — takes *SyntaxKind* enumeration element as a parameter, and returns a Boolean value indicating whether a particular node type corresponds to the node which is passed as an argument.

In addition, a set of properties is defined in the class. Here are some of them:

- *Parent*-returns a reference to the parent node. It is an extremely necessary property, because it allows moving up along the tree;
- *HasLeadingTrivia* — returns a Boolean value which indicates the presence or absence of elements of syntax trivia, preceding this node;
- *HasLeadingTrivia* — returns a Boolean value which indicates the presence or absence of elements of syntax trivia, following this node.

Derived types

Le[illegible] construct, has its own type, defining a number of properties, simplifying the navigation along the tree and obtaining the required data. These types are numerous. Here are some of them and the way they correspond to the language constructs:

- *IfStatementSyntax* — *if* statement;
- *InvocationExpressionSyntax* — method call;
- *BinaryExpressionSyntax* — infix operation;
- *ReturnStatementSyntax* — an expression with *return* statement*;*
- *MemberAccessExpressionSyntax* — access to the class member;
- And plenty of other types.

Example. Parsing the if statement

Let’s have a look at how to use this knowledge in practice, taking *if* statement as an example.

Let there be such a fragment in the code:


```
if (a == b)
  c *= d;
else
  c /= d;
```

This fragment will be represented as a node of *IfStatementSyntax* at a syntax tree. Then we can easily get the necessary information, accessing various properties of this class:

- *Condition* — returns the condition, being checked in the statement. The return value is a reference of*ExpressionSyntax* type.
- *Else* — returns the else branch of *if* statement, if it is available. The return value is a reference of*ElseClauseSyntax* type;
- *Statement* -returns the body of *if* statement. The return value is a reference of *StatementSyntax*type;

In practice, this is the same as in theory:

```
  ElseClauseSyntax elseClause = node.Else;      /* else
                                                     c /= d;
                                                */
}
```

Thus, knowing the type of the node, it is easy to find other nodes in it. A similar set of properties is defined for other types of nodes, characterizing certain constructs — method declarations, *for* loops, lambdas and so on.

Specification of the node type. SyntaxKind Enumeration

Sometimes it's not enough to know the type of the node. One such case would be prefix operations. For example, we need to pick prefix operations of an increment and decrement. We could check the node type.

```
if (node is PrefixUnaryExpressionSyntax)
```

But such checks would not be enough, because the operators '!', '+', '-', '~' will also suit the condition, as they are also prefix unary operations. So what

should we do?

He[illegible] constructs, its keywords, modifiers and others are defined in this enumeration. Using the members of this enumeration, we can set a specific node type. The following properties and methods are defined to specify the node type in the *SyntaxNode* class.

- *RawKind* — a property of *Int32* type, holding an integer value that specifies this node. But in practice,*Kind* and *IsKind* methods are used more often;
- *Kind*-a method that takes no arguments and returns a *SyntaxKind* enumeration element;
- *IsKind*-a method that takes *SyntaxKind* enumeration element as an argument, and returns *true* or*false* value depending on whether the exact node type matches the type of the passed argument.

Using the methods *Kind* or IsKind, you can easily determine whether the node is a prefix operation of an increment or decrement:

Personally, I prefer using *IsKind* method because the code looks cleaner and more readable.

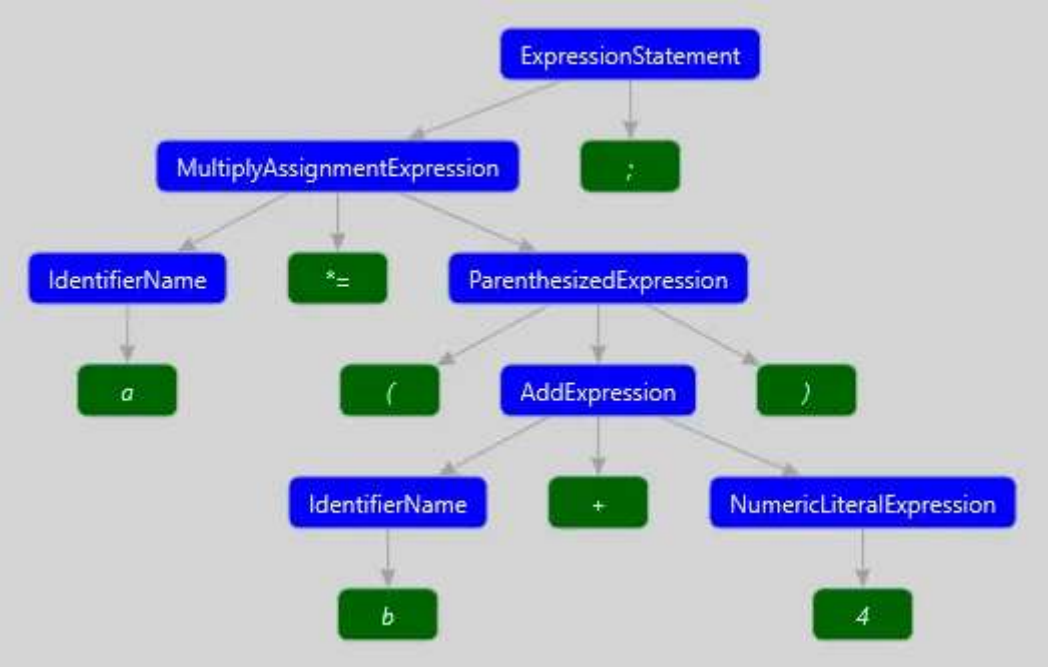
Syntax tokens

Syntax tokens (hereinafter — tokens) are terminals of the language grammar. Tokens are items which are not subject to further parsing — identifiers, keywords, special characters. During the analysis we work directly with them less often than with the nodes of a tree. However, if you still have to work with tokens, this is usually to get the text representation of the token, or to check its type.

Let's have a look at the expression that we mentioned before.

```
a *= (b + 4);
```

The figure shows a syntax tree that is obtained from this expression. But he
co



Usage during the analysis

All tokens are represented by a *SyntaxToken* value type. That's why, to find what a token really is, we use the previously mentioned methods *Kind* and *IsKind*, and enumeration items *SyntaxKind*.

If we have to get a textual representation of the token, it is enough to refer to th

We can also get the token value (a number, for example, if the token is represented by a numeric literal); we should just refer to the *Value* property that returns a reference of an *Object* type. However, to get constant values, we usually use a semantic model and a more convenient method*GetConstantValue* that we will talk about in the next section.

Moreover, syntax trivia (more details in the next section) are also tied to the tokens (actually — to them, rather than to the nodes).

The following properties are defined to work with syntax trivia:

- *HasLeadingTrivia*-a Boolean value which denotes the presence or absence of syntax trivia elements before the token;
- *HasTrailingTrivia*-a Boolean value which denotes the presence or absence of syntax trivia elements after the token;
- *LeadingTrivia*-elements of the syntax trivia, preceding the token;
- *TrailingTrivia*-elements of the syntax trivia, following the token.

Example of usage

Co

```
if (a == b) ;
```

This statement will be split into several tokens:

- Key words: ‘if’;
- Identifiers: ‘a’, ‘b’;
- Special characters: ‘(‘, ‘)’, ‘==’, ‘;’.

An example of getting the token value:

```
a = 3;
```

Let literal ‘3’ come as a node to be analyzed. Then we get the text and numeric representation in the following way:

```
}
```

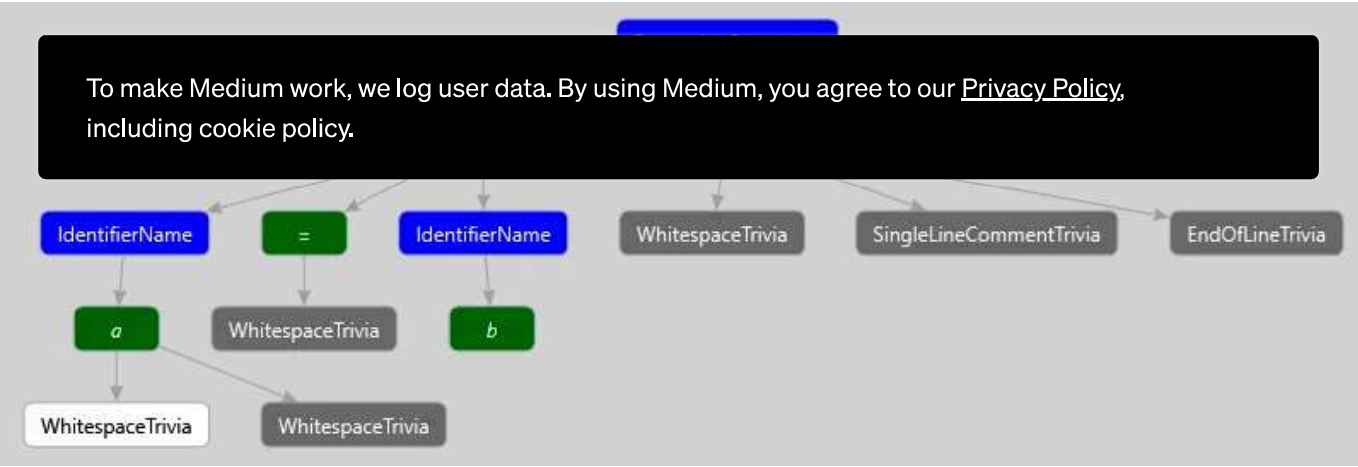
Syntax trivia

Syntax trivia (additional syntax information) are those elements of the tree which won't be compiled into IL-code. These include elements of formatting (spaces, line feed characters), comments, and preprocessor directives.

Consider the following simple expression:

```
a = b; // Comment
```

Here we can see the following additional syntax information: spaces, single-line comment, an end-of-line character. The connection between additional syntax information and tokens is clearly seen on the figure below.



Usage during the analysis

As we have said before, the additional syntax information is connected with tokens. There is Leading trivia, and Trailing trivia. Leading trivia — additional syntax information, preceding the token, trailing trivia — additional syntax information, following the token.

All the elements of additional syntactic information have the type *SyntaxTrivia*. To define what exactly the element is (a space, single-line, multiline comment or something else) we use the *SyntaxKind*enumeration and the methods *Kind* and *IsKind*.

As a rule, the main work with additional syntactic information is aimed at defining what the elements of it are, and sometimes — to the text analysis.

An example of use

Su

```
// It's a leading trivia for 'a' token
a = b; /* It's a trailing trivia for
          ';' token */
```

Here a single-line comment will be bound to the token ‘a’, and the multiline comment — to the token ‘;’.

If we get *a=b;* expression as a node, it’s easy to get the text of a single-line and multiline token like this:

```
void GetComments(ExpressionSyntax node)
{
  String singleLineComment =
    node.GetLeadingTrivia()
        .SingleOrDefault(p => p.IsKind(
                                SyntaxKind.SingleLineCommentTrivia))
        .ToString();

  String multiLineComment =
    node.GetTrailingTrivia()
        .SingleOrDefault(p => p.IsKind(
                                SyntaxKind.MultiLineCommentTrivia))
```

```
            .ToString();
```


Summary

Summing up the information from this section we can see the following points regarding the syntax tree:

- A syntax tree is a basic element necessary for static analysis;
- A syntax tree is immutable;
- Doing the traverse of the tree, we traverse different language constructs; each of them has it's own type defined.
- For each type that corresponds to a syntax language construct, there is a traverse method; we can override it and specify the node processing logic;
- There are three main elements of the tree — syntax nodes, syntax tokens, syntax trivia;
- Syntax nodes — syntax language constructions. These are declarations, attributions, operators, etc.

- Syntax tokens are the terminals of the language grammar. Syntax tokens

- Syntax trivia-additional syntax information. These are comments, preprocessor directives, spaces, etc.

Semantic model

A semantic model provides information on objects, and the types of objects. This is a very powerful tool that allows you to carry out deep and complex analysis. This is why it's very important to ensure correct compilation, and a correct semantic model. Reminder: the project must be a compiled one.

We should also remember that we work with nodes, not objects. That's why neither *is* operator, nor*GetType* method, will work to get the information, as they give information about the node, not about the object. Let's analyze the following code, for example.

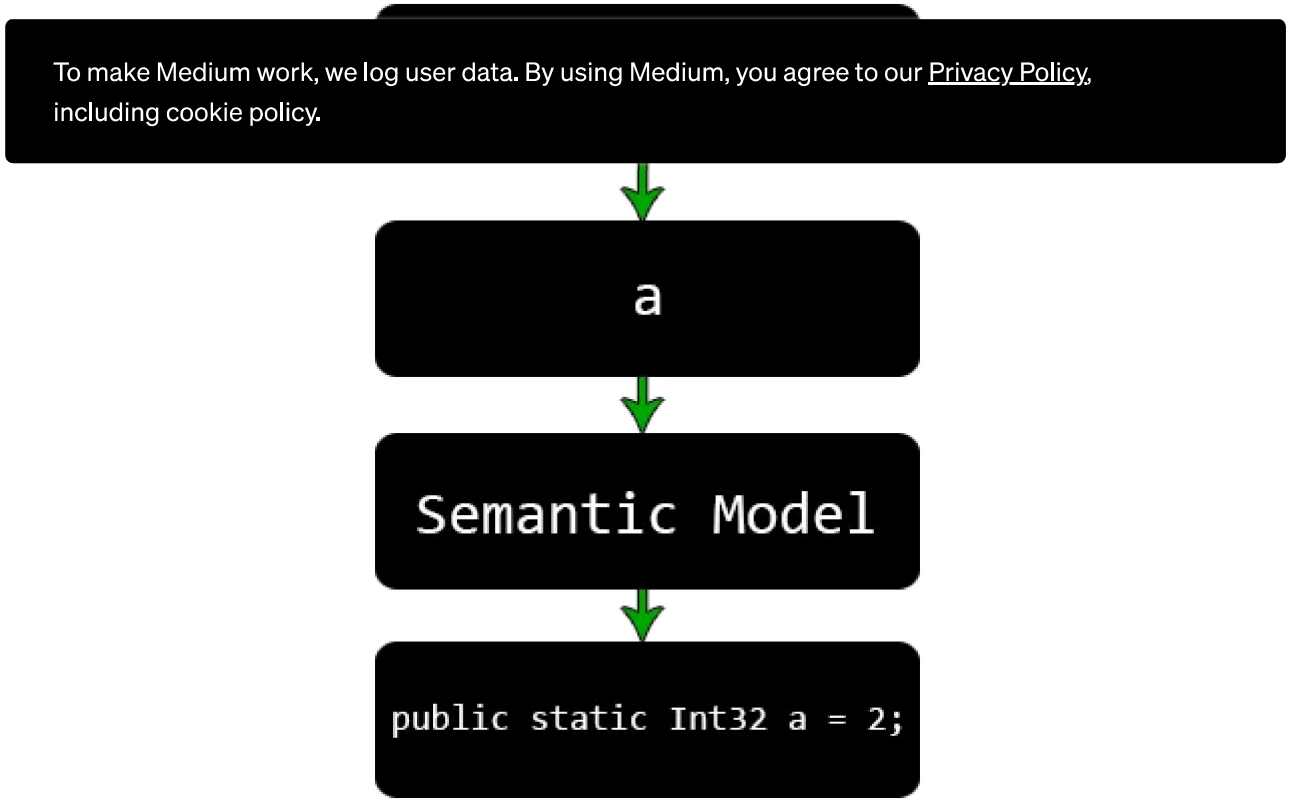
```
a = 3;
```

We can only suggest what *a* is in this expression. It's impossible to say whether it is a local variable, a property, or a field; we can only make an

assumption. Yet no one is interested in seeing the guesses, we need exact

We could have tried to move up along the tree until we see the variable declaration, but this would be too lavish from the point of view of the performance and the code size. Moreover, this declaration might be located somewhere in a different file, or even in a third-party library, whose source code we don't have.

So, a semantic model is of great use for us here.



We can say that there are three functions used most often, which are provided by a semantic model:

- Getting information about the object;
- Getting information about the type of an object;
- Getting constant values.

We will speak in detail about these points, as they are really important, and wi

Getting information about the object. Symbol

So-called symbols provide information on an object.

The base interface of the symbol — *ISymbol*, which provides methods and properties that are common for all the objects, regardless of whether they are — fields, properties, or something else.

There is a number of derived types that a programmer can cast to, to get more specific information about the object. Such methods are *IFieldSymbol*, *IPropertySymbol*, *IMethodSymbol* and others.

For instance, if we use the casting to the interface *IFieldSymbol*, and address the field *IsConst* you can find out if the node is a constant filed. If we use the *IMethodSymbol* interface, we can learn if this method returns any value.

There is also a *Kind* property that is specified for the symbols, and returns the enumeration elements. This enumeration is similar to the *SyntaxKind*

regarding its meaning. That is, by using the *Kind* property, you can see what we

An example of use. Let's find out, whether this node is a constant field

For example, suppose you have a following field definition:

```
private const Int32 a = 10;
```

And somewhere below — the following code:

```
var b = a;
```

Let's suppose that we need to find out if *a* is a constant field. Using a semantic model, we can get the necessary information about the *a* node from the given expression. The code for getting the information will be like this:

```
  ISymbol smb = model.GetSymbolInfo(identifier).Symbol;
  if (smb == null)
    return null;
  return smb.Kind == SymbolKind.Field &&
         (smb as IFieldSymbol).IsConst;
}
```

First we get a symbol for the identifier, using the *GetSymbolInfo* method of an object having*SemanticModel* type, after which we address the Symbol field (it is this field that contains the necessary information, so there is no point in storing the structure *SymbolInfo* that is returned by *GetSymbolInfo*).

After the verification against *null*, using the *Kind* property which specifies the symbol, we are sure that the identifier is a real field. If it is really so — we'll cast to the derived interface *IFieldSymbol*, which will allow addressing the *IsConst* property, and let us get the information about the constancy of the field.

Getting information about the type of an object Interface ITypeSymbol

It is often necessary to know the type of the object that is represented by a node. As I wrote before, the*is* operator and the *GetType* method are not

suitable because they work with the node type, rather than the analyzed ob

Fortunately, there is a way out, and quite a graceful one. You can get the necessary information using the *ITypeSymbol* interface. To get it we use *GetTypeInfo* method of an object having *SemanticModel*type. In general, this method returns the *TypeInfo* structure that contains two important properties:

- *ConvertedType*-returns information about the type of an expression after the implicit casting. If there wasn't any cast, the returned value would be similar to the one that is returned by the *Type* property;
- *Type*-returns the type of the expression given in the node. If it's not possible to get the type of the expression, the *null* value is returned. If the type cannot be defined because of an error, then the*IErrorTypeSymbol* interface is returned.

Using the *ITypeSymbol* interface that is returned by these properties, you can get all the information about the type. This information is retrieved due to the access to the properties, some of which are listed below:

- *AllInterfaces*-a list of all the interfaces that are implemented by the type

- *BaseType* — a base type;
- *Interfaces* — a list of interfaces that are implemented specifically by this type;
- *IsAnonymousType* — information on whether the type is an anonymous one;
- *IsReferenceType*-information on whether the type is a reference one;
- *IsValueType*-information on whether the type is a value one;
- *TypeKind*-specifies the type (similar to the *Kind* property for *ISymbol* interface). It contains information about what the type is — a class, structure, enumeration, etc.

We should note that you can see not only the object type, but the entire expression type. For example, you can get the type of the expression *a* + *b*, and the types of the variables *a* and *b* separately. Since these types may vary, it's very useful during the development of some diagnostic rules to have the possibility of getting the types of the whole expression.

Besides, as for the *ISymbol* interface, there is a number of derived interfaces, wh

An example of use. Getting the names of all the interfaces, implemented by the type

To get the names of all interfaces, implemented by the type and also by the base type, you can use the following code:

```
List<String> GetInterfacesNames(SemanticModel model,
                                IdentifierNameSyntax identifier)
{
  ITypeSymbol nodeType = model.GetTypeInfo(identifier).Type;
  if (nodeType == null)
    return null;
  return nodeType.AllInterfaces
                 .Select(p => p.Name)
                 .ToList();
}
```

It’s quite simple, all the methods and properties were described above, so you shouldn’t have any difficulties in understanding the code.

Getting constant values

A semantic model can also be used to get constant values. You can obtain th
ha

A semantic model provides a more convenient interface for this. In this case we don't need tokens, it is enough to have the node from which you can get a constant value — the model will do the rest. It is very convenient, as during the analysis the main workload is connected with the nodes.

To get constant values we use *GetConstantValue* method that returns a structure *Optional<Object>*using which it's very easy to verify the success of the operation, and get the needed value.

An example of use. Getting constant field values

For example, suppose you have the following code to analyze:

```
private const String str = "Some string";
```

If there is a *str* object somewhere in the code, then, using a semantic model, it's easy to get a string that the field refers to :

```
  if (!optObj.HasValue)
    return null;
  return optObj.Value as String;
}
```

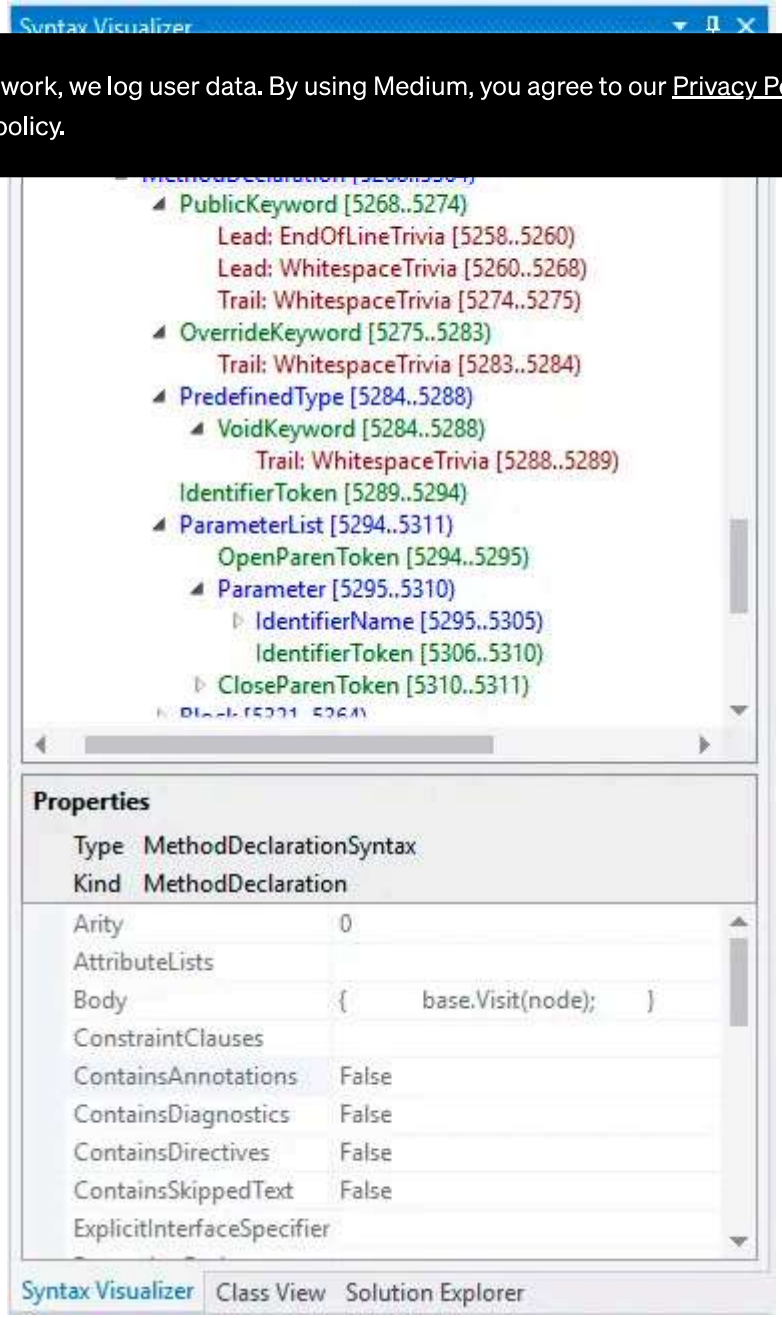
Summary

Summing up the information from this section we can see the following points regarding the semantic model:

- A semantic model provides semantic information (about objects, their types, etc.);
- It is necessary to do in-depth and complex analysis;
- The project must be compiled to get a correct semantic model;
- *ISymbol* interface provides information about an object;
- *ITypeSymbol* provides information about the type of an object;
- We can the values of constant fields and literals with the help of a semantic model.

Syntax visualizer

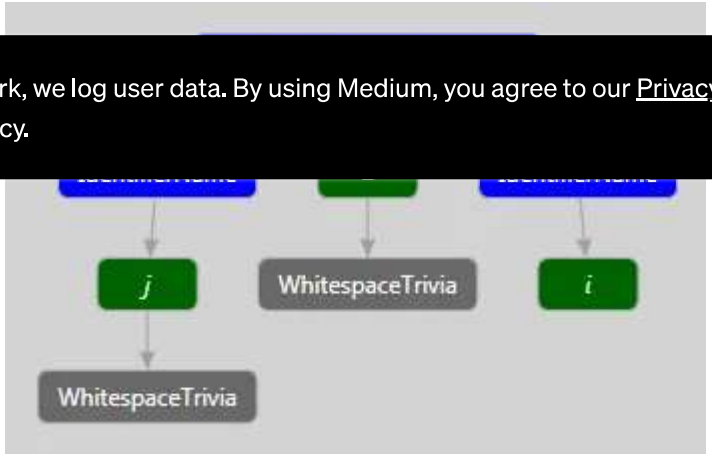
Sy al
St Visual Studio Gallery). This tool, as the name suggests, displays the syntax tree.



As you can see on the picture, blue elements are the nodes, green are tokens, re
fir
possibility to get the *ISymbol* and *ITypeSymbol* interfaces for the nodes of the tree.

This tool is useful indeed in the TDD methodology, when you write a set of unit-tests before the implementation of a diagnostic rule, and only after that start programming the logic of the rule. The visualizer allows easy navigation along the written code; it also allows you to see which node traverse needs to be subscribed to, and where to move along the tree; for which nodes we can (and need) to get the type and the symbol, which simplifies the development process of the diagnostic rule.

There is one more variant for displaying the tree, besides the format that we have just seen. You should open a context menu for the element and choose *View Directed Syntax Graph*. I got the trees of various syntactic constructs, given in this article, by means of this mechanism.



True life story

Once during the development of PVS-Studio we had a situation where we had a stack overflow. It turned out that one of the projects we were analyzing — ILSpy — had an auto-generated file Parser.cs that contained a crazy amount of nested *if* statements. As a result, the stack was overflowed during the attempt to traverse the tree. We have solved this problem by increasing the maximum stack size for the threads, where the tree is traversed, but the syntactic visualizer and Visual Studio still crash on this file.

You can check it yourself. Open this awesome file, find this heap of *if* statements, and try to have a look at the syntax tree (line 3218, for example).

Factors to consider when creating a static analyzer

There is a number of rules that should be followed during the development of

hi

- To do an in-depth analysis we have to have full information about all the types in the code. In most diagnostic rules it's not enough to do a simple traverse of the nodes of a tree; often we have to process the types of expressions, and get the information about the objects to be analyzed. This requires a semantic model that needs to be correct. And of course, the project should be compiled, having all necessary dependencies. Nevertheless, even if it is not so, we shouldn't disregard various checks of the results that we get by means of a semantic model;
- It is important to choose the type of the node to start the analysis. This will allow for less navigations along the tree and various castings. It will also reduce the amount of code, simplifying its support. In order to determine the starting node of the analysis, use a syntactic visualizer;
- If there is no certainty that the code is wrong, it is better not to issue the warnings. Within reason, of course. The thing is that if the analyzer gives too many warnings, there will be too much noise from all those false positives, making it hard to notice a real error. On the other hand, if there are no warnings at all, there is no use in the static analyzer. That's why we

have to compromise, but the final goal is to minimize the number of false

- When developing diagnostic rules, it is important to foresee all possible, impossible, as well as improbable cases that you may encounter in the course of the analysis. To do that it's necessary to write a large number of unit tests. They should be positive — code fragments that trigger the diagnostic rule, and negative — those fragments that the warnings shouldn't be issued for;
- TDD methodology fits the development of diagnostic rules perfectly well. Initially, the developers start by writing positive and negative unit tests, and only then start implementing the diagnostic rule. This will make it easier to navigate along the syntax tree as the implementation goes on, because you will have examples of various trees. Moreover, at this stage, a syntactic visualizer will be especially useful;
- It is important to test the analyzer on real projects. But in reality, it's almost impossible to cover all the cases that the analyzer will encounter with unit-tests. Checking the analyzer on real projects will allow you to detect the spots where the analyzer fails to work correctly, track the changes in the work of the analyzer, and increase the base of unit-tests.

Algorithm for writing diagnostic rules

Searching for errors is mostly done by means of various diagnostic rules.

Th

sp

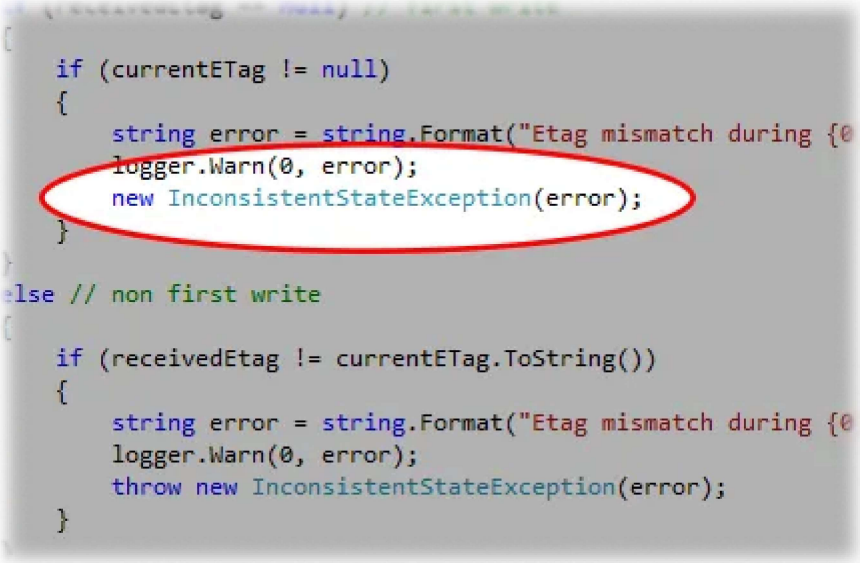
- The first step is to formulate the main point of the rule. Before the

- When there is a kind of a form for the diagnostic rule, and it's quite clear in which situations the warnings will be issued, we have to start writing unit-tests; specifically — develop sets of positive and negative tests. Positive tests should trigger your diagnostic. In the early stages of development, it is important to make the base of the positive unit tests as big as possible, as this will help catch more suspicious cases. Negative tests also deserve attention. As you develop and test the diagnostics, the base of negative unit tests will be continuously replenished. Due to this fact, the amount of false positives will decrease, leading the ratio of good to bad warnings in the desired direction;
- Once the basic set of unit tests is ready, we can start implementing the diagnostic. Do not forget to use a syntactic visualizer-this tool can be of great help in the programming process;
- After the diagnostic is ready, and all unit-tests pass successfully, we proceed to testing on real projects. This detects false positives (and maybe even crashes) in your diagnostic, and enlarge the base of unit tests. The more open source projects are used for testing, the more

possible options of the analyzed code you are considering, the better and

- After testing real projects you will most likely have to refine your diagnostic, because it's very hard to hit the bull's-eye the first time. Well, okay, this is a normal process! Make the necessary changes and test the rule again;
- Repeat the previous point until the diagnostic shows the desired result. After that you can be proud of the work done.

An example a diagnostic rule. Searching for a missing throw statement



In the PVS-Studio static analyzer, there is a diagnostic V3006 that searches for ... ob... ed anywhere, it doesn't return from the method, and so on.) Then most likely, the programmer missed a *throw* statement. As a result the exception won't be generated, and the object will be destroyed during the next garbage collection.

As we have thought out the rule, we can start writing unit tests.

An example of a positive test:

```
if (cond)
  new ArgumentOutOfRangeException();
```

An example of a negative test:

```
if (cond)
  throw new FieldAccessException();
```

We can point out the following points in the algorithm of the diagnostic's wo

- Subscribe to the traverse of the nodes of *ObjectCreationExpressionSyntax* type. This node type corresponds to the creation of an object with the *new* statement — it's exactly what we need;
- We make sure that the object type is compatible with the *System.Exception* (i.e. either with this type of or with a derived one). If it is so, we will consider this type to an exception one. To get the type we will use the semantic model (the model gives the ability to get the type of the expression);
- Then we check that the object is not used (the reference to the object is not written anywhere, and not passed anywhere);
- If the previous points are done — we'll issue a warning.

We will give the description of a possible implementation of such this diagnostic rule. I have rewritten the code, and simplified it, to make it easier to understand. But even such a small rule copes with this task and finds real errors.

The general code for searching the missing *throw* statement:

```
{
  if (!IsExceptionType(model, node))
    return false;

  if (IsReferenceUsed(model, node.Parent))
    return false;

  return true;
}
```

You can see the steps of the algorithm, described earlier. In the first condition there is a check that the type of object is the exception type. The second check is to determine whether the created object is used or not.

SemanticModelAdapter can be a little confusing. There is nothing tricky here, it's just a wrapper around the semantic model. In this example, it is used for the same purposes as the general semantic model (*SemanticModel* object type).

Method of checking whether the type is the exception one:

```
Boolean IsExceptionType(SemanticModelAdapter model,
                        SyntaxNode node)
```

```
{
    nodeType = nodeType.BaseType;

  return Equals(nodeType?.FullName(),
                ExceptionTypeName);

}
```

The logic is simple — we get information about the type, and check the whole inheritance hierarchy. If we see in the result that one of the basic types is *System.Exception*, we think that the type of the object is the exception type.

A method to check that the reference isn't passed anywhere and isn't stored anywhere.

```
Boolean IsReferenceUsed(SemanticModelAdapter model,
                    SyntaxNode parentNode)
{
  if (parentNode.IsKind(SyntaxKind.ExpressionStatement))
    return false;

  if (parentNode is LambdaExpressionSyntax)
    return (model.GetSymbol(parentNode) as IMethodSymbol)
             ?.ReturnsVoid == false;
```

```
return true;
```


We could check if the reference is used, but then we'll have to consider too many cases: return from the method, passing to the method, writing to the variable, etc. It's much easier to have a look at cases where the reference isn't passed anywhere, and not written anywhere. This can be done with the checks that we have already described.

I think the first one is quite clear — we check that the parent node is a simple expression. The second check isn't a secret either. If the parent node is a lambda expression, let's check that the reference is not returned from lambda.

Roslyn: Advantages and disadvantages

Roslyn is not a panacea. Despite the fact that it is a powerful platform for parsing and analyzing code, it also has some drawbacks. At the same time we see plenty of pluses. So, let's have a look at the points from both categories.

Advantages

- A large number of node types. This may be quite scary in the early stages [illegible] language constructs, and thus analyzing the necessary code fragments. Besides that, each node type offers a distinctive set of features, making the task of getting the required data easier;
- Easy navigation along the tree. It's enough to address the properties of the nodes to move along the tree and obtaining the necessary data. As it was said before, every type of the nodes has its own set of properties, which simplifies the task;
- A semantic model. The entity, which allows receiving information about objects and types, providing the same convenient interface, is a very strong side of the platform;
- Open source code. You can follow the development process of the platform, if you want to see what and how it goes. Of course, you can also take part in the development process by telling the developers about the bugs you find — it will be beneficial for everybody.

Disadvantages

- Making the source code of some projects open can cause various

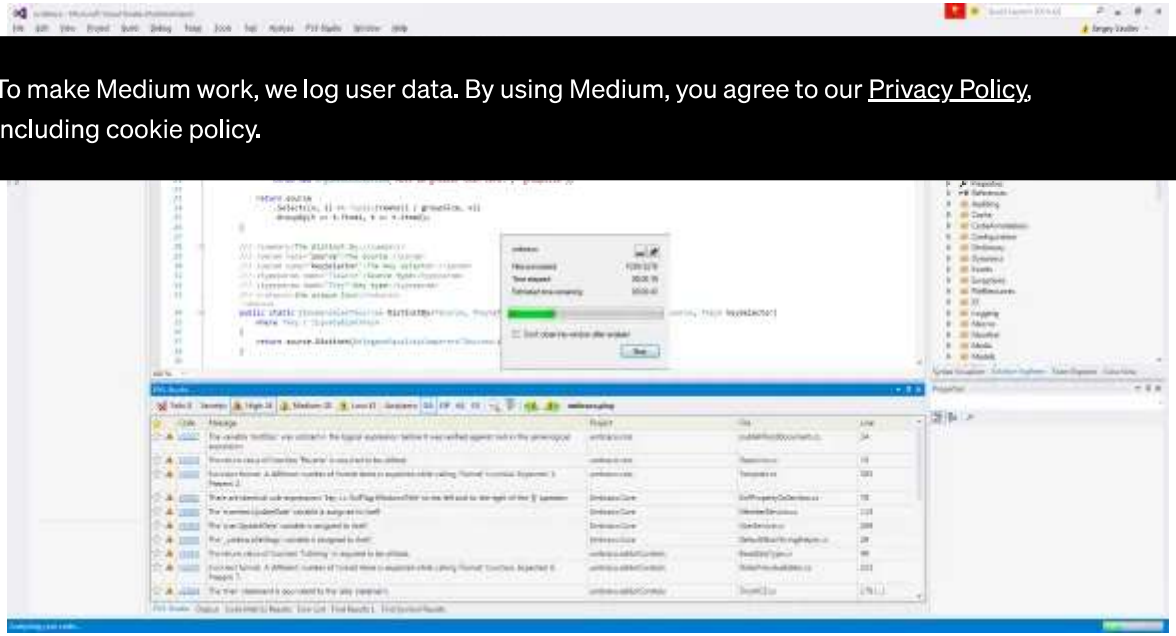
  getting the correct compilation, and a semantic model as a result. This cuts deep analysis to the root, because without a semantic model deep analysis is not possible. You have to use additional resources (for example, MSBuild) to parse solutions/projects correctly;

- You have to invent your own specific mechanisms for seemingly simple things. For example-comparison of nodes. The *Equals* method simply compares the references, which is clearly insufficient. So you have to invent your own mechanisms for comparison;

- A program built on the basis of Roslyn, can consume lots of memory (gigabytes). For modern 64-bit computers with large storage capacity it is not critical, but this feature is worth keeping in mind. It is possible that your product will be useless on slower obsolete computers.

PVS-Studio is a static code analyzer that uses Roslyn API

PVS-Studio is a static analyzer for bug detection in the source code of programs, written in C, C++ and C#.



That part of the analyzer, which is responsible for checking the C# code is written on Roslyn API. The knowledge and rules that are described above aren't pulled out of a hat, they are obtained and formulated during the work with the analyzer.

PVS-Studio is an example of a product you can create using the Roslyn. At this point we have more than 80 diagnostics implemented in the analyzer. PVS-Studio has already found a lot of errors in various projects. Some of them:

- Roslyn;

- MSBuild;
- [redacted]
- SharpDevelop;
- MonoDevelop;
- Microsoft Code Contracts;
- NHibernate;
- Space engineers;
- And many more.

But the proof of the pudding is in the eating, in our case — it's better to have a look at the analyzer yourself. You can download it here and see what it will find in your projects.

Some may wonder: "Have you found anything of interest during the checking process?" Yes, we have. Plenty of bugs. If someone thinks that professionals don't make mistakes, I suggest looking at an error base, found in open source projects. Additionally you may read about the checks of various projects in the blog.

Overall results

General

- This opens up the space to create a variety of applications, including static analyzers;
- For a serious analysis, the project must be compiled, as it is the main prerequisite for getting a correct semantic model;
- There are two entities that the analysis is based on — a syntax tree, and semantic information. Only using both of them combined is it possible to do really serious analysis.
- The code of it is open — download and use;
- Syntax visualizer is a useful extension that will help you to work with the platform.

A syntax tree

- Is built for each file and is unalterable;
- It consists of 3 main components — syntax nodes, syntax tokens, syntax trivia;
- Nodes are the main elements of the tree that we work with;

- A certain type is defined for each node, which allows you to easily get the

- Tokens — terminals of the language grammar, representing identifiers, keywords, dividers, etc.;
- Additional syntax information — comments, spaces, preprocessor directives, etc.;
- Use the *IsKind* and *SyntaxKind* enumeration to specify the type of the tree element.

Semantic model

- It should be correct, in order to do qualitative analysis;
- It allows you to get information about the objects and their types;
- Use the *GetSymbolInfo* method, *ISymbol* interface and its derivatives to get the information about the object itself.
- Use the *GetTypeInfo* method, the *ITypeSymbol* interface and its derivatives to get information about the object's type or expression;
- Use the *GetConstantValue* method to get constant values.

Static analysis

- warning. You shouldn't clutter the result of the analyzer's work with false positives;
- We can see a general algorithm for writing the diagnostics, which will help implement powerful and functional diagnostic rules;
- Use a syntactic visualizer;
- The more unit tests, the better;
- When developing diagnostic rules, it's important to test them on various real projects.

Conclusion

Summing up, Roslyn is a really powerful platform, on the base of which you can create different multifunctional tools — analyzers, refactoring tools, and many more. Big thanks to Microsoft for the Roslyn platform, and the chance to use it for free.

However, it's not enough just to have the platform; you must know how to work with it. The main concepts and principles of work are described in this article. This knowledge can help you get a deeper insight into the development process on the Roslyn API, if you wish.

Programming Csharp Tutorial

Written by Unicorn Developer

753 Followers · 75 Following

Follow

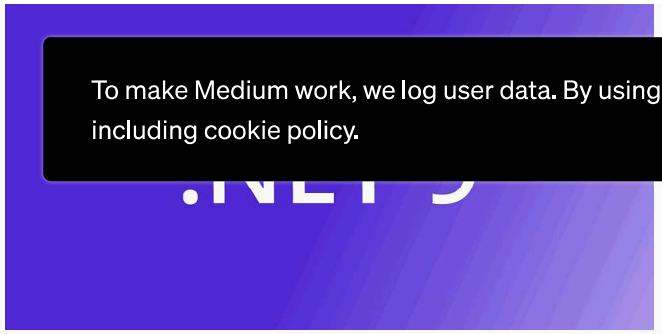
The developer, the debugger, the unicorn. I know all about static analysis and how to find bugs and errors in C, C++, C#, and Java source code.

No responses yet

Write a response

What are your thoughts?

More from Unicorn Developer

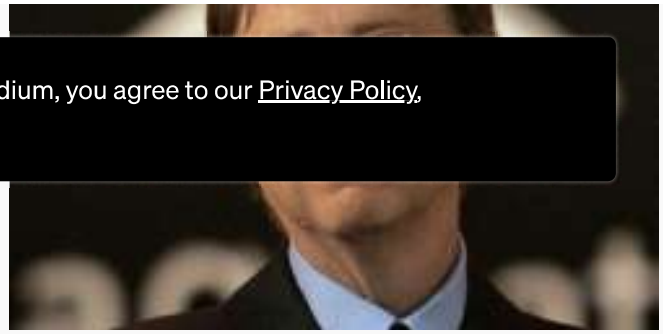


Unicorn Developer

What's new in .NET 9?

.NET 9 has been released, so it's time to start porting our projects to the new version! In th...

Nov 14, 2024 10



Unicorn Developer

How to capture a variable in C# and not to shoot yourself in the foot

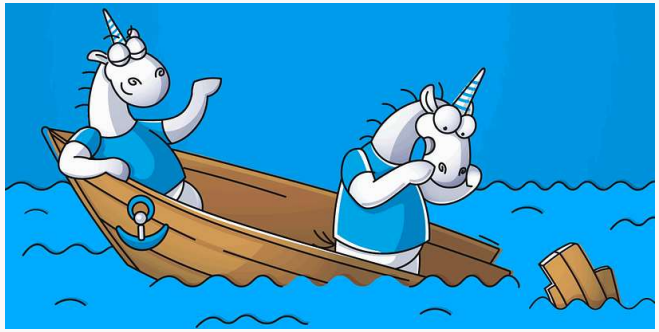
Back in 2005, with the release of C# 2.0 standard we got a possibility to pass a...

Jan 27, 2017 145 2

Unicorn Developer

What's new in Java 24

What's new in Java 24

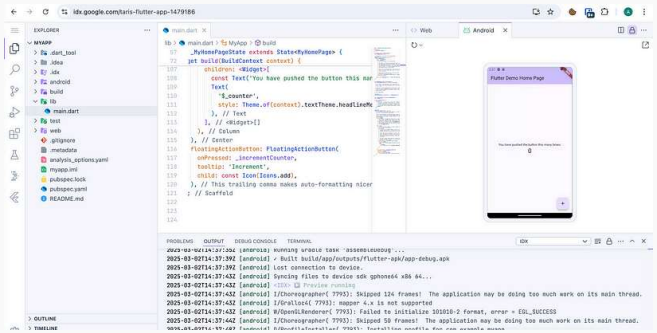


Unicorn Developer

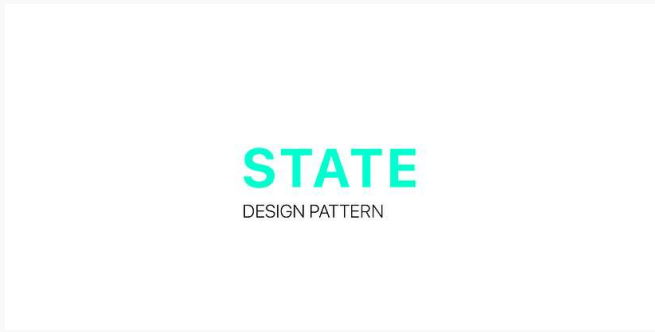
Safe array handling? Never heard of it

See all from Unicorn Developer

Recommended from Medium



In Coding Beauty by Tari Ibaba



Maxim Gorin

This new IDE from Google is an ab

Th
revolutionary.

Mar 11 4.8K 278

Stop Writing If-Else Trees: Use the

State pattern makes your code cleaner,...

Apr 10 1.3K 39

In Learn AI for Profit by Nipuna Maduranga

You Can Make Money With AI Without Quitting Your Job

I'm doing it, 2 hours a day

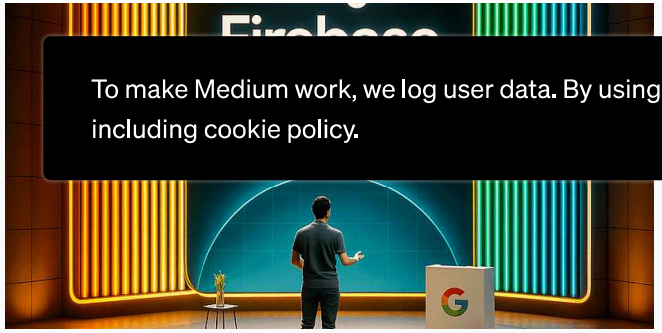
Mar 25 7.2K 329

In Let's Code Future by TheMindShift

10 AI Tools That Replace a Full Dev Team (Almost)

The future isn't coming—it's already here. And it's writing your code, fixing bugs, and...

Apr 6 1.96K 71



In AI Advances by Ashen Thilakarathna

Google's SECRET AI Just Killed Cursor! (Firebase Studio is...

WARNING: This FREE Google AI Will STEAL Your Coding Job!

Apr 14 1.5K 37

Devlink Tips

The end of Docker? The Reasons Behind Developers Changing The...

Docker once led the container revolution—but times have changed. Developers are...

Mar 21 1.7K 53

See more recommendations