C# 8 Ranges and Recursive Patterns

Key Takeaways

- C# 8 Adds Ranges and Recursive Patterns
- Ranges easily define a sequence of data, replacing the Enumberable.Range()
- Recursive Patterns brings an F#-like construct to C#
- Recursive Patterns is an awesome feature, it giving you the flexibility to testing the data against a sequence of conditions and performing further computations based on the condition met.
- Ranges is very useful to generate sequences of numbers in the form of a collection or a list.

Jan 21, 2015 is one of the most important days in the C# modern history. On this day, the C# Gurus like Anders Hejlsberg and Mads Torgersen and others have discussed the future of C# and considered in which directions the language should be expanded.

C# Design Meeting Notes for Jan 21, 2015[6]

The first result of this meeting was C# 7. The seventh version added some new features and brought a focus on data consumption, code simplification and performance. The new proposals for C# 8 do not change this focus on features, but this might yet change in the final release.

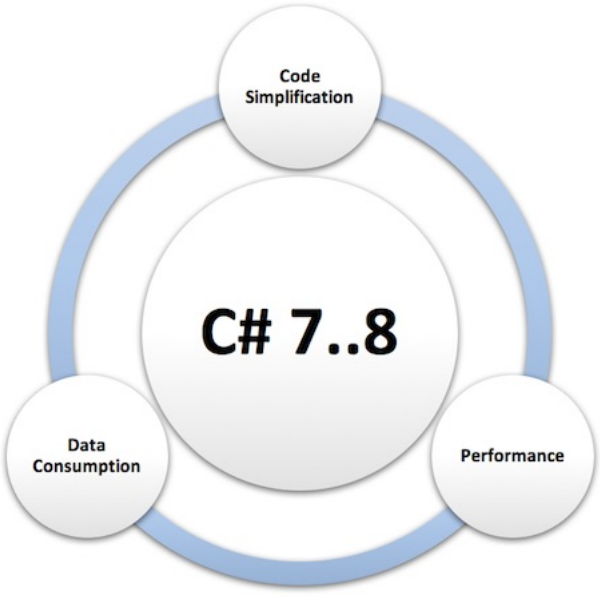

Figure -1- C# 7..8 features focus

In this article, I will discuss two features proposed for C# 8. The first one is the Ranges and the second one is Recursive Patterns, both of which belong to the category of Code Simplification. I will explain them in detail with many examples, and I will show you how those features can help you to write better code.

Ranges easily define a sequence of data. It is a replacement for `Enumerable.Range()` except it defines the startand stop points rather than start and count and it helps you to write more readable code.

Example

```
foreach item
```

```
 ConsoleWriteLineitem
```

Recursive Patterns matching is a very powerful feature, which allows code to be written more elegantly, mainly when used together with recursion. The RecursivePatterns consists of multiple sub-patterns like as Positional Patterns, e.g.,`var isBassam = user is Employee ("Bassam",_)`, Property Patterns, e.g.,`p is Employee {Name is "Mais"}`, Var Pattern, Discard Pattern (‘_’),and so forth.

Example

Recursive Patterns with tuples (The below example is also known as Tuple Pattern)

```
 employee  Name "Thomas Albrecht" Age
switch employee

 _  employeeTmp employeeTmpName  "Thomas Albrecht "

  ConsoleWriteLine$ "Hi {employeeTmp.Name} you are now 43!"

 break

// If the employee has any other information then executes the code below.
 _
 ConsoleWriteLine"any other person!"
 break
```

The `case (_, 43)` can be interpreted as follows. The first part, ‘ _‘,means ignore the item’s property Name but the second part, the Age, must be 43. If the employee tuple contains this pair of information (any string, 43),then the case block will be executed. As shown in figure -1-.

Playground for the employee code above:

Link 1, title: Playground for the employee code above

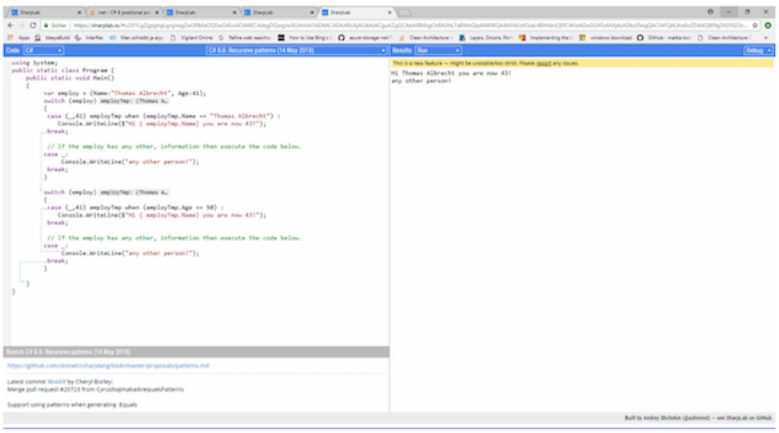

Figure -2- basicexample for Recursive Patterns

We have handled this theme in the past in more than one article, but this is the first time we are going to take a deep dive into pattern matching. Regarding to our old article, I recommend you read:

Ranges

This feature is about delivering two new operators (Index operator ‘^’ and range operator ‘..’) that allow constructing System.Index and System.Range objects and using them to index or slice collections at runtime. The new operators are syntactic sugar and making your code more clean and readable. The code for the operator index ^ is implemented in System.Index and for the range ‘..’ in System.Range.

System.Index

Excellent way toindex a collection from the ending.

Example

```
var lastItem = array[^1]; this code equivalent to: var lastItem = array[collection.Count-1];
```

System.Range

Ranges way to access "ranges" or "slices" of collections. This will help you to avoid LINQ and making your code compact and more readable. You can compare this feature with Ranges in F#.

New style	Old style

`var thirdItem = array [2];`
 	
`// Code behind: array [2]`
	`var thirdItem = array [2];`

`var lastItem = array [^1];`
 	
`// Code Behind: [^1] = new Index(1, true); true = bool fromEnd`
	`var lastItem = array [array.Count -1];`
	
	`var lastItem = array.Last; // LINQ`

`var subCollection = array[2..^5]; // Output: 2, 3, 4, 5`
 	
`// Code Behind: Range.Create(2, new Index(5, true));` as you see here we have used the both operators! Range and Index. The Range is for the operator … and the index is for the operator ^. Means skip until the index 2 from the begin and ^5 means ignore the first 5 elements from behind.
	`var subCollection = array.ToList().GetRange(2, 4);`
	
	or with LINQ
	
	`var subCollection = array.Skip(2).Take(4);`

Examples

Consider the following array:

```
var array = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Value	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

We can access the array values with the following indexes:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Now we cut a slice view from this array as below:

```
var slice= array[2..5];
```

Value	**2**	**3**	**4**

We can access the slice values with the following indexes:

Index	**0**	**1**	**2**

Note: the start index is inclusive (included to the slice), and the end index is exclusive (excluded from the slice).

```
slice1  array  // Range.Create(4, new Index(2, true))
```
and the slice1 will be of type Span<int>. [4..^2] Skip from the begin until the index 4 and skip 2 from the ending.

```
Output
 slice2  array  // Range.ToEnd(new Index(3, true))
Output
 slice3  array  // Range.FromStart(2)
Output
 slice4  array // array[Range.All]
Output
```

I have made the Ranges playground for you under the following link. Enjoy it!

Bounded Ranges

In the bounded ranges, the lower bound (start index) and the upper bound(end index) are known or predefined.

```
arraystartend // Get items from start-1 until end-1
arraystartendstep // Get items from start-1 until end-1 by step
```
The above Range syntax (step at end) is inspiredfrom Python. Python supports the following syntax(lower:upper:step), with :stepbeing optional and :1 by default, but there are some wishes in the community to use the F# syntax (lower..step..upper).

You can follow up the discussion here: Discussion: Range operator[7]

Range syntax in F#.

```
array { 5 .. 2 .. 20 } // where 2 = step [start .. step .. end]
```

Output:

```
5 7 9 11 13 15 17 19
```

Example for the bounded ranges

```
array
subarray  array  // The selected items: 3, 4
```
The code above is equal to `array.ToList().GetRange(3, 2);`.If you compare `array.ToList().GetRange(3, 2);` with `array[3..5]` you can see that the new style is cleanerand more human readable.

There is a feature request to use the Range in the “if” statement or with the pattern matching as described below:

With “in” operator

```
anyChar
anyChar

ConsoleWriteLine$"The letter {anyChar} in range!"



Output The letter b  range
```
Range Pattern is one of the new proposed pattern matching which can be used to produce simple range checks. Range Pattern will allow you to use the range operator ‘..’ in the select case" statement (switch).

```
switch anyChar

    ConsoleWriteLine$“The letter anyChar  range”
```

```
    ConsoleWriteLine$“Something ”



Output The letter b  range
```

It is worth to mention that not everyone happy by using the “in” operator in Ranges. The community is divided between using “in” or “is”; you can follow up the whole discussions here: Open issues for C#range[8]

Unbounded Ranges

When the lower bound is omitted, it's interpreted to be zeroor the upper bound is omitted, it's interpreted to be the length of the receiving collection.

Examples

```
arraystart  // Get items start-1 with the rest of the array
arrayend    // Get items from the beginning until end-1
array       // A Get the whole array
```

Positive Bounds

```
 fiveToEnd        // Equivalent to Range.From(5) i.e. missing upper bound
 startToTen       // Equivalent to Range.ToEnd(1). Missing lower bound. Result: 0, 1
 everything        // Equivalent to Range.All. Missing upper and lower bound. Result: 0..Int.Max
 everything   // Equivalent to Range.Create(5, 11)

 collection
collection  // Result chars: c
collection  // Result chars: a, b
collection   // Result chars: a, b, c
```

Negative Bounds

You can use negative bounds. They are interpreted as being relative to the length of the receiving collection, soone is the last element; two is the second to last elementand so on.

Examples

```
 collection
collection  // Result chars: b, c
collection   // Result chars: c
collection // Result chars: a, b
```

Note: Currently, the negative bound cannot be testedas shown below:

Figure-3- ArgumentException by using negative indexes

Ranges with Strings

Ranges allow creating substrings by using the indexer:

Example

```
 helloWorldStr  "Hello, World!"
 hello  helloWorldStr
ConsoleWriteLinehello // Output: Hello


 world  helloWorldStr
ConsoleWriteLineworld // Output: World
```

Or you can write it like that:

```
 world  helloWorldStr // Take the last 6 characters
ConsoleWriteLineworld // Output: World
```

Ranges ForEach loops

Examples

Ranges implement `IEnumerable<int>` which allowing the iteration over a sequence of data.

```
foreach  i


   ConsoleWriteLine“number i”
```

Recursive Patterns

Pattern matching is one of the powerful constructs, which is available in many functional programming languages like F#. Furthermore, pattern matching provides the ability to deconstruct matched objects, giving you access to parts of their data structures. C# offers a rich set of patterns that can be used for matching.

Pattern matching was initially planned for C# 7, but after while .Net team has find that he need more time to finish this feature. For this reason, they have divided the task in two main parts. BasicPattern Matching, which is already delivered with C # 7, and the AdvancedPatternsMatching for C# 8. We have seen in C# 7 Const Pattern, Type Pattern, Var Pattern and the Discard Pattern. In the next C# 8 version, we will see more patterns like Recursive Pattern, which consist of multiple sub-patterns like the Positional Pattern, and Property Pattern.

To understand the Recursive Patterns, we need many code examples. I have defined two classes. Employee and Company as defined below which I use them to explain the Recursive Patterns.

```
public class Employee

 public string Name




 public  Age




public Company Company




 public  Deconstruct string name   age  Company company

  name  Name
  age  Age
  company  Company



public class Company

 public string Name




 public string Website




 public string HeadOfficeAddress




 public  Deconstruct string name  string website  string headOfficeAddress

  name  Name
```

```
 website  Website
 headOfficeAddress  HeadOfficeAddress
```

Positional Pattern

Positional Pattern decomposes a matched type and performs further pattern matching on the values that are returned from it. The final value of this pattern/expression is true or false, which led to execute the code block or not.

```
 employee  Employee_ _ "Stratec" _ _ employeeTmp

 ConsoleWriteLine$ "The employee: {employeeTmp.Name}!"



Output
The employee Bassam Alugili
```

In this example, I have used the pattern matching recursively. The first part is Positional Pattern `employee is Employee(…)`and the second part is the sub-patterns within the brackets `(_,_, ("Stratec", _,_))`

The code block after the 'if' statement will only executed if the conditions in the root Positional Pattern(employee object must be of type Employee) with its sub-pattern `(_,_, ("Stratec",_,_)` the company name must be "Stratec") are satisfied, and the rest is discarded´, in other words, the if evaluates true only if the company name is Stratec.

Property Pattern

Property Patterns are straight forward. You can access a type fields and properties and apply a further pattern matching against them.

```
 bassam  Employee Name "Bassam Alugili" Age

 ConsoleWriteLine$ "The employee: {bassam.Name} , Age {bassam.Age}"
```

Old style C# 6:

```
 firstEmployeeGetType  typeofEmployee

  employee  Employee firstEmployee

  employeeName  "Bassam Alugili"  employeeAge

  ConsoleWriteLine$ "The employee: {employee.Name} , Age {employee.Age}"



// Or we can do it like this:

 employee  firstEmployee  Employee

 employee

  employeeName  "Bassam Alugili"  employeeAge

  ConsoleWriteLine$ "The employee: {employee.Name} , Age {employee.Age}"
```

Compare the pattern matching code with the C# 6 and look how the C# 8 code is more explicit. The new style removes the redundant code and the type casting or the ugly operators like “`typeof`” or “`as`”.

Recursive Patterns

RecursivePatterns are nothing more than a combination of the above-described patterns. The type will be decomposed to subparts so that the subparts may be matched against subpatterns. Behind the scene, this pattern deconstructs the type by using the Deconstruct()method and applying onthe deconstructed value further pattern matching if needed. If your type does not have a Deconstruct() method or it is not a tuple, then you need to write it by yourself.

If you remove the Deconstruct method from the Company class above, you will havethe following error:

error CS8129: No suitable Deconstruct instance or extension method was found for type ‘Company’, with 0 out parameters and a void return type.

Let's get a look atthe Positional Pattern and Property Pattern.

Example

I have created two employees and two companies and map each employee to a company.

```
stratec   Company

 Name  "Stratec"
 Website  "wwww.stratec.com"
 HeadOfficeAddress  "Birkenfeld"


firstEmployee   Employee

 Name  "Bassam Alugili"
 Age
 Company  stratec


microsoft   Company

 Name  "Microsoft"
 Website  "www.microsoft.com"
 HeadOfficeAddress  "Redmond, Washington"


secondEmployee   Employee

 Name  "Satya Nadella"
 Age
 Company  microsoft


WriteBioMedicalCompanyNamesstratec // Stratec is a biomedical company!

DumpEmployfirstEmployee
DumpEmploysecondEmployee

public static  DumpEmployeeEmployee employee
```

```
switch employee
  Employee_ _ _ employeeTmp

   ConsoleWriteLine$ "The employee: {employeeTmp.Name}! "

  break

 default
  ConsoleWriteLine"Other company!"
  break




Output
The employee Bassam Alugili
The employee Satya Nadella
```

In the example above the case condition matching any employee with any data, it is a combination of the Deconstruction Pattern and the Discard Pattern. Now we will go one step further. We need only to filter the Stratec employees.

There is more than one approach to do that with pattern matching. We will replace/rewrite the below case condition from the example with some different techniques.

```
Employee_ _ _ employeeTmp


  ConsoleWriteLine$ "The employee: {employeeTmp.Name}! "

break
```

The first approach, by using the Recursive Patterns matching (Deconstruction Pattern) in the switch statement like the following.

Replace the above code with the code below.

```
Employee_ _ "Stratec" _ _ employeeTmp


 ConsoleWriteLine$ "The employee: {employeeTmp.Name}! "

break
```

Output:

```
The employee  Bassam Alugili
Other company
```

The second approach by using guards (Constraints).

Replace the case code with following.

```
Employee_ _ _ _ _ employeeTmp when employeeTmpCompanyName  "Stratec"


 ConsoleWriteLine$ "The employee: {employeeTmp.Name}! "

break
```

Likewise, we can rewrite the case expression in different ways:

```
Employee_ __ employeeTmp when employeeTmpCompanyName  "Stratec"
Employee employeeTmp when employeeTmpCompanyName  "Stratec"
```

We can also combine Deconstruction Pattern with Var Pattern like the following:

```
Employee_ _ _companyNameTmp_ employeeTmp when companyNameTmp  "Stratec"
```

Another approach to filter the data by using Property Pattern recursively as shown below:

```
 Employee CompanyCompanyName"Stratec" employeeTmp
Output  the above examples
The employee  Bassam Alugili
Other company
```

One important note I want to mention about using the switch statement with the pattern matching:

The anatomy of the new switch expression looks like this:

```
 switch value

      pattern guard  Code block to be executed

      _  default
```

Back to our example, look to the following Recursive Patterns matching examples:

```
switch employee

  Employee Name "Bassam Alugili" Company Company_ _ _ employeeTmp

      ConsoleWriteLine$ "The employee:  {employeeTmp.Name}! 1"

  break

  Employee_ _ "Stratec" _ _ employeeTmp

      ConsoleWriteLine$ "The employee:  {employeeTmp.Name}! 2"

  break

  Employee_ _ Company_ _ _ employeeTmp

      ConsoleWriteLine$ "The employee:  {employeeTmp.Name}! 3"

  break

  Employee_ _ _ employeeTmp

      ConsoleWriteLine$ "The employee:  {employeeTmp.Name}! 4"

  break

 default
  ConsoleWriteLine"Other company!"
  break
```

The above switch is working fine. If we move one of those switch case somewhere else up/down.Let's say you will movecase Employee(_, _, _) employeeTmp:at the beginning (first case after the switch) as shown below:

```
switch employee

  Employee___ employeeTmp
```

```
  ConsoleWriteLine$ "The employee: {employeeTmp.Name}! 4"
```

Then we will get the following errors:

1. **error** CS8120: The switch case has already been handled by a previous case.
2. **error** CS8120: The switch case has already been handled by a previous case.
3. **error** CS8120: The switch case has already been handled by a previous case

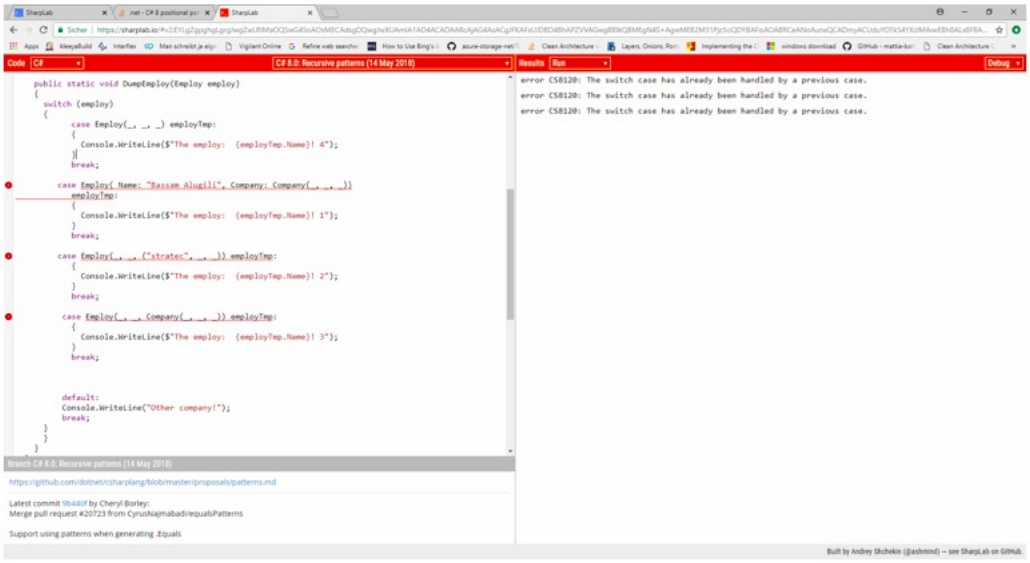

Figure -4- Error after changing the switch case position in SharpLab

The compiler knows that the cases after your most generic switch case cannot be reached (dead code) and telling you with the error that you are doing something wrong.

Pattern matching with Collections

Examples

```
switch intCollection

    x

  // This block code will executed and return if the first two items in intCollection is 1 and 2 and the third one will be copied into th
  ConsoleWriteLine $ "it's 1, 2, {x}"



    // This block will be executed if the intCollection starts with 1 and ends with 20.


  _

    // This block will be executed for any other use cases except the tow above defined.



intCollection
```

```
   // Execute this block only if the lasts items in the collection are 99 and 100


intCollection

   // Execute this block only if the first items in the collection are 1 and 2


intCollection

   // Execute this block only if the first items in the collection is 1 and the last one 100
```

Recursive Patterns (C# 8) Playground

1. Copy the below employee code example
2. Open in the web browser: https://sharplab.io
3. Paste the code and select “C# 8.0: RecusivePatterns (14 May 2018)”, then select the “Run” as shown in figure4 below.

Alternatively, You can use my following link, which I have prepared.

Code:

```
using System
namespace RecursivePatternsDemo

class Program

 static  string args

   stratec   Company

    Name  "Stratec"
    Website  "wwww.stratec.com"
    HeadOfficeAddress  "Birkenfeld"


   firstEmployee   Employee

   Name  "Bassam Alugili"
   Age
   Company  stratec


   microsoft   Company

    Name  "Microsoft"
    Website  "www.microsoft.com"
    HeadOfficeAddress  "Redmond, Washington"


   secondEmployee   Employee

   Name  "Satya Nadella"
   Age
```

```
    Company  microsoft


   DumpEmployeefirstEmployee
   DumpEmployeesecondEmployee


  public static  DumpEmployeeEmployee employee

   switch employee

      Employee Name "Bassam Alugili" Company Company_ _ _ employeeTmp

      ConsoleWriteLine$"The employee:  {employeeTmp.Name}! 1"

     break

     Employee_ _ "Stratec" _ _ employeeTmp

      ConsoleWriteLine$"The employee:  {employeeTmp.Name}! 2"

     break

     Employee_ _ Company_ _ _ employeeTmp

      ConsoleWriteLine$"The employee:  {employeeTmp.Name}! 3"

     break

    default
     ConsoleWriteLine"Other company!"
     break




public class Company

 public string Name




 public string Website




 public string HeadOfficeAddress
```

```
 public  Deconstruct string name  string website  string headOfficeAddress

  name  Name
  website  Website
  headOfficeAddress  HeadOfficeAddress



public class Employee

 public string Name




 public  Age




 public Company Company




 public  Deconstruct string name   age  Company company

  name  Name
  age  Age
  company  Company
```

Figure -5- SharpLab settings

Summary

Ranges is very useful to generate sequences of numbers in the form of a collection or a list. Combining Ranges with for each loops or and Pattern matching, etc., makes the C# syntax more simple and readable.

Recursive Patterns is the core of the Pattern matching. Pattern matching helps you to compare the runtime data with any data structure and decompose it into constituent parts or extract sub data from data in different ways and the compiler supporting you to check the logic the of your code.

Recursive Patterns is an awesome feature, it giving you the flexibility to testing the data against a sequence of conditions and performing further computations based on the condition met.

About the Author

 **Bassam Alugili** is Senior Software Specialist and databases expert at STRATEC AG. STRATEC is a world-leading partner for fully automated analyzer systems, software for laboratory data management, and smart consumables.

This content is in the .NET[9] topic

Related Topics:

- **Development**[10] Development Follow[11] 586 Followers
- Follow[12] 0 Followers
- Follow[13] 357 Followers
- Microsoft[14] Microsoft Follow[15] 15 Followers

Tell us what you think

Email me replies to any of my messages in this thread

Typos 2 hours ago by Jonas Widarsson [16]

Thank you for the article, very interesting!

It contains a lot of typos, however - this being the least coherent part:

```
WriteBioMedicalCompanyNames(stratec); // Stratec is a biomedical company!


DumpEmploy(firstEmployee);
DumpEmploy(secondEmployee);


public static void DumpEmployee(Employee employee)
```

WriteBioMedicalCompanyNames does not seem to be covered by the article and **DumpEmploy** should be **DumpEmployee**.

There are a number of other less significant errors in the article as well.

Links

1. javascript:void(0);
2. https://www.infoq.com/profile/Bassam-Alugili
3. javascript:;

4. https://www.infoq.com/profile/Jeff-Martin
5. javascript:;
6. https://github.com/dotnet/roslyn/blob/master/docs/designNotes/2015-01-21%20C%23%20Design%20Meeting.md
7. https://github.com/dotnet/csharplang/issues/198
8. https://github.com/dotnet/roslyn/issues/23205
9. https://www.infoq.com/dotnet
10. https://www.infoq.com/development
11. javascript:;
12. javascript:;
13. javascript:;
14. https://www.infoq.com/Microsoft
15. javascript:;
16. https://www.infoq.com/profile/Jonas-Widarsson.1