DOTNEXT

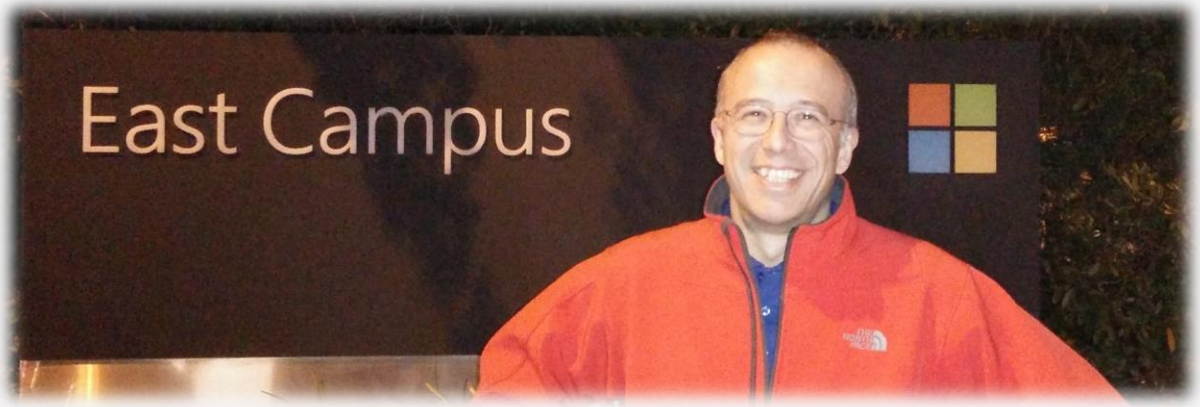
Write your own C# static code analysis tool to drive business rules

Raffaele Rialdi
Senior Software Architect
Microsoft MVP

@raffaeler
https://github.com/raffaeler
http://iamraf.net

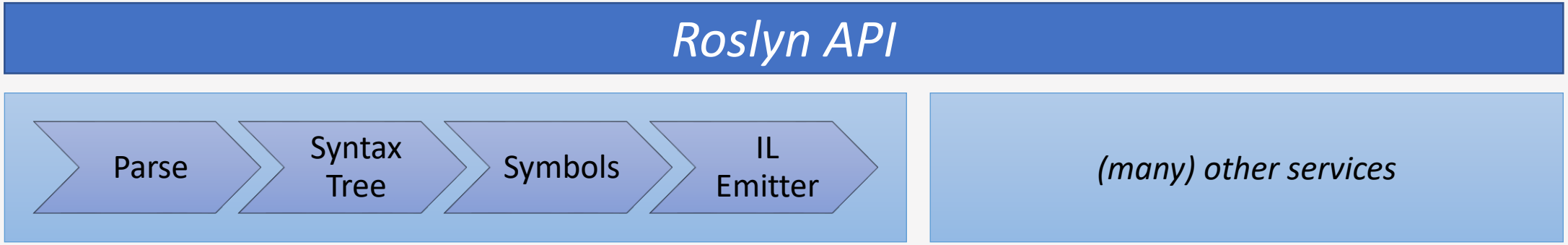
Who am I?

- Raffaele Rialdi, Senior Software Architect in Vevy Europe – Italy
  @raffaeler also known as "Raf"
- Consultant in many industries
  Manufacturing, racing, healthcare, financial, …
- Speaker and Trainer around the globe (development and security)
  Italy, Romania, Bulgaria, Russia (CodeFest @ Novosibirsk), USA, …
- And proud member of the great Microsoft MVP family since 2003



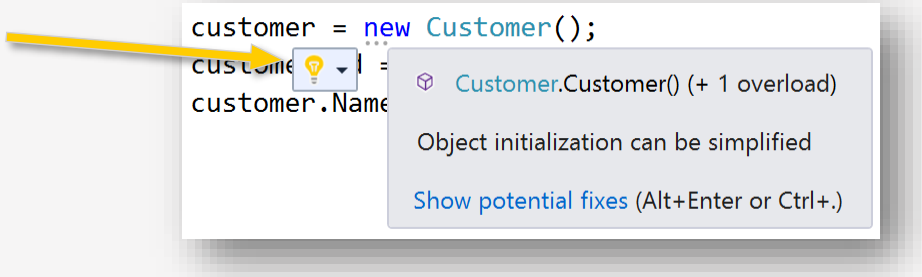
CodeAnalysis

- Is the process of analyzing the source code without running code
- The Roslyn Compiler provides APIs to **read** and extract information:
  - The Abstract Syntax Tree provides the lexical structure and the graph of **all** the possible execution paths
  - The Semantic Model enriches the AST by applying language rules and providing a better understanding of the nodes (types, properties, methods, …)



We already use CodeAnalysis …

- During development in the IDE
  - Intellisense, code completion and refactoring
  - Microsoft and third parties Analyzers
    - Suggest changes, reveal errors and fix the code
    - Naming conventions, language features, …



- On the build servers
  - Enforcing «StyleCop» or other similar tools
    - Code-quality measurement
    - Banned APIs
    - Documentation correctness
    - …

Leveraging CodeAnalysis to add our own Business Rules

- An inspirational example from C# 8.0: *Nullable Reference Types*
  - Code Analysis will trigger a message for null on reference types
  - The compiler will ask the user to express the will to use a reference type with or without nulls
  - But reference types will not change from a CLR perspective
  - The feature is similar to detecting an uninitialized variable

```
string hello;    // field in a class
// ...
var size = hello.Length; // warning!
```

- Why not using the Compiler to enforce our own rules?

The serialization example

- We want the entity Customer to be constructed with:

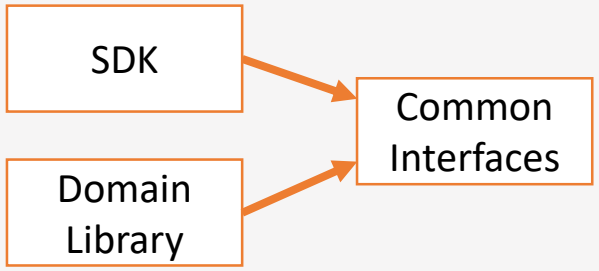
```
public Customer(int id, string name) { ... }
```

- But if the default constructor is private, Json.NET will complain
  - The solution is defining a custom converter or the *JsonConstructor* attribute
  - This translates in: additional code or undesired dependency

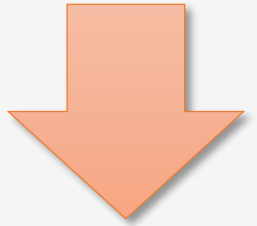
- What about leaving the default constructor public and firing a warning or error when using the default constructor?

The interface versioning example

- A new SDK can send commands or publish events
- Another assembly defines ICommand and IEvent




```
public interface IBusManager
{
    bool Publish(IEvent event);
    bool Send(ICommand command);
}
```



- What about getting rid of the interfaces?
  - object is not a good choice, BUT …
  - we can use CodeAnalysis to enforce the types

```
public interface IBusManager
{
    bool Publish(object event);
    bool Send(object command);
}
```

What we did until now

- We let the IDE walk the graph
  - As soon as the given IOperation is verified, our callback is invoked
  - Analyzer class validate the code
  - CodeFix class, if any, fix the bug

- BUT
  - Analyzers cannot access the workspace or the entire solution
  - Even if they are asynchronous, it can take time to make complex analysis

- Writing a custom tool
  - Same APIs but stand-alone tool (can be used on build servers)

Reading the solution (.sln) with Roslyn

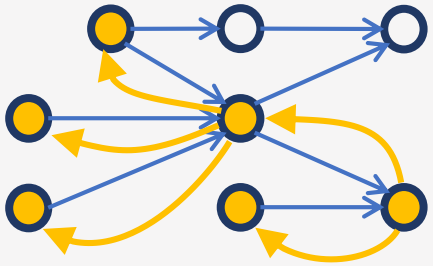
- A Console app referencing Microsoft.CodeAnalysis nuget packages
  - Loading the solution using Microsoft.Build.Locator (by msbuild)
  - Compiling the solution to ensure there are no errors (by Roslyn)
  - Processing the syntax and semantic data (our tool)

- What we will see now:
  - Extract all the members from all the types defined in the solution
  - Walking the graph upwards and downwards

Walking the graph

From Method Definitions to Member Callers

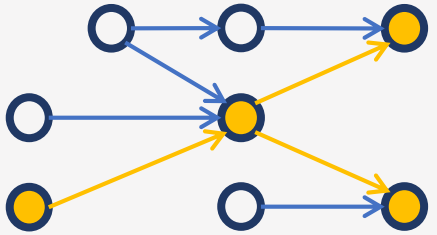
```
var refs = SymbolFinder.FindCallersAsync(memberSymbol, _context.Solution).Result;
foreach (var referenced in refs)
{
    foreach (var definition in referenced.CallingSymbol.DeclaringSyntaxReferences)
    {
        var callerDeclarationSyntax = definition.GetSyntax();
        Visit((MemberDeclarationSyntax)callerDeclarationSyntax); // recurse
    }
}
```



From Method Invocations to Member Definitions

```
var semanticModel = _context.GetSemanticModelFor(invocationExpressionSyntax);
var method = semanticModel.GetSymbolInfo(invocationExpressionSyntax).Symbol;

foreach (var declaringSyntax in method.DeclaringSyntaxReferences)
{
    var declarationSyntax = (MethodDeclarationSyntax)declaringSyntax.GetSyntax();
    StartInternal(declarationSyntax);
}
```



Security Check Example

- When compiling code on the fly (provided by user)
  - Security checks are mandatory

- Load and compile the code, then walk the syntax nodes
  - Visit all the invocations
  - Accept only the ones that are whitelisted

```
private static IList<ISecurityRule> GetBlackWhitelist()
{
    var list = new List<ISecurityRule>();
    list.Add(new SecurityRuleByNamespace(true, "System"));
    list.Add(new SecurityRuleByNamespace(true, "System.Collections"));
    list.Add(new SecurityRuleByNamespace(true, "System.Collections.Generic"));
    list.Add(new SecurityRuleByType(true, typeof(System.IO.DirectoryInfo)));
    list.Add(new SecurityRuleByType(false, typeof(System.Activator)));
    return list;
}
```

But what about *business logic*?

- Business logic is a set of rules imposing constraints, actions and data transformation to *govern the business behavior*.
- Examples:
  - Never apply twice a discount
  - Agent rebates must always occur after the discount (if any)

- Can we use CodeAnalysis to enforce these validations?

  (Of course Yes ☺)

What we have seen

1. Get all the solution declared methods
2. Walk the graph to the top declarations
3. Walk the graph down (all the paths) looking for method invocations on any object implementing IBizRule
4. Validate all the possible sequences are correct

But there is a flaw! … a bug on the logic of the tool

- ***Rules validation only makes sense if applied on the same instance!***

IT COULD WORK!

Statically tracking objects identity

- Visit maintaining a stack of dictionary<varName, identity>
  - When visiting the **Assignment** and **Declaration** nodes
    - Add to the dictionary the new variable with its identity (new or copied from right identifier)

```
Order order = o;
```

  - When visiting an **Invocation** to a method
    - Creates a new dictionary for in the stack and copy the variables passed as parameters
    - Variables are renamed according to the parameter name of the declaration

```
p.ProcessOrder(order);
```

```
public void ProcessOrder(Order o)
```

- The demo ignore other important syntax nodes
  - "out", "ref", constructors, properties, etc.

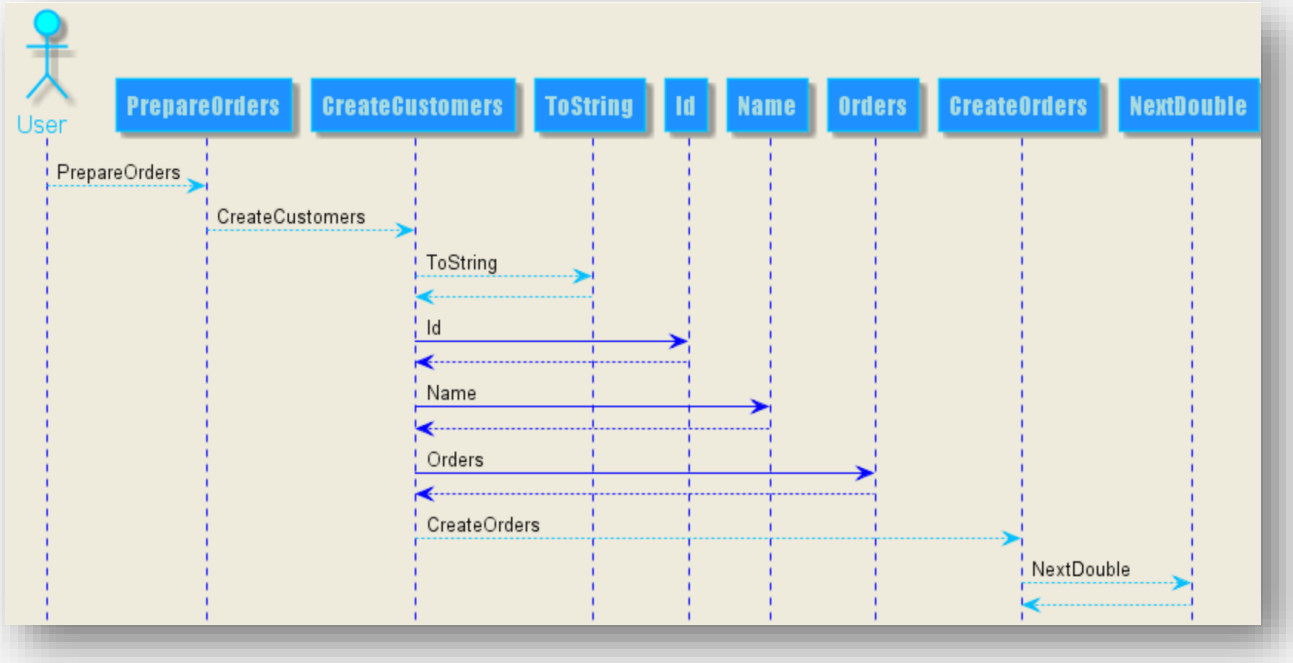
Runtime vs CodeAnalysis rule validations

- CodeAnalysis is broader then any runtime tests
  - Analyze all the possible invocation paths (even the "impossible by logic" ones)
- CodeAnalysis is NOT a replacement for tests!
  - May result in false positives
  - Loops may result in multiple invocations
  - The demo omits the analysis of properties, constructors and delegates
- CodeAnalysis cannot replace any validation
  - Runtime environment is totally different!!!

There is more!

Documentation is an example

1. Walk the graph
2. Capture relevant info
3. Draw a dependency diagram



To sum up

- The static flow is different from execution flow
  - It is the largest possible graph
- Pros
  - Call graph / sequences
  - Dependencies
  - Whitelisting / Blacklisting method calls
- Cons
  - It may result in false positives
  - Object identities are difficult to track
  - Operations requiring runtime execution cannot be evaluated

Questions @ booth #1

```
var semanticModel = _context.GetSemanticModelFor(
    memberDeclarationSyntax);

var memberSymbol = semanticModel.
    GetDeclaredSymbol(memberDeclarationSyntax);
```

Demos at:
https://github.com/raffaeler/dotnext2018Piter

Thank you!