ORGANIZER

COMPILE TIME SOURCE CODE GENERATOR

PREPARED BY
MOHAMAD FARAJ FIRAS MAKKAWI
MOHAMMAD SALEH AMIRI

PREPARING YEAR
2023 – 2024

Gift

Mohamad Makkawi

For The Sake Of Those Who Are In My Heart.

Mohammad Amiri

I Gift This Work To My Family And My Friends
And Everyone Loves Me.

ABSTRACT

This project is in the field of code generation, exactly in compile-time source code generation (Source Generator), It generates source code files during compile time.

This project presents a Source Code Generator created with Roslyn Source Generator (SG) for .NET developers. the SG presented is called Organizer.

The Organizer meant to organize any C# unorganized code as requested by the client, by restructuring the **base types** (classes, interfaces, records, enumerations, structs) in different folders and files according to the needs of the client using a set of services.

The meaning of unorganized code in the scope of The Organizer is C# code files full of base types.

The services provided by The Organizer are:

- A service to create folder(s) to contain generated files.
- A service to include base type(s) in a specific generated folder depending on a type name or a pattern.
- A service to change the name(s) of specific type(s) depending on a base type name or pattern of multiple base types.
- A service to ignore type(s) depending on a base type name or a pattern of multiple types.
- Ability to exclude specific types from the creation or update patterns depending on a type name.

The Organizer was developed using .NET Compiler (Roslyn), .NET Standard 2.0 for SG development. And .NET 7.0, xUnit were used for unit testing.

The result of this project is a tool called Organizer, that could be used by the client to organize the structure of their source code or a part of it in compile time. This will result in more usable source code as it will be easier to surf and access and read.

The Organizer is supported on every IDE as long as it supports C#, like VS code, Visual Studio, Rider.

Keywords: Generator, Source Code, Compile Time, Syntax, Services.

Contents

Table Of Figures

TERMS

API: Application Programming Interface.

SG: Source Generator.

IDE: Integrated Development Environment.

ST: Syntax Tree.

JIT: Just in Time.

CodeDOM: Code Document Object Model.

MVC: Model View Controller.

T4: Text Template Transformation Toolkit.

Chapter 1: General Project Framework

1.1 Introduction

Sometimes when during your workflow as a coder, you come across code files that contains a lot of base types[1] like classes, which are related, or unrelated, or contains even enumerations (Enums), these kinds of files are structured in a bad manner, code files must not be crowded with a lot of types. because reading and surfing these files will be hard because of the large number of lines and a lot of unrelated classes.

Wouldn't it be better to surf the same classes if they were separated into multiple files with each file containing only related base types? One solution is to cut and paste all related base types from the badly structured file into other file, this will group the related classes and make them more accessible. Another problem might be that you want to have an identical prefix or suffix to these related base types, the solution might be renaming each class including the suffix or the prefix you prefer. This process is time consuming.

This documentation an implementation of a tool that automates this process. This tool will be based on C#. It will generate **C#** source code files structured in developer-built architecture [2][3] from an unorganized -crowded and full of unrelated code- source code files to organize the code and make it more usable and readable.

The tool contains other features such as ignoring specific base types, or even renaming related base types with identical suffixes or prefixes if the developer (client) wanted.

This section will discuss the problem of the project and the solution provided by the tool. It will also discuss the tools used during the development and documentation writing. It will also discuss background studies.

[1] Base Type: in .NET. classes, enumerations, structs, records, interfaces are considered Base Types and derived from a class called BaseTypeDeclarationSyntax [10].
[2] Architecture: an architecture in software is the high-level design decision of a software system [11]. High level design refers to the structure of the software folders and files in this documentation use case.
[3] Developer-built architecture: this term refers to a custom architecture of the system or a part of the system developed by the developer of the client software.

1.2 Project Problem

The problem of this project lies with the fact that as the code gets bigger in a bad structure in a single file or multiple files as it becomes harder to read and use. Imagine reading a file similar to the file in (figure 1-1), it will be a hard task to use such a code because of bad organization, the base types are jammed into a single file. Many unrelated types are presented in the same file. And if the project contains multiple files structured in a similar manner, it will become a very hard task to follow the code and understand it.

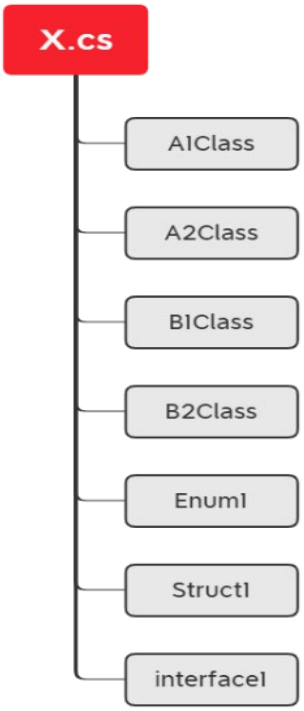

Figure 1- 1 Bad Code Architecture

The simplest architectural solution is to see which types are related. It is obvious that base types A1-A2 are related, and base types B1-B2 are related. But base types A and B are unrelated. Also, the struct and the enumeration and the interface are out of context. Thus, it is nonprofessional for these base types to be in the same file. It will be better to have the structure in (Figure 1-2)

Although this is not the best architectural approach, it is better than the architecture in (figure 1-2). This kind of problems is obvious when working with web services API[4] Interfacing[5] services.

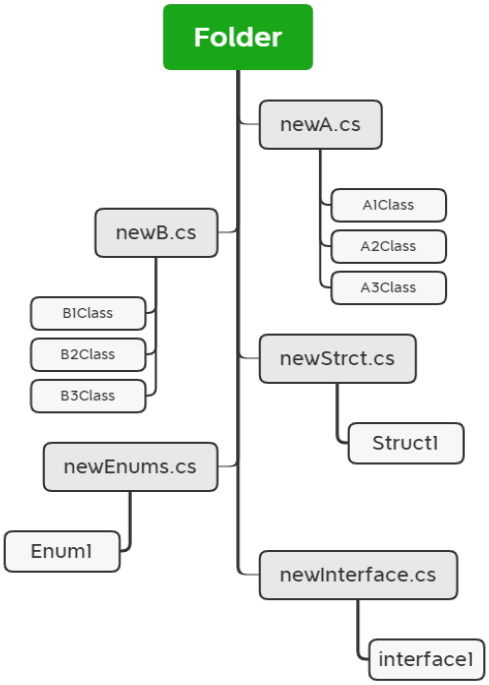

Figure 1- 2 Better Architecture for Figure 1-1

[4] Application Programming Interface: it is a software intermediary that allows two software apps to communicate with each other [12].
[5] API Interface: A way to describe and understand and explore an API functionalities and capabilities without diving into its source code.

1.2.1 Web Services Communication

When building web services, it is important to provide a well-structured API for developers to interact with your services.

The developers interact with a web service through an API provided by the builder of the service. It works in a request-response pattern. Where the request in the client side is structured with a data serialization language like JSON or YAML or markup language like XML. Then sent to the API. Then the logic happens in the server side and the return is response, which is also a structured data that is serialized in the client side and interpreted to what the client expects.

A simple example to illustrate this workflow is search by username workflow. The client inputs the username he wants, then sends a Get User request. The server is expected to return either a valid user or a not valid user or anything else as a response. (Figure 1- 3)

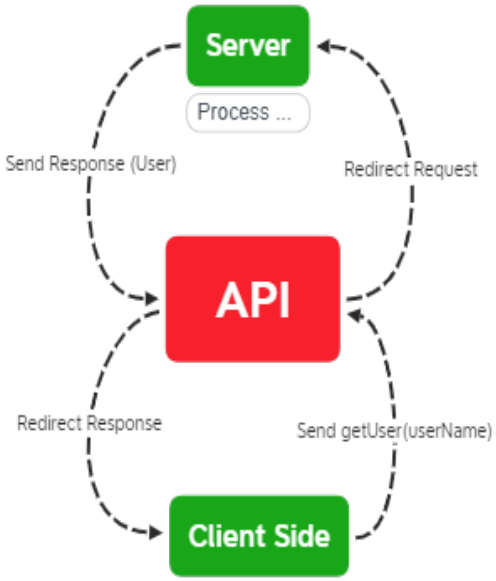

Figure 1- 3 Simple API Work flow

1.2.1.1 Open API – Swagger

Open API is a company that provides specification standards to describe web HTTP services APIs. It works as an interface for any API that uses it. It is used to discover and understand the capabilities of an API without interacting with its source code [1]. The name of Open API specifications is swagger.

Some API interfacing services like Swagger, provide a tool to generate all models[6] into a single file as base types that could easily be enormous in lines number. Using this file is disastrous. It will be very hard to surf through it and locate the specific model because of the large line number. This report will take Swagger generated models code as a use case example problem solved by the tool discussed in this documentation.

There are a lot of details concerning swagger and its tools, this report is only interested in swagger code generation tool.

[6] Model: this term refers to every data class and request and response.

1.2.1.2 Swagger Code Generation

Usually, the API that uses Swagger can provide a JSON file that describes the whole API as a JSON. This JSON contains all possible model types and more. Swagger also provide tools to generate a programming language file from this API JSON file, there are various languages to choose from, for example C# programming language. This generated file will contain a reflection of the JSON in C# language as types, where each model will be a base type.

The problem here is all models will be in the same file (Figure 1-4). Thus, it is causing a hard usage of code (hindrance in code surfing, reading, reaching specific types).

The solution to this issue will be to cut each request class into its own file and group all requests into requests folder and do the same to responses and models (Figure 1-5).

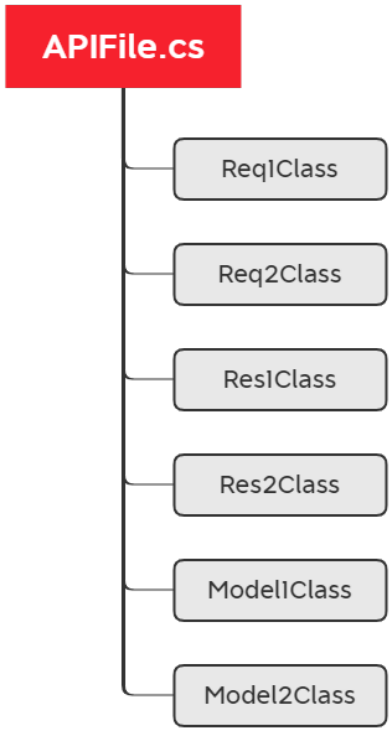

Figure 1- 4 The Structure of Swagger API Generated C# File

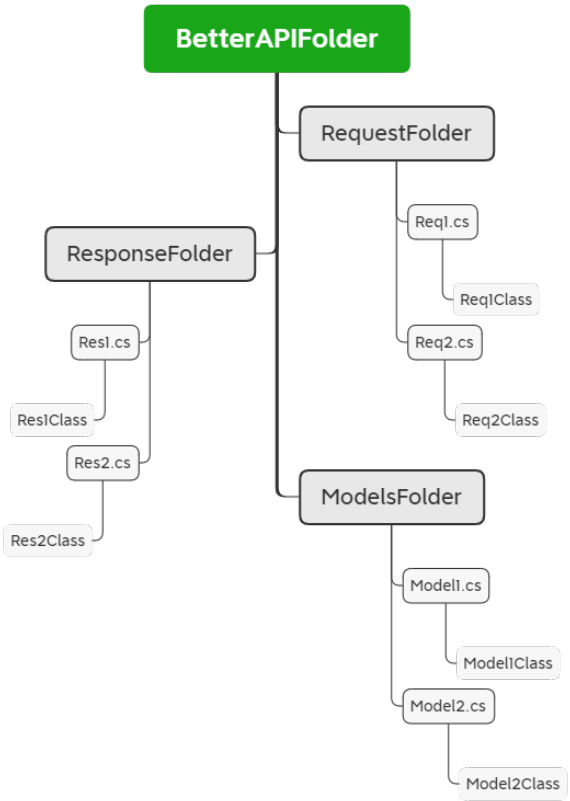

Figure 1- 5 Better Structure for API

The tool developed and documented in documentation will take responsibility and automate the process of generating the organized folders and files and types from any C# source code provided by the client. The swagger generated file use case example will provide a real-world example of the usefulness of such a tool.

1.3 Project Purpose and importance

The purpose of this project is to assists .NET developers when using unorganized code by creating a Source Generator[7](SG). This source generator will take unorganized C# source code as an input, and outputs organized source code files structured in a developer-built architecture. The SG this document is presenting is called (Organizer). And it is a **compile time source code generator**[8]. The Organizer SG will be published as a *NuGet package.*

This SG provides a set of services for the developer to organize a C# code of their choice through generating .cs files from the .cs files provided. The way the files are structured is governed by a developer-built architecture.

1.4 Organizer Source Generator Features

There are multiple features provided to organize code:

- This SG supports anything considered a base type: Classes, Enumerations, Structs, Records Interfaces.
- Giving multiple source code files as an input to the tool.
- Ability to choose the generated source code files paths independently.
- The ability to create folders and files contains the output base types depending on their names or on a pattern.
- The ability to ignore specific types depending on their names or on a pattern.
- The ability to rename types depending on their names or on a pattern.

1.5 Organizer Source Generator Development Language and Tools

This section will discuss the tools used during the development process and how they were used and how they work. The Organizer (SG) is Developed in **C# language**, used .NET Compiler, .NET7 and .NET Standard 2.0 specification. .NET compiler was used to analyze the C# code to extract required data. It is also used for generating C# code. The name of .NET compiler is **Roslyn**. The Organizer used xUnit for unit testing. The SG build based on .NET Standard 2.0.

[7] Source Generation: Source Generation is a technology to generate source code files. This action could be done in compilation time, or in run time.
[8] Compile Time Source Code Generation: is a technology to generate source code files during compile time.

1.5.1 Tools

1) Roslyn (.NET Compiler)

Roslyn is the name of the compiler used in .NET frameworks and languages (C#, Visual Basic). It is delivered to the developers for use as an API in the namespace[9] (Microsoft.CodeAnalysis. *). The source generator tool is shipped as a part of Roslyn.

Understanding how all Roslyn components work is not a part of this report. This documentation will discuss only Roslyn components used during development. This project only used part of the syntax component and some of source generator components

- Syntax Component (Microsoft.CodeAnalysis.CSharp. Syntax)

This component is a Roslyn package that represents the various syntax components of the C# language, this package works with the C# sentence model (part of syntax model). It was used in the scope of the Organizer project to identify the following components:

- **BlockSyntax**: the object of this class contains multiple information include everything from open to a close bracket {…} is considered a block syntax in C# language. in the scope of this project, this syntax kind is used by the Organizer services to identify the scope of each service used by the developer. E.g.

```
CreateFolder("FolderName1");
{
    CreateFolder("FolderName2");
}
```

It is obvious that the folder “FolderName2” is nested in folder “FolderName1”, this type of situation is detected by checking the scope of FolderName1.

- **Base Type Declaration Syntax**: This class represents every syntax element declared as Base Type (classes, structs, enumerations, records, interfaces). used to determine which elements in the file to look for. The elements that are targeted by the Organizer services are Base Types.

- **Class Declaration Syntax**: This class is assigned to all declared classes in a C# Code. It is used to identify the Organizer class in the client code, this class is the class used to execute the Organizer services at the system side.

- **Constructor Declaration Syntax**: This class is assigned to the constructors. And it is used in the scope of this project to identify the constructor in the Organizer class in the client side.

[9] Namespace: Is the C# way to organize code units in hierarchical structure. Where the name of the namespace represents the root of all the components beneath it. the components could be: Namespaces, Base Types, Functions, any C# component.

- **Invocation Expression Syntax**: This class represents every invocation (which is a function call). It is the composition of the caller object if existed and the function name with parenthesis and everything in between but without the semicolon.

  E.g.: `Obj.Func(param1, param2); // the whole expression is an invocation expression but without the semi colon`

  Also, any function call is an invocation expression. E.g. `Func(param1, param2);`

  This type is used to identify the calls of the Organizer services in the client side.

- **Attribute Syntax**: This type of syntax represents attributes[10] in C#, in other languages like java, dart, they are called Annotations. Attributes are used to trigger some kind of behavior of action by the compiler. In the case of The Organizer project, attributes were used to distinct the implementer SG class in the system side with the attribute [Generator]. Attributes were also used to determine the path(s) of unorganized code (input) and path of organized code (output) folder. [From: "Path/To/Files" , To: "Path/To/Folder"].

  - Source Generator Component (Microsoft.CodeAnalysis)

  The Source Generator is a component shipped with **Roslyn SDK**. This component is an interface called "**ISourceGenerator**". It represents an interface that contain two methods:

  ```
  - void Initialize (GeneratorInitializationContext context):
  ```

  This method represents a callback action that will be executed before generation occurs. The callback could be as simple as a console date logging action to a .txt file to inform the client (user) that a code generation was about to happen in a specific moment. The parameter (context) used to add information to the context used during code generation, and used to inform the generator if the initialization has failed for a reason. It was not used in the case of The Organizer project.

  ```
  - void Execute (GeneratorExecutionContext context):
  ```

  This method will perform the code generation action. The parameter (context) represents the compilation object[11] alongside other properties like the information from **Initialize** function and information about the parsed code. only the compilation object was used in The **Organizer** to access the whole source code from the top of the project as a syntax trees. Where each syntax tree represents a namespace.

  - Syntax Tree Component (Microsoft.CodeAnalysis)

  Syntax Tree (ST) is a tree like structure representation of a parsed code syntax. The ST use is inevitable in **The Organizer SG**, it was used to access the C# source code components as objects -such as ClassDecalarationSyntax, BaseTypeDecalarationSyntax, InvocationExpressionSyntax-.

[10] Attribute: An attribute represents a meta data, which is a data about data.
[11] Compilation Object: An object that contains the client code of the current project with its meta data.

The only part of the compilation cycle was required in **The Organizer** SG is the Syntactic Analyzer because the SG requires a ST in order to work.

2) Frameworks

The Organizer SG used .NET Standard v2.0, .NET v7.0, xUnit.

- .NET Standard v2.0

It is a specification represent a set of APIs implemented by various .NET Platform frameworks. This specification represents the components that are required to exist by every framework implements it. a simple example will be the numeric library in .NET (**System.Numerics**). this library contains the available no primitive numbering types such as **BigInteger** and its constructors and available functions and operators. System.Numerics library has an API in .NET Standard represents the abstract library [2]. The .NET Standard v2.0 is important to the .NET Platform when distributing packages. Any package distributed in .NET Standard v2.0 will work on the major .NET frameworks [3].

.NET Standard 2.0 is the used by the C# source generators. So, the source generators will work on frameworks supported by .NET Standard 2.0 specifications.

- .NET v7.0

It is the latest version of the major .NET Frameworks until this documentation writing time. .NET v7.0 provides the latest C# .It was used in unit testing because it supports the latest C# version (C#11).

- xUnit

Unit Testing is a software testing method that means to test each component of the software separately as a unit. The unit is variable, but usually each unit represent a function, or small functionality in the software. The scope of a unit extends up to a single class.

xUnit is a package used for unit testing in .NET platform. The Organizer performed 55 unit tests across all of its components as needed.

Unit testing with xUnit is done with functions headed with an attribute called **[Fact]**, this attribute indicates to the test runner that this is a test function and should be ran as a test function. There are two other attributes for variable tests using a single function, any function headed with **[Theory] [InlineData(Param)] [InlineData(Param2)]…** represent a test function that will be testing with multiple variables (Param, Param2). Unit test functions are modularized, each test function usually split to three parts:

- Arrangement: In the first part of the function, the data meant to be tested will be defined.

- Act: In the second part of the function, the functionality meant to be tested will be executed on the data defined.
- Assertion: In the third part of the function, the execution results will be compared to the expected results of the functionality.

3) System.IO

The Organizer used the C# Input/Output library to create organized files and folders.

4) Integrated Development Environment (IDE)

The Organizer SG has been developed using Visual Studio 2022 17.5.5, Visual Studio provide useful tools to explore C# code syntax tree, also, tools to deploy the SG as a package for publishing.

1.6 Background Studies

Even though the Source Generators are relatively new as a technology, there is several use cases of it, one of them is about serialization. [4]

Serialization is a pattern that lets you save and restore the previous state of an object without revealing the details of its implementation, E.g., converting object to JSON. Serialization is often implemented using dynamic analysis, serializers often use reflection to examine the runtime state of a given type and generate serialization logic. This can be expensive. If the compile-time type and the runtime-type are similar, it could be useful to move much of the cost to compile-time, instead of run-time. Source generators provide a way to do this.

The process of compile time serialization is similar to how The Organizer work. It works by first annotating a type with an attribute created by the developer of the serialization SG, for example, the (Figure 1 – 6) is a class with an annotation to indicate that this class is a serializable class.

```
[GeneratorSerializable]
partial class MyClass
{
    public string Item1 { get; }
    public int Item2 { get; }
}
```

Figure 1- 6 Annotated Type with Serialization Attribute to mark it as serializable.

This Serialization SG aims to generate a JSON from the marked type with the serialization attribute, like in (Figure 1 – 7).

After that, the developer must use Roslyn to detect the syntax nodes of "MyClass" type and implement the logic of conversion to JSON and creating a String template of the desired generated form.

Then pass MyClass syntax nodes to the generator so the JSONs get generated.

```
{
"Item1": "abc",
"Item2": 11,
}
```

Figure 1- 7 Serialized Object of MyRecord Type

1.7 Conclusion

This chapter has discussed the project idea and overviewed the goal. as discussed, the project idea is to create a Source Generator (SG) for C# developers in order to assist them during their development process. The SG is used to organize any unorganized C# source code files of the client choice. The organization process is achieved through generating C# files and folders structured in a manner governed by a developer-built architecture. The SG will be published as a NuGet package called **Organizer**. This chapter also discussed the tools used for development, which were: .NET Compiler (Roslyn SDK), .NET v7.0 framework for the latest C# version in unit testing, .NET Standard v2.0 specification because SG can only be developed on .NET Standard v2.0, xUnit for unit testing.

Chapter 2: System Analysis and Design

2.1 Introduction

This chapter will analyze **The Organizer** SG as a software system in order to understand it. this chapter will discuss multiple diagrams.

2.2 The Organizer Diagrams

The diagrams included in this report, are only the essential diagrams required to understand how **The Organizer** SG work.

2.2.1 Class Diagram

This circular diagram in (Figure 2 – 1) describes the structure of the Node (major class) in The Organizer system. The (Figure 2 – 2) is a class diagram describes the Node structure in an official manner. The node is a type that represent a set of services requested by the client in a specific scope with additional information about the services. the services are structured in a hierarchy/tree (this structure is discussed in 2.2.2 FFBD 4th function). (Figure 2 – 1) represent what does the fields mean to the system. the "additional info" is the node description, it is a reference to the node parent in the hierarchy, and the direct descendent children, the field "Is Leaf" represent whether the node has children or not. The node value has a (block) field which it is the block the node contained in, and it is of type (BlockSyntax) and (Header) represent a collection of invocations. The node value has a (block) field which it is the block the node contained in, and it is of type (BlockSyntax) and (Header) represent a collection of invocations. This field is used to help the system internally in improving its performance. The invocation are other services (ignore types, contain types, update type, create folder).

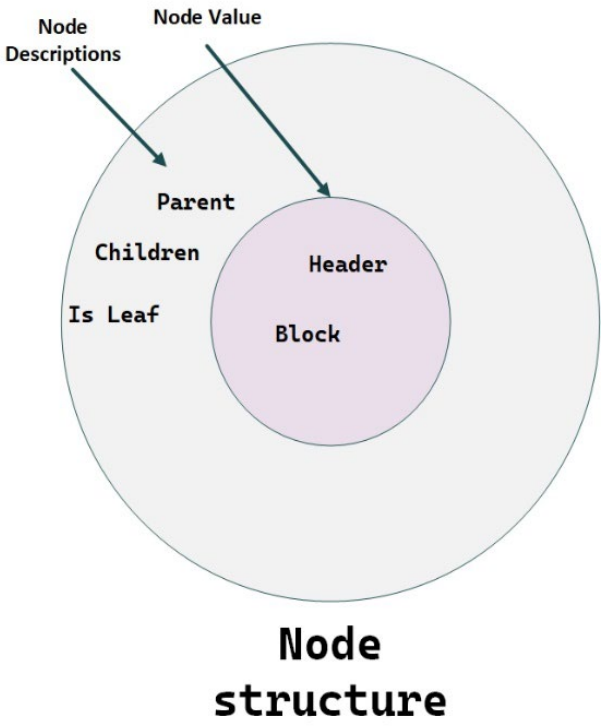

Figure 2 - 1 Node Structure

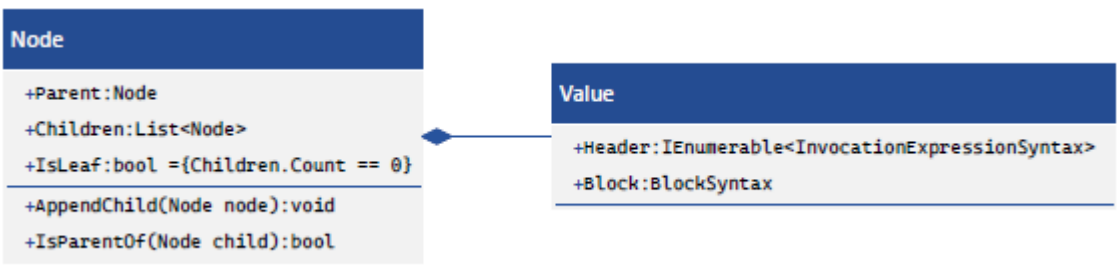

Figure 2 - 2 Node Class Diagram

2.2.2 Functional Flow Block Diagram (FFBD)

Functional Flow Block Diagram (FFBD) is a diagram used to show the sequential relationship of all functions that must be executed by a system. The FFBD is functionally oriented. The process of defining lower-level functions and sequencing relationships is often referred to as functional decomposition. It allows vertical traceability of the functional flow [5].

In order for The Organizer to work in the client side, the client must have a class that inherits The Organizer services class, inside this class there must be a declare constructor, inside this constructor will be the usage of the organizer services (create folder, get types, exclude types, etc...). the goal of FFBD is to describe the functional flow of reaching the organizer class in the client side and understanding the services requested by the client and execute them.

In **The Organizer**, the functional flow required to accomplish the task of source generation is described in the FFBD diagram in (Figure 2 – 3). The flow starts from top to bottom.

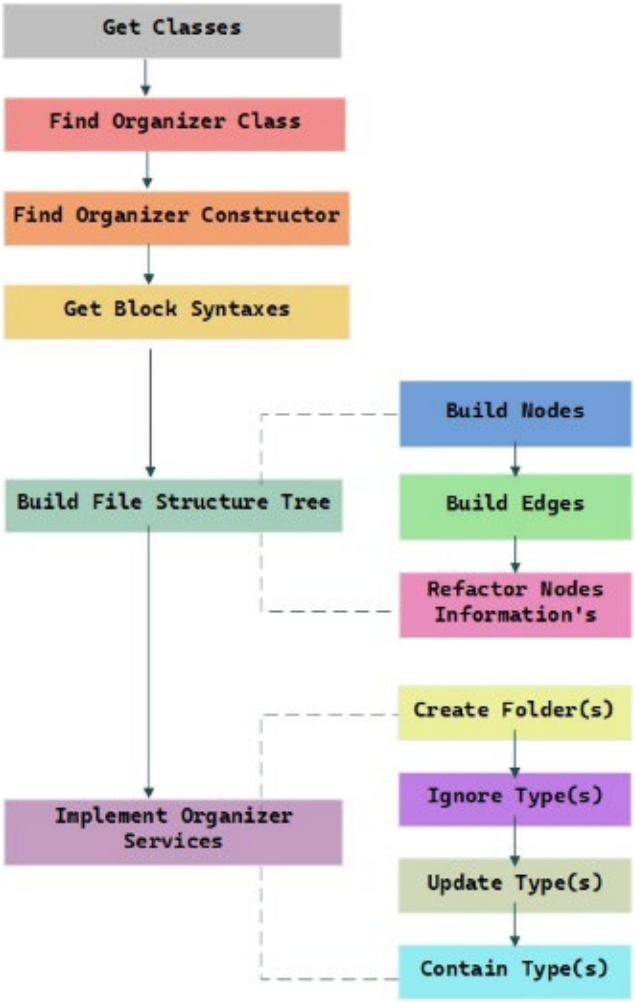

Figure 2 - 3 System Functional Flow Block Diagram (FFBD)

1- **Get Classes**: in this step the system was designed to retrieve all classes in the project that contains an organizer class in the client side. This is done using built-in Roslyn functions by retrieving a compilation object. then retrieving all syntax trees using a getter, then using a function responsible for getting all classes from all syntax trees available in the project.

2- **Find Organizer Class**: after getting all classes, the system will filter the classes depending on classes they inherited, the system will look for a class that inherit **The Organizer Services** class. It is important to note that the project must contain a single organizer services inheritor; if more than one organizer class presented in the classes list, the first one will be considered the organizer class. If no organizer class presented at all, the process will continue, but the output will be none.

3- **Find Organizer Constructor**: after getting the organizer class. The only important component in it is its defined constructor because it will contain the services requested by the client. So, in this step the system will filter the elements available in the class and retrieve the defined constructor. The only constraint is the class must only contain a single constructor; when there more than a constructor the generation will not occur.

4- **Get Block Syntaxes**: The Organizer services must be used in blocks pattern, because any structure in this pattern will be easy to imagine as a tree or hierarchy. The (Figure 2 – 3) represents a files structure as a tree, and as an imagination code. it is easy to think of the code in a hierarchical manner, where the blocks (Everything inside Curley Brackets {}) represent the scope of the folder and its content. The number of the blocks in (Figure 2 – 4) is three: the "Top Folder" scope, "folder 1" scope, "folder 2" scope. This hierachical code representation of files structure was introduced by The **Organizer** SG.

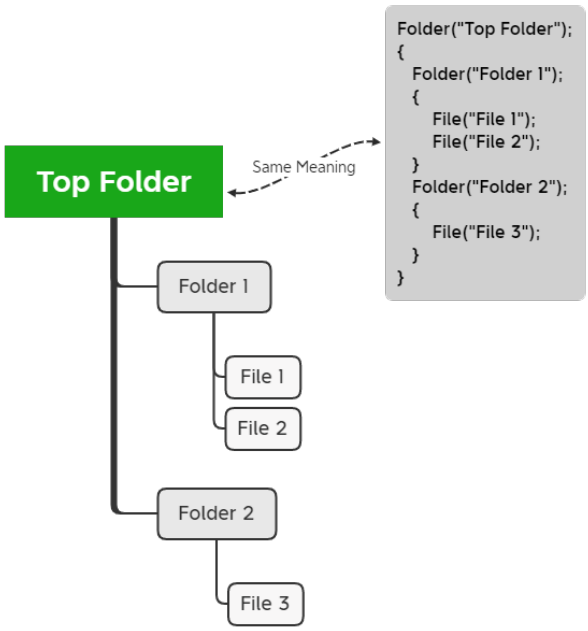

Figure 2 - 4 Structural Block Pattern Representation.

after the system retrieved the organizer constructor, it will look for blocks inside the constructor, the blocks will represent the structure of the generated files and what services were used. this step will return a list of blocks. An illustration of services in (Figure 2 – 4) is (Folder, File) pseudo functions.

5- **Build Files Structure Tree**: Up to this point, the relations between the blocks are anonymous, it is unknown which folder is the parent of which folder, what services it contains. The system will use the blocks data to build a representation of the files structure as a tree (which folders contains which types and what process must be performed upon these types). After getting the available blocks starting from the brackets of the constructor, the system will enter the step of understanding what services were requested by the client. This function will treat every block as a type called "Node", each Node contain the service it represents and any descendent Nodes -blocks- from it. And which Node it was descendent from. The (Figure 2 – 5) describes the phases of building the files structure tree in a Properties Assignment Phases Diagram[12]. This diagram has a new component: **snapshot:** it is a moment of time where the object properties values has changed. **Arrow Between Snapshots:** it represents the flow of object between functions.

the building of the Node objects happens in three steps:

1- **Build The Nodes**: in this step the system will initiate an empty node and assign a block of type BlockSyntax to it.
2- **Build Edges**: in this step each node parent (the node that this node is contained in) will be assigned.

[12] Properties Assignment Phases Diagram: is a diagram meant to illustrate object properties assignments in phases when an object is changing multiple times during its assignment.

3- **Refactor Node Information**: in this step the system will assign the Header field inside each node. The header will contain all invocations between a node and the previous node -like “folder 2” and “Folder 1” in (Figure 2 – 4)-. If no previous node available like in case of it is being the first node inside a scope -like “Folder 1” node in (Figure 2 – 4)- then it will get invocations between its block and the parent node block. This step will also update the CreateFolder(s) services paths inside each header, this step will improve the performance of the system by reducing complexity required to create nested folders. This function details are discussed in (2.2.4 Build File Structure Tree Build Design Decisions section).

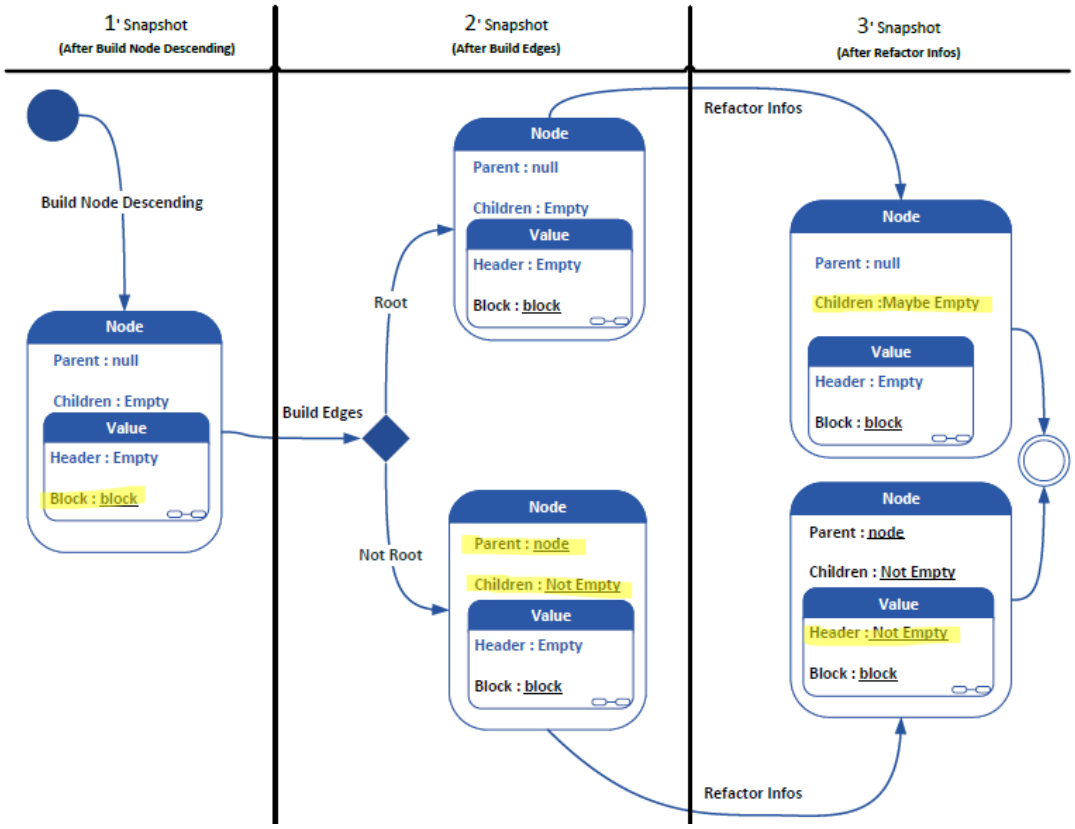

Figure 2 - 5 Build Nodes: Properties Assignment Phases Diagram. The yellow overlay indicate that this value has been assigned in this phase. The diagram shows the values of other fields as well in the current snapshot.

6- **Implement Organizer Services**: This function will look into each node header and check its service and execute the requested service and generate the code. This functionality is discussed in Section (2.2.5 Implement Requested Services Design) in details.

2.2.3 Activity Diagrams

The activity diagrams that are important to understand the complicated parts system are diagrams for the Header property in Node class.

2.2.3.1 Implement Organizer Services Activity Diagram

The (Figure 2 – 6) is an activity diagram that describes the sequence of services execution. The system always executes the CreateFolder(s) services first, these services create the folder(s) that will contains the generated files. After that, the system will check the .cs file(s) the client determinate it and the SG will use it to extract the Base Types from them. When it finds any Base Types. It will iterate over all available services in the system in the order Ignore Type(s) => Update Type(s) => Contain Type(s) and execute services according to what the client requested. it will look for nodes with **CreateFolder(s)** services first and create folders accordingly, after creating the folders, the system will look into services in the following order: **IgnoreType(s)** services => **UpdateType(s)** services => **ContainType(s)** services. This order is the best order performance wise, because ignore types will reduce types before update. Thus,

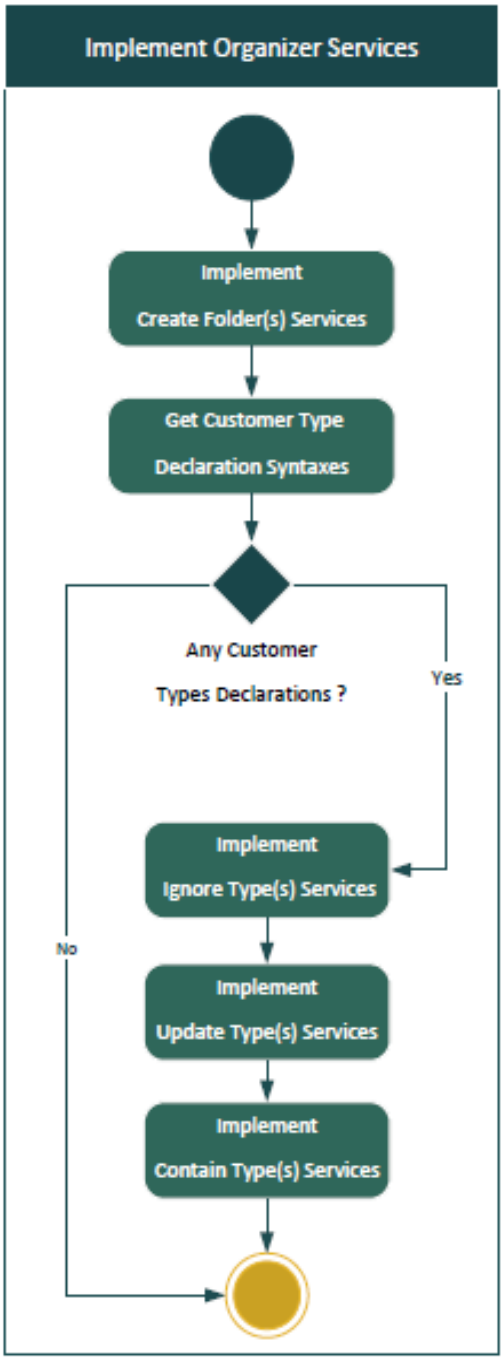

Figure 2 - 6 The Organizer Services Execution Sequence

2.2.3.2 Update Headers Activity Diagrams

The header in the nodes is treated according to the place of the node relative to its parent or previous node (sibling node). The (Figure 2 – 7) describes the activities of updating headers. The first child header of a node is treated differently from the header of the rest of children. As discussed in section 2.2.2 FFBD, the header of a node represents a collection of invocations but some of these invocations' parameters will update as seen in (Figure 2 – 3) in step 5.3.

- **Child to Parent nodes invocations:** Collection of invocations between the **"{"** in parent block and **"{"** that represent starting of first child node block , these invocations will be updated then assigned to the first child node header, (Figure 2 – 8) will descript these activities.
- **Child to Child nodes invocations:** Collection of invocations between the **"}"** that represent the end of previous child block node and **"{"** that represent start of current child node block, these invocations will be updated then assigned to the current child node header, (Figure 2 – 9) will descript these activities.

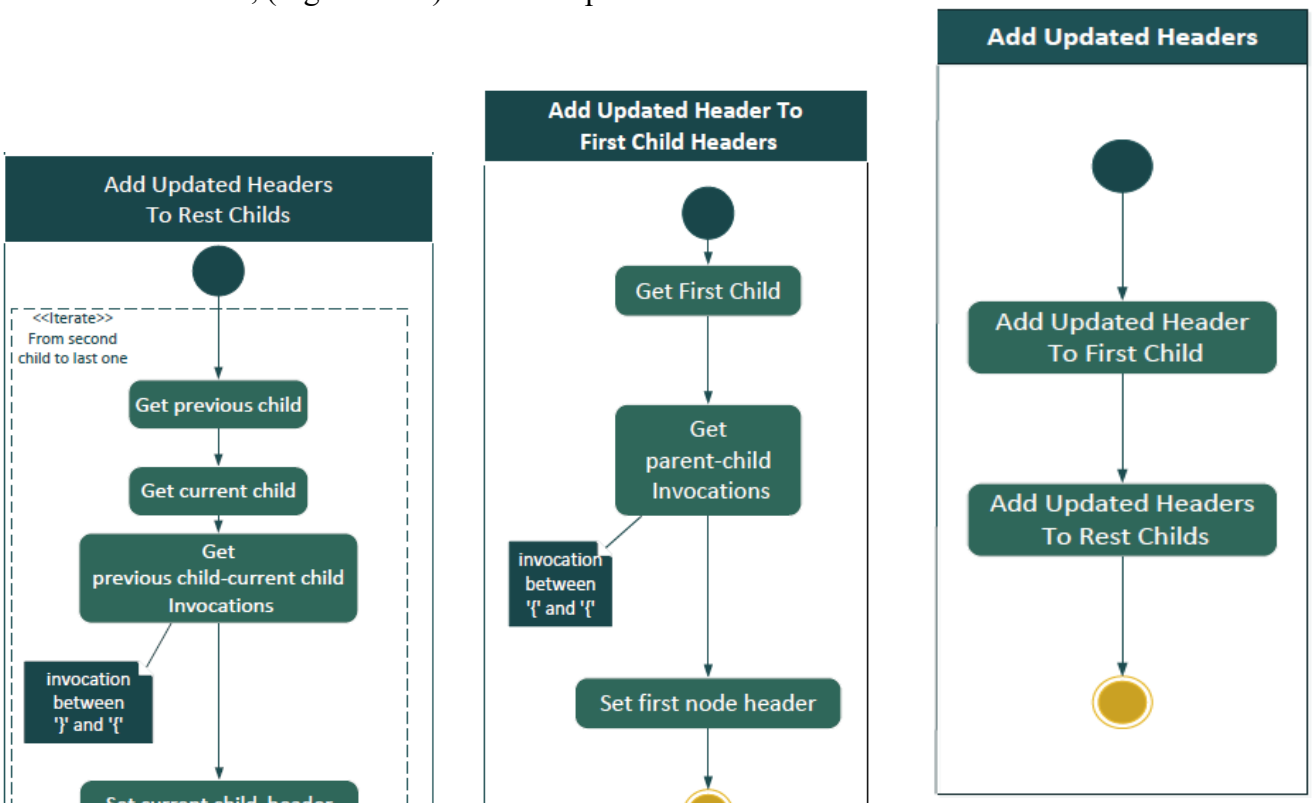

Figure 2 - 7 General Headers' Update Activity Diagram

Figure 2 - 8 Update First Child Header Activity Diagram

Figure 2 - 9 Update the Rest of Children Headers.

2.2.4 Performance Design Decisions of Build Files Structure Tree functionality.

In FFBD (Figure 2 – 3) the fifth function was **Build Files Structure Tree**, as discussed, this function is responsible to assign the relations between the nodes such as which node is a parent of which node, what services this node has. This function was designed to perform as good as possible because it is a crucial function in the system. It was composite of three steps:

1- Build The Nodes.
2- Build Edges.
3- Refactor Node Information.

The first step "**Build the Nodes**": internally, the nodes list is built in reversed order, from the inside out. (Figure 2 – 8) describes the way the list is built. The approach of building the nodes list this way assist the system in determining which node is a parent of which node in an efficient manner.

The second step "**Build Edges**": This step design is crucial for performance.

A way to assign the parent of each node, is to iterate over each node and save its block scope (where it starts and where it ends). And check if the rest of the nodes blocks scopes are inside this saved scope. This approach comes with a problem: the grandchildren are also in the saved scope. The (Figure 2 – 9) illustrates the problem. If the "Top Folder" asks who are my children? It is will check which nodes (blocks) are inside it (between where it starts and ends). But the problem is even "Folder 1.1, 1.2" and "Folder 2.1, 2.2" are inside it. Thus, it must come into the burden of filtering only the children which are "Folder 1" and "Folder 2".

If the question is reversed, and the node representing "Folder 1.1" searches the list for its parent. The node will search the list for the first node scope it is in and assign that node as its own parent; there will be no filtration as it is guaranteed that the assigned node is the direct parent. In (Figure 2 – 8) if the "Folder 2.2" searches for its parent, it will iterate up to the third index for "Folder 2". As it appears, the reversed list approach provided a drastic decrease in the

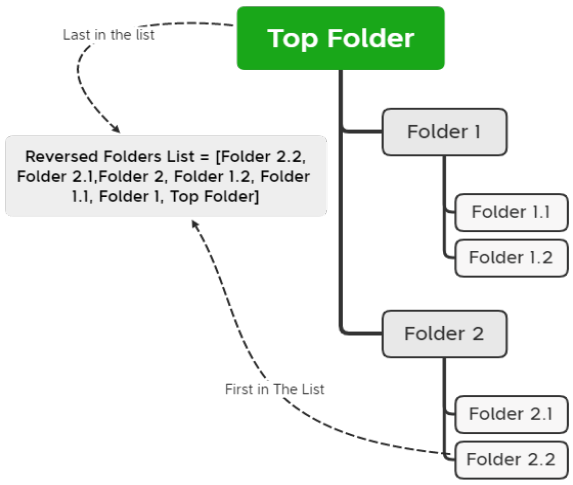

Figure 2 - 10 Reversed Nodes List Order

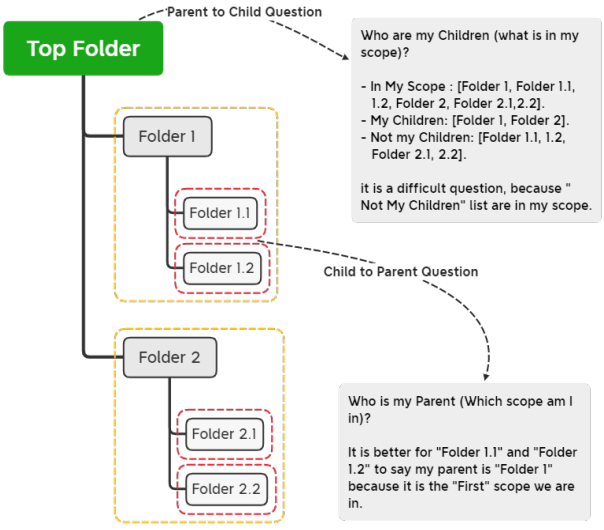

Figure 2 - 11 Folders Scope Problem. Yellow and red dotted squares represent the scope. The red scopes are contained inside the yellow scope.

possible conditions (if-else) to assign the parents of the nodes. After the parents are assigned, the list is reversed so the tree becomes in the right order, the from the "top folder" to "folder 2.2".

The final step "**Refactor Node Information**": As discussed, this step is responsible for assigning the header of each node and to update the CreateFolder(s) service parameter. In (Figure 2 – 10) there are 5 CreateFolder service usages. If the system wants to take a look into CreateFolder("Folder 1.1") parameter it will be "Folder 1.1". this piece of data is insufficient in term of creation path, there are several approaches get the path of "Folder 1.1".

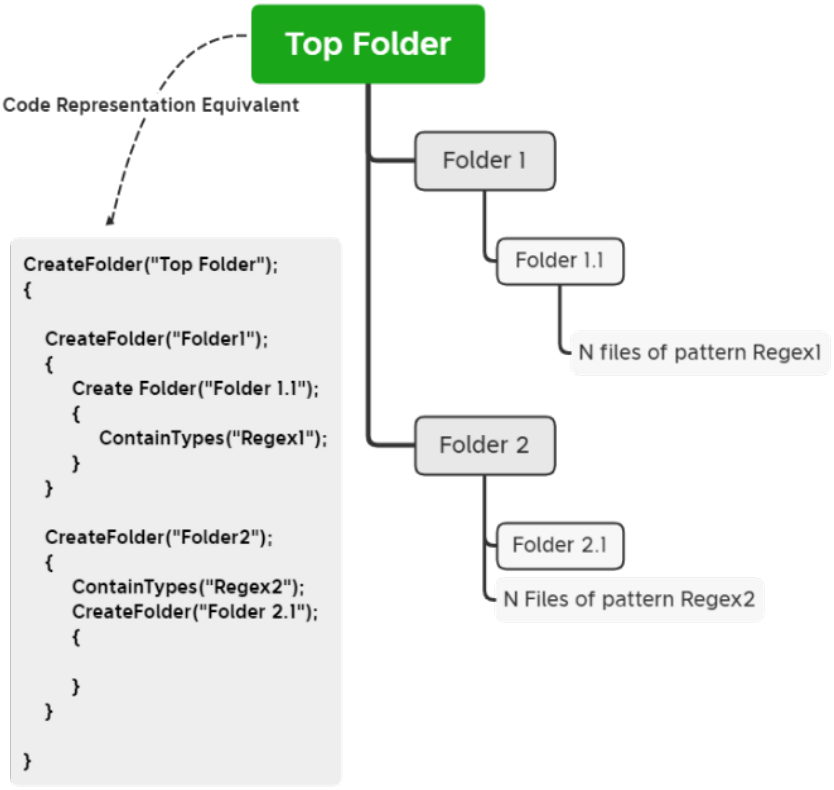

Figure 2 - 12 Nodes Example used to illustrate multiple CreateFolder service usages and its equivalent code.

one approach could be to not do anything at this step. And when the time for creating folders comes the system must go up in the tree to get each CreateFolder parameter (folder name) and create a composite path like (Figure 2 - 11). this kind of recursive pattern must be done to each CreateFolder service. As it appears, this approach is faulty in term of performance because of the large amount of repetitive work that must be done.

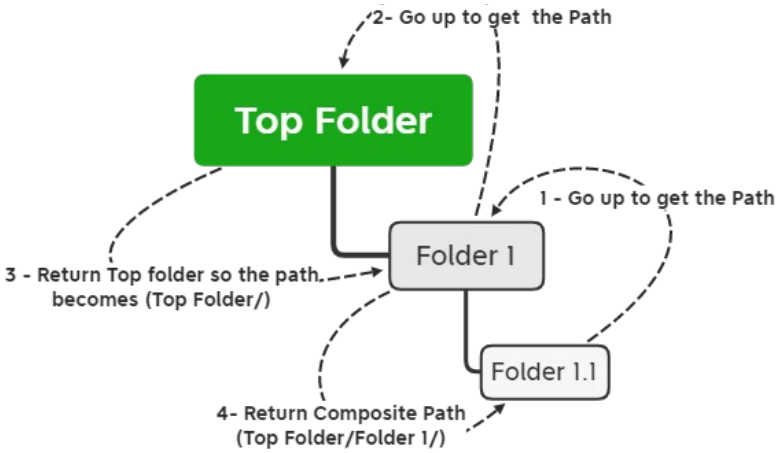

Figure 2 - 13 Bad Way to Get Folders Paths is to get them at the point of generation.

A better approach is to use the field "**Header**" inside each node. This approach starts assigning from the root node of the tree. the root node in (Figure 2 – 10) is "**Top Folder**" node. The system will search for its parent and it will be empty. So, the header will remain the same and no CreateFolder service is altered. After that, the system will treat the rest of children with two different relations, the *First Child to Parent* relation, and *Child to Child*

relation. In (Figure 2 – 12) The first child of “Top Folder” node will ask its parent node about its CreateFolder parameter (path). The parent will already have its own path which is “Top Folder”. So, the first child node CreateFolder parameter (path) will be changed to: CreateFolder(**“Top Folder/Folder 1”**). Then there is a second *First Child to Parent* relation between **“Folder 1.1”** node and **“Folder 1”** node. The first child “**Folder 1.1**” node will get its parent node **“Folder 1”** CreateFolder parameter, which is **“Top Folder/Folder 1”**, and append it to its own CreateFolder parameter. So, the parameter of CreateFolder(“Folder 1.1”) will change to CreateFolder(“**Top Folder/Folder 1/Folder 1.1**”) internally. but to the client it appears as is .

After this, there is a *Child to Child* relation between **“Folder 2”** node and **“Folder 1”** node. it is simple, the second child **“Folder 2”** node will get all invocations between itself and its previous sibling **“Folder 1”** node , then it will get the last CreateFolder service, which will be CreateFolder(“Folder 2”). and change the path accordingly. After this, the system will retrieve the parent header path and combine it with “Folder 2”. It will become CreateFolder(“**Top Folder/Folder 2**”). This approach reduced the number of iterations to get the absolute paths from the top of the tree down to the deepest leaf folder by only asking the direct parent or the closest sibling about its path. It is important to note that the internal workflow of editing the CreateFolder usages parameter is being done internally. To the client they appear as is.

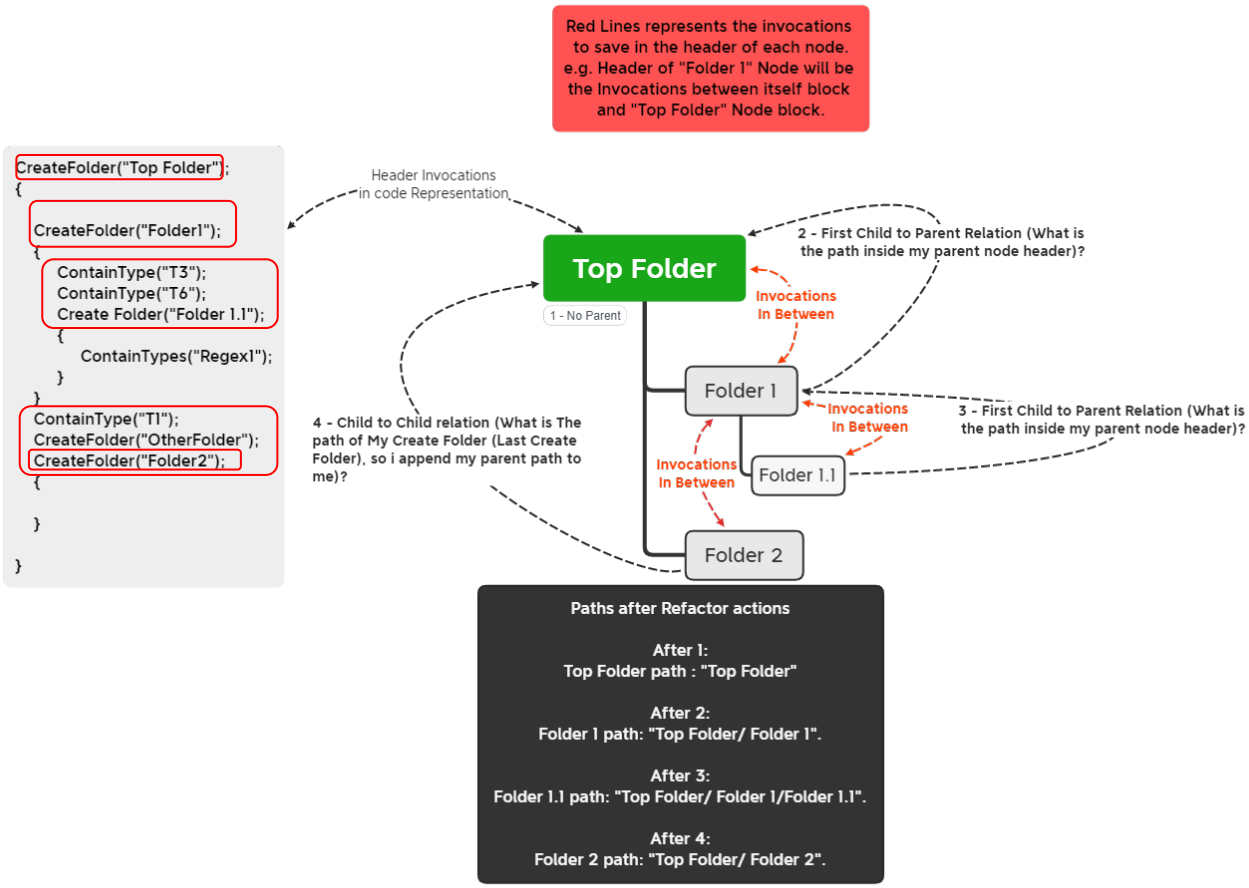

Figure 2 - 14 This Figure Represent the Nodes Folders Only for Illustrative Purposes. This Good Way to Get Folders Paths. The red squares on the code are invocations in the header of the node. And to edit the paths, the system is only interested in the last CreateFolder invocation in the header, like “Folder2”.

2.2.5 Implement Requested Services Function Design

In (2.2.2 FFBD) section, the 6th function in the Organizer system is to implement the requested services by the client. The implementation starts with Create Folder services. Then implementing the rest of services in a specific order: Create Folder(s) => Ignore type(s) => Update Type(s) => Contain Type(s).

Create Folders: This function has a single parameter representing the folder name. For starters, in Node Class Diagram (Figure 2 – 2), the property "IsLeaf" has never been used up to this point in the system. It is used during the creation of folders. A Leaf Node is a node with no children (descendent nodes). In (Figure 2 – 13) there are two leaf nodes: **"Folder 2"**, **"Folder 1.1.1"** nodes. These nodes do not have any children nodes (Nested folders).

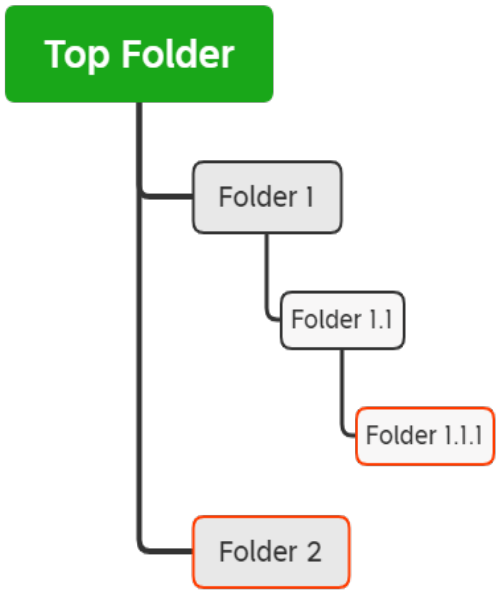

Figure 2 - 15Folders Tree Hierarchy to Represent the Meaning of Leaf Nodes. Red Marked Nodes are leaves.

The Organizer system will get these nodes, and attempt to get their path through CreateFolder service parameter. As discussed in "2.2.4 section", the system changed the parameter of CreateFolder services so they hold the path from the top folder down to the current folder, so the parameter of CreateFolder("Folder 1.1.1") **Internally** is "Top Folder/Folder 1/Folder 1.1/Folder 1.1.1". the system will get the absolute path of the destination folder from the Attribute [To: "Absolute/Destination/Path"] in the organizer class in the client side. Then append the absolute path to the leaf nodes. After this, the system will take advantage of a property in "System.IO" package. This package is capable of creating nested folders just by assigning the absolute path. So, the system will use System.IO to generate the folders, and only use it for the leaf nodes; when creating Folder1.1.1, all the precedent folders (Folder1.1,folder1, Top Folder) will be created in the way.

Ignore Type(s): this function has two kinds:

- `IgnoreType("Type Name")`
- `IgnoreTypes("Types Name Pattern")`

The process of executing the ignoring type(s) function is done internally using a function of the following structure:

```
IgnoreForTypes(Base Types Collection, Organizer Constructor Syntax Node)
```

This function is a composition of Three steps:

1- Filter the IgnoreType(s) invocations from the organizer constructor in the client side.
2- Filter types depending on the ignored type parameter (name) by returning a collection of only the base types that their names don't "**equal**" to any ignored types names.

3- Filter types depending on the ignored types pattern, by returning a collection of base types that their names don't "**match**" any ignored type regular expression (pattern).

the returned value from this function is a collection of base types without the ignored types.

Update Type(s): this function contains two kinds:

- `UpdateType("Old Type Name","New Type Name")`
- `UpdateTypes("Old Types Pattern","Updated Pattern")`

The process of updating the base types names is done internally using a function of the following structure:

```
UpdateForTypes(Base Types Collection After Ignoring Process, Organizer
Constructor Syntax Node)
```

This function it is composite of six steps:

1- Get Update Type(s) service invocations from the organizer class constructor in the client side.
2- The system split the processing of Update Types by names and Update types by pattern into separate phases, but the workflow is similar. The system will Map all Update Type(s) invocations parameters ("old name/pattern", "new name/pattern") into a single list containing both "old names" and "new names" for each invocation.
3- In this step, the system is supposed to update the names of the base types, but the problem is the Roslyn syntax nodes does not allow changing the name of a base type while maintaining its namespaces and using directive. So, the solution to this problem was to refactor each base types with updated namespace name and the using directives into a string like in (Figure 2 – 14).

```
using Library;
namespace A
{
    namespace B
    {
        class MyClass
        {

        }
    }
}
```

→

```
using Library;
namespace A.B
{
    class MyClass
    {

    }
}
this Text is a string
```

Figure 2 - 16 Refactor every base type with the closest name space and using directives to a "String".

4- The system will then iterate over each base type String and check if it contains the old type string pattern and replace it with the updated pattern.
5- The system will then parse all syntax nodes strings back to Roslyn syntax nodes and extract the Base Types after updating as a list.

The returned value after this step is a list of base types after updating specified base types and after refactoring the base types with their using statements and namespaces if exists.

Contain Type(s): This function has three kinds:

- `ContainType("Type Name")`
- `ContainTypes("Types Names Pattern")`
- `ContainTypes("Types Names Pattern", "Exclude A Type Name From The Pattern")`

The Process of containing types in a specific folder is done internally using a function of the following structure:

```
ContainForTypes(Base Types Collection After Update, Nodes Collection, Absolute Destination Path)
```

It is important to note that "Absolute Destination Path" parameter is the path in [To:"Absolute\Destination\Path"] attribute. Not the path of any folders to generate.

After ignoring some types and updating their names, now it is the time to create the files that contain the base types. This function workflow is as following:

1- Extract the path of each folder from CreateFolder service parameter and append it to the absolute path. So, the composite path is used for creating files in the correct path.
2- Iterate over each node and extract all (Primary) invocations within its block scope except (not primary) invocations inside **other** blocks like in (Figure 2 – 15).
3- Filter ContainType(s) services from the extracted services.
4- In this step, the system will first implement contain types by name services. It will create a list with all contained types names and compare the list with base types names in the base types list. When it finds a type that matches a name in the contained types list, it will create a file using **StreamWriter** from System.IO package with the extension ".g.cs", to indicate that this file is a generated file, where g stands for generated.
5- The system will then implement create files by pattern. The process is similar to the creation by name, the only difference is the system will check for any "Except" parameter and check for pattern matching except of equality.

```
CreateFolder("path1");
{
    ContainType("T1");
    CreateFolder("path2");
    {
        ContainType("T2");
    }
    IgnoreType("T3");
    ContainTypes("B");
    UpdateType("T4","TUp4");
}
```

Figure 2 - 17 Squared Services are extracted as they are primary invocations. They are in the scope of "Path1" Node.

The output of this function is the generated files with each file containing a single base type.

2.2.6 Conclusion

This chapter has analyzed the major classes of the system with a class diagram, and the major class is a class called “Node”. It represents a part of the structure in the generation hierarchy with additional info about it such as what services were used in what scope. and a circular diagram used to illustrate the Node and Its value in more illustrative way. This chapter also analyzed the functions of the system with a FFBD. And analyzed how each function work with activity diagrams and multiple illustrative diagrams meant to enhance the imagination of the workflow in the reader’s mind. This chapter also provided a deep dive through the major functions of the system which are building the file structure tree and the process of executing the services. It also provided a diagram called Properties Assignment Phases Diagram meant to illustrate the phases of the node properties values being assigned in multiple steps in the system.

Chapter 3: High Level Information on Source Code Generators and Their Types

3.1 Introduction

This section will discuss the types of source code generators and their history and how they were used.

3.2 Code Generation Types

3.2.1 Run Time Code Generation

Code generation as a technique started in the early days of programming languages with run time code generation which used Just in Time compilers[13] (JIT). When talking about compilation and executing code in JIT, the cycle may differ from programming language to other, but in general, JIT compilers have a fundamental way of work. The JIT compiler takes a pre-compiled byte code/Intermediate Language[14] of a software, and compile only required functions for execution, then execute them. During run time, the JIT check the call count of functions stored in the cache dynamically, when a count exceeds a predefined number, the JIT will understand that this function is used excessively and must be optimized for better performance, it will generate a reoptimized version of the byte code for that specific function and recompile it [6]. this workflow of re-optimization is called Tiered compilation in .NET [7]. (Figure 3 – 1) describe the phases of .NET JIT compilation.

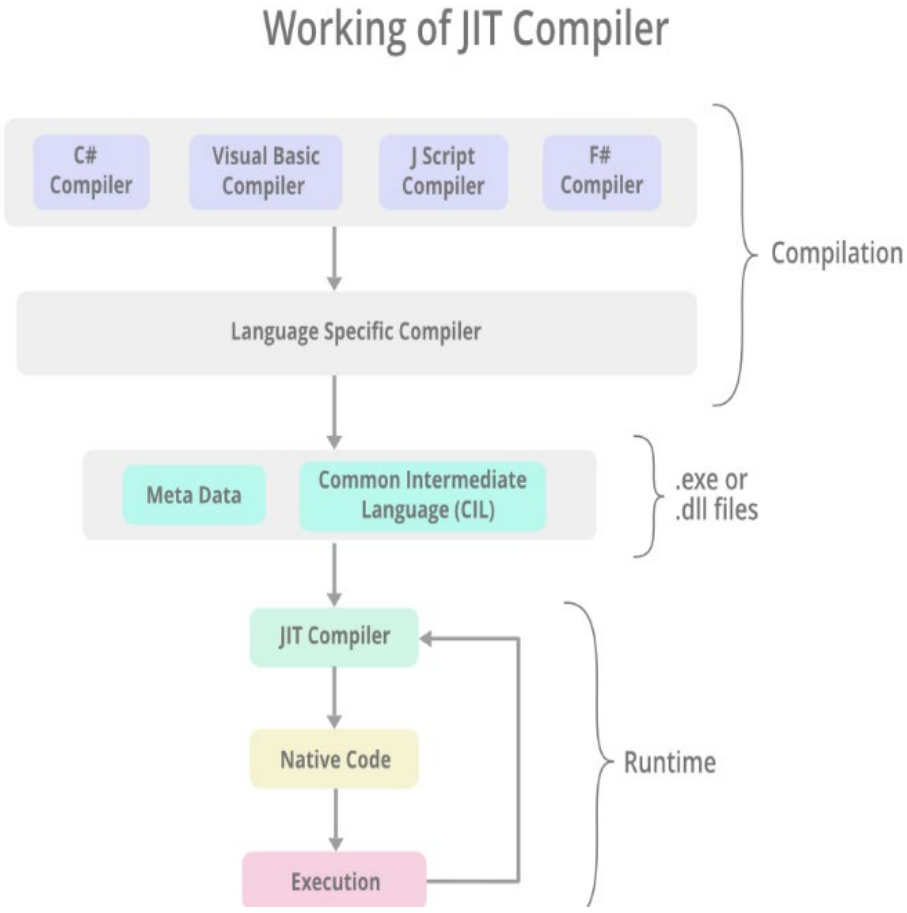

Figure 3 - 1 compilation phases with JIT compiler [13]

The JIT compilers take advantage of run time code generation to generate even more optimized code during run time analysis.

[13] Just In Time Compilation: is a type of compilation that does not compile the whole program before execution, rather than that, it compiles used functions only during run time and caches them.
[14] Byte Code: is a form of non-machine dependent code. This code is compiled by a language specific compiler. [14]

3.2.1.1 CodeDOM .NET

In .NET, there is a technology for generating source code files dynamically at run time. It is called Code Document Object Model (CodeDOM). This technology was introduced in 2003, so it is relatively old technology in .NET. it allows developers to create programs that can produce source code files in different programming languages in run time. It works by implementing an API interface in a C# class. This class will be a model that describes the code to generate. E.g., if the generated code supposed to be a class, the model class will describe the name of the generated class, its properties, methods and everything concerning them like return type, parameters, content. To accomplish this description, CodeDOM defines a set of types that correspond to various elements of source code, such as types, members, statements, expressions, etc. it allows the developer to use these types to build something called CodeDOM graph. This graph represents the structure of the generated code, it is similar to syntax tree, but it is not syntax tree because it is not language specific. To use the graph in generation, the graph will be inputted into a component called CodeDOM code provider, it is interface API that must be implemented by every programming language supports code generation of CodeDOM. the implemented interface will work as a translator for the graph. It will translate the CodeDOM graph into source code of the implementer programming language. CodeDOM provider is implemented in multiple languages: C#, Visual Basic, JScript. You can also use a CodeDOM code provider to compile the translated source code into an intermediate language (non-machine dependent assembly).

The code snippet in (Figure 3 – 2) is a snippet that represent a simple empty class creation using CodeDOM [8]. as it appears, the types defined in CodeDOM are very abstract and common across all the languages that supports CodeDOM code generation. All of them have classes with public and sealed access modifiers. The generated class is in (Figure 3 – 3).

```
CodeCompileUnit targetUnit;
        CodeTypeDeclaration targetClass;
        private const string outputFileName = "SampleCode.cs";

targetUnit = new CodeCompileUnit();
    CodeNamespace samples = new CodeNamespace("CodeDOMSample");
    samples.Imports.Add(new CodeNamespaceImport("System"));
    targetClass = new CodeTypeDeclaration("CodeDOMCreatedClass");
    targetClass.IsClass = true;
    targetClass.TypeAttributes =
        TypeAttributes.Public | TypeAttributes.Sealed;
    samples.Types.Add(targetClass);
    targetUnit.Namespaces.Add(samples);
```

Figure 3 - 2 CodeDOM Empty Class Generator

```
namespace CodeDOMSample

{
    using System;


    public sealed class CodeDOMCreatedClass
    {
    }
}
```

Figure 3 - 3 Generated CodeDOM Empty Class

CodeDOM is used in scenarios where there is a repetitive code with different input. such as unit test functions stubs. the process of writing empty unit test functions definitions could be automated with CodeDOM if a developer writes a function that take functions names as a parameter and outputs empty test functions.

Some of the disadvantages of CodeDOM is it does not support the whole syntax range of all supported languages, only most of common syntax. For example, CodeDOM does not support yield and async/await expressions and switch expression in C#. Also, another disadvantage is the hard use, the code snippet above represented the creation of an empty class, and it used a lot of code. So, working with this technology when creating large codes will be catastrophic.

3.2.2 Compile Time Code Generation

Compile time code generation introduced a new era of source code generation (SG); it was hard to create SG earlier on run time because of amateur technologies like CodeDOM. in codeDOM, it was hard to create SG because of the large amount of code required, and it lacked multiple common programming language components. And it slowed the development process of SG because every time a change occurs in the generator during development, it must run for changes to take place, because it is a run time SG.

Compile time SG solved the issue of slow development process by transforming the compilation of source code generators from run time to compile time. This resulted in much quicker debugging during development of SG and much easier use for clients.

In .NET, there are multiple technologies introduced by Microsoft for compile time source code generation. Two of them are T4 Templates, which is the first compile time SG. The other technology is Roslyn SG, which was used in The Organizer for code generation.

3.2.2.1 T4 Templates

T4 Templates stands for (Text Template Transformation Toolkit), it is a technology introduced by Microsoft for .NET frameworks in a time around 2005-2006 for compile time source code generation. A T4 text template is a mixture of text blocks and control logic that can generate a text file in any extension

type, the control logic is written as code in C# or Visual Basic. The generated file can be of any type, such as a web page, or an XML file, or a program source code in any language.

T4 Templates were also used to generate repetitive code structures in ASP .NET MVC[15], structures like model-controller view code. a controller in .NET MVC is a class entangled with a model, the controller represents the various http operations implemented by the developer on a model. controller view in .NET MVC is an HTML5 page that contains a visualization of the model and its controller http operations; thus, it makes interaction with the model and testing the operations during development easier.

T4 templates work through a file with either of the extensions .t4, .tt, tt was t4 before 2008, after that year the extension name is changed to .tt. This file will contain a text template for the text to be generated, it starts with meta data about the code, like in what language the template will be written in, and what .NET libraries will be used by the template, and the extension type of the generated file. It uses tags to distinguish various elements of the T4 templates, (<#@ Data Here #>) for template directives (meta data), (<# Data Here #>) for statement blocks which are the blocks to contain the text to be generated with its logic like loops, name spaces, classes, etc, (<#= Data Here #>) to use variables defined in a statement block.

The code snippet in (Figure 3 – 4) is a snippet from T4 template that generates a simple class with three properties [9].

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ output extension=".cs" #>
<# var properties = new string [] {"P1", "P2", "P3"}; #>
// This is generated code:
class MyGeneratedClass {
<# // This code runs in the text template:
  foreach (string propertyName in properties)  { #>
  // Generated code:
  private int <#= propertyName #> = 0;
<# } #>
}
```

Figure 3 - 4 T4 Template File Example

When saving the template file, it will generate the following class in a file named with the exact name of the template file but with .cs extension. The resulted generated class in (Figure 3 – 5).

As appears, the T4 templates has huge advantages over CodeDOM, it was able to generate a class with multiple fields with less code, it also solved of problem of slow development process by making the changes on the generator apply by just saving the file instead of rebuilding the whole project and running it.

```
class MyGeneratedClass {
  private int P1 = 0;
  private int P2 = 0;
  private int P3 = 0;
}
```

Figure 3 - 5 Generated T4 Class

[15] ASP .NET MVC: is a framework developed by Microsoft used to develop web applications in model view controller (MVC) design pattern.

Although, the problem with T4 template is that they are completely text based; thus, there is no syntax errors warnings, nor advanced IDE features. This problem resulted in very error prone environment.

3.2.2.2 Roslyn Source Generator

The newest type of Source Code Generators in .NET is Roslyn Source Generator (SG). This type of generators was introduced by Microsoft in 2021. It relies on the compiler of .NET itself and the components of it rather than relying on text error-prone input like T4 templates or on a not complete model like CodeDOM.

The SG has two tasks, first is to retrieve a compilation unit that represent the current compilation of the user code (every code in the project except generated code), the second task, SG will add source code to that compilation unit, which will be reflected to the user as generated files. the compilation object retrieved can be used to work with syntactic and semantic models of the compilation. Any data could be available in the syntactic analysis or semantic analysis phase will be available in the object. The (Figure 3 – 6) represent how source generator work as a part of the compiler.

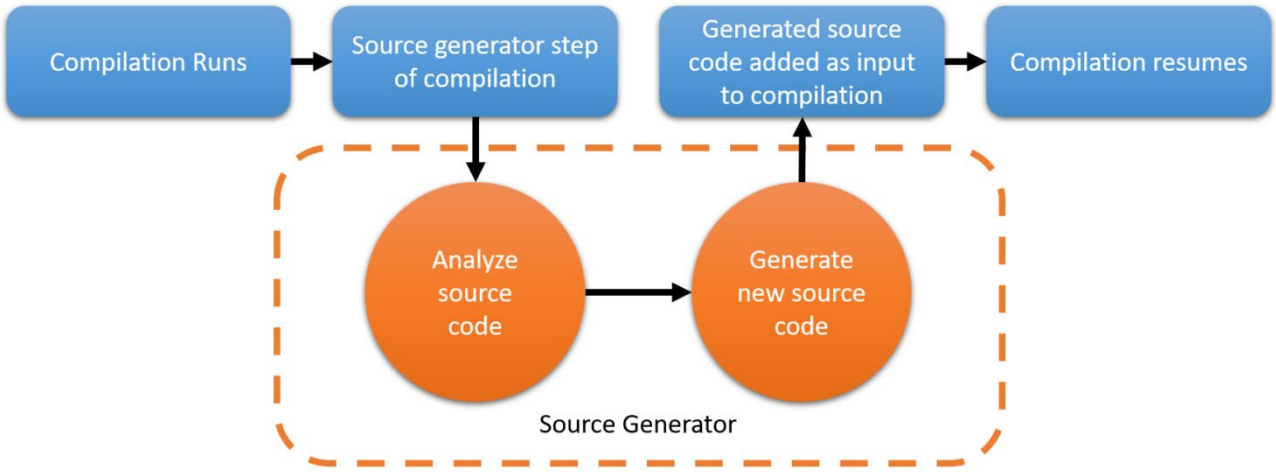

Figure 3 - 6 Roslyn Source Generator Workflow [15]

The source generators development is only supported by .NET Standard 2.0 specification, and can be used by every .NET framework supports .NET Standard 2.0.

The SG improves performance of code generation over runtime code generation by not relying on reflection[16]; instead, SG relies on information available in compile time.

[16] Reflection: is a technique used to interact with a system code in run time and inspect it, it is used to trigger a behavior in run time, this behavior in this documentation scope is code generation.

In order for the SG to be implemented, the project of the SG must target .NET Standard 2.0 assemblies. To assign the target, edit the Project.csproj[17] file. (Figure 4 – 1) shows the file after adding the targeted assemblies.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp"
Version="4.5.0" PrivateAssets="all" />
    <PackageReference Include="Microsoft.CodeAnalysis.Analyzers"
Version="3.3.4" PrivateAssets="all" />
  </ItemGroup>

</Project>
```

Figure 3 - 7 SG Generator prject.csproj file

To start using SG generators, a class must implement an interface called **ISourceGenerator**, like in (Figure 4 – 2). in the very high level, ISourceGenerator interface is responsible for the execution of the code generation process.

```
namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Code generation goes here
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // Information required before executing the code generation.
            // In most cases no additional info required.
        }
    }
}
```

Figure 3 - 8 SG Generator inheriter class

[17] Project.csproj file: is a file that contains meta data about the project such as its dependencies (included libraries) and targeted frameworks and other info.

To use the SG in other projects, it must be referenced by its project or by its assemblies. The reference must be added to the project.csproj file as well.

```
<!-- Add this as a new ItemGroup, replacing paths and names appropriately -->
<ItemGroup>
    <ProjectReference Include="..\PathTo\SourceGenerator.csproj"
                      OutputItemType="Analyzer"
                      ReferenceOutputAssembly="false" />
</ItemGroup>
```

Figure 3 - 9 Client project.csproj file that want to use a SG.

Chapter 4: Implementation and Results

4.1 Introduction

This section will discuss an execution of **The Organizer** and how it managed to achieve its goal of assisting the developers.

4.2 Execution

The execution will be a test on a base types generated from API JSON description using swagger documentation tools this API created for educational purposes, this API contains three kinds of models: requests, responses, data models. The API models are is contained in a single C# file. Imagine using this code to surf and locate models, it will be hard. In API C# file there is 2136 code line. As it appears, the file is very large.

To use The Organizer SG, the client must edit the project.csproj file and add the reference of the SG project or assemblies to it like how it was discussed in the previous section. After the client add the reference.

Here is where **The Organizer** will take place. The client will create a class to inherit a class from The Organizer called **OrganizerServices** like in (Figure 4 - 1). The client will specify the path of the API file in the [From] attribute, and the folder path that the organized generated code will be in. in the constructor of the class, the client will specify the preferred hierarchy with the services to request.

The client decided to create three folders with (CreateFolder service), where each folder contain related base types (ContainTypes service), a folder for requests, a folder for responses, a folder for data models. The client also decided to change the word "response" to "res" in every base type with (UpdateTypes service).

```
class Organizer : OrganizerServices
{
    [From("..Path\\To\\UnStructuredCode.cs")]
    [To("..Path\\To\\Destination\\Organizer.TestConsole")]
    public Organizer()
    {
        CreateFolder("OrganizedCode");
        {
            CreateFolder("Requests");
            {
                ContainTypes("Requests");
            }
            CreateFolder("Responses");
            {
                UpdateTypes("Response","Res");
                ContainTypes("Res");
            }
            CreateFolder("Models");
            {
                ContainTypes("Model");
            }
        }
    }
}
```

Figure 4 - 1 Client-Side Organizer Class

To run The Organizer SG, the client must build the project with the built tool, as soon as the project builds, The Organizer SG will run and execute the services. As it appears in (Figure 4 - 2), the files have been generated in separate folders like how the client intended, with each file contains a single model.

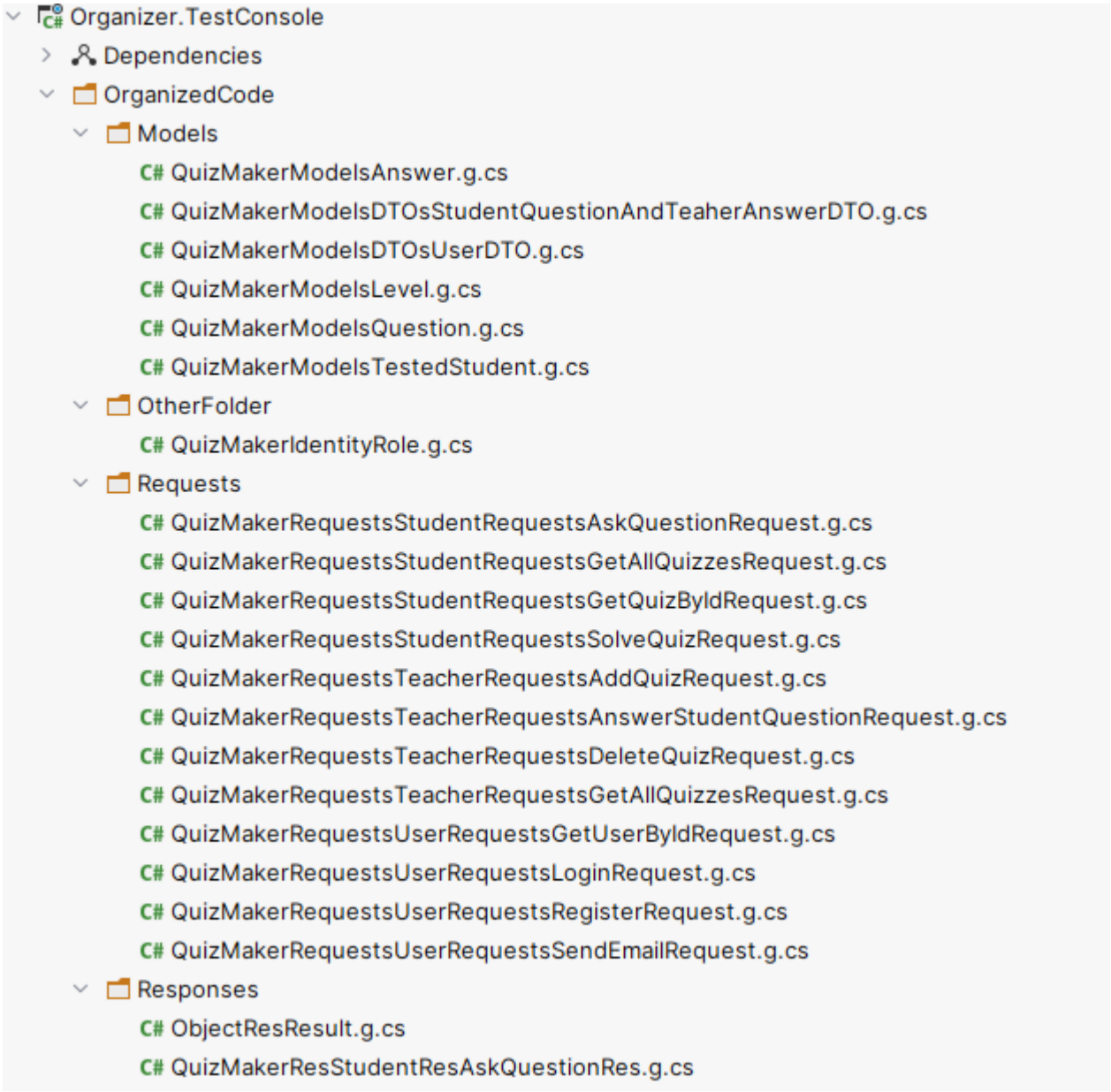

Figure 4 - 3 Project folders after generation

As soon as the project has been built once, The Organizer SG will keep running in compile time, if the client decided to add other folder to the hierarchy with a type like (Figure 4 – 3), changes will occur in compile time like in (Figure 4 – 4).

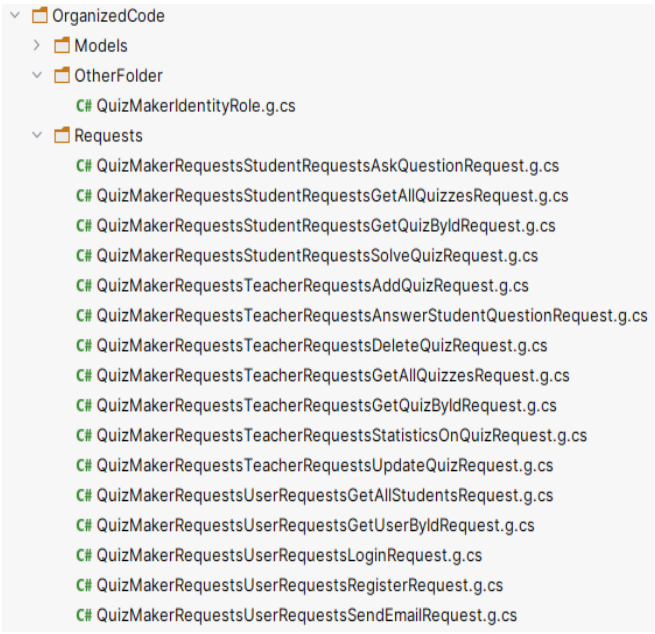

Figure 4 - 4 Folders After Adding a Folder with a type in compile time

```
public Organizer()
{
    CreateFolder("OrganizedCode");
    {
        CreateFolder("Requests");
        {
            ContainTypes("Requests");
        }
        CreateFolder("Responses");
        {
            UpdateTypes("Response","Res");
            ContainTypes("Res");
        }
        CreateFolder("Models");
        {
            ContainTypes("Model");
        }

        CreateFolder("OtherFolder");
        {
            ContainTypes("Role");
        }
    }
}
```

Figure 4 - 5 Edited Organizer Constructor, added Other Folder

4.3 Results and Conclusion

The execution of The Organizer SG resulted in an organized API models code from the client perspective. The original code will still be in the source code folders, but the client has the option of not using it, the client can simply use the new generated organized code.

The Organizer SG is supported in every IDE as long as it supports C# like rider and VS code and visual studio.

Chapter 5: Project Conclusion

5.1 Introduction

This chapter is an epilogue of the project. It will discuss the project conclusion and challenges during development and proposals with the future of The Organizer SG.

5.2 Project Conclusion

This project resulted in a tool called Organizer created using .NET Compiler (Roslyn). this tool is a compile time source code generator. It is used on C# code. it provides a set of services meant to organize (restructure) any unorganized source code the organization process is done by restructuring the unorganized code in multiple generated files and folders according to what the client desire. This tool will assist the developers in making the unorganized code far more usable and readable.

The Organizer SG will be published on NuGet Package Manager, this package manager is the manager of .NET package. The purpose of publishing is to make the Organizer available to all C# developers for use.

5.3 Proposals

- File Keyword: *File* keywork is an access modifier used to make a type accessible only at file scope. This keywork is used in place of *public* access modifier keyword. It is useful if the client wants to name the organizer class with name X and he also has other class of the name X. so the organizer class will be limited only to the organizer file scope.
- Resources for unstructured codes: Do not make the unstructured code within the scope of the project you use to organize it, for example: there is an X project that you want to organize, do the organizing process with another project Y so that there is no dependency between the two projects X and Y.

5.4 Challenges

- Folders as output: the source generator as a technology does not support generating folders, only files. This challenge was encountered by creating the folders using System.IO library.
- Custom generation paths: Source Generators as a technology has several limitations such as limiting the generation path to the dependencies; by default, any generated files are located in the dependencies. This problem was encountered by generating the files using System.IO library.
- Add support to C# 7.3: the Source generators can't be developed using newer versions of C# as the latest C# version they support is C# 7.3. Thus, it limits the developers to the features of this older version of C# instead of the newer versions such as C# 11.

5.5 Future Prospects

- Publish The Organizer as a package on NuGet.
- Solve Folders as output in an efficient way: relying on System.IO to create folders and files resulted in a long execution time when dealing with very large unorganized source code. the solution is to add creation on a specified path to the source generators, as it will be more efficient Than System.IO. issue will be solved in Visual Studio 2022 V16.
- Increase the scope of the project to other programming languages such as Dart and Java.

References

[1] "Documentation," OpenAPI, 2023. [Online]. Available: https://swagger.io/docs/specification/about/. [Accessed 9 4 2023].

[2] weshaggard and akoeplinger, "System.Numerics.cs," 22 5 2017. [Online]. Available: https://github.com/dotnet/standard/blob/release/2.0.0/netstandard/ref/System.Numerics.cs. [Accessed 19 4 2023].

[3] ".NET Standard," 11 08 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-2-0. [Accessed 19 4 2023].

[4] "Source Generators Cookbook," 17 3 2023. [Online]. Available: https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md. [Accessed 15 5 2023].

[5] D. G, "Space Systems Engineering: Functional Analysis," *NASA,* vol. 1, no. 16, 2008.

[6] FreeCodeCamp, "Just in Time Compilation Explained," FreeCodeCamp, 1 2 2020. [Online]. Available: https://www.freecodecamp.org/news/just-in-time-compilation-explained/#:~:text=Just-in-time%20compilation%20is%20a%20method%20for%20improving%20the,performance.%20It%20is%20also%20known%20as%20dynamic%20compilation.. [Accessed 26 4 2023].

[7] mattwarren, "A look at the internals of 'Tiered JIT Compilation' in .NET Core," 15 12 2017. [Online]. Available: https://mattwarren.org/2017/12/15/How-does-.NET-JIT-a-method-and-Tiered-Compilation/. [Accessed 27 4 2023].

[8] Microsoft, "How to: Create a Class Using CodeDOM," 15 9 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/how-to-create-a-class-using-codedom. [Accessed 27 4 2023].

[9] Microsoft, "Design-Time Code Generation by using T4 Text Templates," 9 3 2023. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/modeling/design-time-code-generation-by-using-t4-text-templates?view=vs-2022&tabs=csharp. [Accessed 28 4 2023].

[10] "BaseTypeDeclarationSyntax Class Reference," Microsoft, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/Microsoft.CodeAnalysis.CSharp.Syntax.BaseTypeDeclarationSyntax?view=roslyn-dotnet-4.3.0&viewFallbackFrom=netstandard-2.0. [Accessed 14 4 2023].

[11] R. C. Matrin, in *Clear Architecture, A Craftman's Guide to Software Architecture and Design*, 2017, p. 30.

[12] M. Stowe, in *UndistributedRest*, MuleSoft, 77 Geary Street, Suite 400, San Francisco, CA 94108, 2015, p. 1.

[13] harkiran78, "What is Just-In-Time(JIT) Compiler in .NET," GeeksForGeeks, 3 2 2021. [Online]. Available: https://www.geeksforgeeks.org/what-is-just-in-time-jit-compiler-in-dot-net/. [Accessed 26 4 2023].

[14] A. S. Tanenbaum, "Assembly Language Level," in *Structured Computer Organization*, Pearson, 2006, pp. 344, 518.

[15] P. Carter, "Introducing C# Source Generators," 29 4 2020. [Online]. Available: https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/. [Accessed 28 4 2023].

الملخص

هذا المشروع في مجال توليد التعليمات البرمجية (code generation)، بالضبط توليد التعليمات البرمجية في وقت التجميع. حيث مولد التعليمات البرمجية (Source Generator) يقوم بإنشاء ملفات التعليمات البرمجية أثناء قيام المترجم (compiler) بعملية التجميع.

يقدم هذا المشروع مولد تعليمات برمجية تم إنشاؤه باستخدام Roslyn Source Generator (SG) لمطوري .NET. ,مولد التعليمات البرمجية المدروس في هذه الاطروحة سمي المنظم (Organizer).

عَمل المنظم (Organizer) على تنظيم –حسب طلب العميل– أي تعليمات برمجية بلغة #C غير منظمة، من خلال إعادة هيكلة **الأنواع الأساسية** (الفئات ، الواجهات ، السجلات ، التعداد ، الهياكل) في مجلدات وملفات مختلفة وفقًا لاحتياجات العميل باستخدام مجموعة من الخدمات.

معنى الكود غير المنظم في نطاق المنظم هو ملفات التعليمات البرمجية #C المليئة بالأنواع الأساسية.

الخدمات التي يقدمها المنظم هي:

– خدمة لتوليد مجلد/مجلدات لاحتواء الملفات التي تم توليدها.

– خدمة لتضمين النوع /الأنواع الأساسية في مجلد مُولد بناءً على اسم النوع أو النمط.

– خدمة لتغيير اسم/أسماء نوع/أنواع معينة اعتمادًا على اسم نوع أساسي أو نمط لأنواع أساسية متعددة.

– خدمة لتجاهل النوع/الأنواع اعتمادًا على اسم النوع الأساسي أو نمط من أنواع متعددة.

– القدرة على استثناء أنواع معينة من التوليد أو التحديث بناءً على الأنماط في اسماء هذه الانواع.

تم تطوير المنظم باستخدام .NET Compiler (Roslyn) و .NET Standard 2.0. لتطوير SG . تم استخدام .NET 7.0. و xUnit لاختبار الواحدة.

الناتج عن تطوير المنظم, أداة يمكن أن يستخدمها العميل لتنظيم هيكل تعليمات برمجية الخاصة به أو جزء منها, في وقت التجميع . سيؤدي ذلك إلى المزيد من التعليمات البرمجية المصدر القابلة للاستخدام حيث سيكون من الأسهل تصفحها والوصول إليها وقراءتها.

يتم دعم المنظم على كل IDE طالما أنه يدعم #C ، مثل كود VS و Visual Studio و Rider.

Keywords: Generator, Source Code, Compile Time, Syntax, Service