Università
Ca'Foscari
Venezia

Master's Degree

in Data Management and Analytics (LM-18 – Computer Science)

Final Thesis

Run time code optimization using MEF

Supervisor

Ch. Prof. Claudio Silvestri

Graduand

Fatjona Ismailaj

Matriculation number

860833

Academic Year

2019 / 2020

CONTENTS

Acknowledgment

I would like to thank everyone who helped me with this thesis, especially my supervisor and my company tutor for his help with the development of this project.

I would also like to thank all my family for their support, encouragement and patience, for believing at me and helping me out in every moment during this journey.

And at least, but not less important, I would like to thank all my friends and university colleagues who have shared with me this difficult and beautiful journey.

Abstract

The aim of this thesis is to create an application, making use of C# language and the Managed Extensible Framework (MEF) library, able to be extended at run time choosing new libraries to add to the program or writing the code of the own library using the code editor provided by the application.
The code editor runs the user code at run time making possible to add the created library to the principal program by extending it. It will also give a research on the possibility to optimize this code by reducing its execution time.
Throughout this thesis, are going to analyze the tools that are used and identify their advantages and disadvantages, the requirements for this application, the research that are done, the future works and develop the application.
The result of the thesis is a small, extended application with an integrated code editor that gives the possibility to extend the project and a research on how the optimization can be done, evolved and be more complete and more precise.

1. Introduction

C# is a type-safe object-oriented language, a very popular tool for application development. It runs in the .NET ecosystem that is composed of all the implementations of .NET, including .NET Core and .NET Framework.
C# enables developers to create a variety of secure and robust applications like Windows client applications, client-server applications, XML Web services, etc.
C# syntax is highly expressive that provides powerful features such as nullable types, enumerations, lambda expressions, delegates and direct memory access. It supports generic types and methods, which provide increased type safety and performance, and iterations, which gives the possibility to the implementers of collection classes to define custom iteration behaviors which are simple to use.
During its lifetime, the C# language has been extended with different interesting features like LINQ or asynchronous programming support that has permitted to users to produce a lot of different frameworks and libraries for various purposes. In addition, this language allows a greater level of abstraction and is more forgiving to beginners.

As said before, C# language has been extended with new features like libraries, frameworks etc., and one of them is MEF, Managed Extensibility Framework that has a fundamental role in the development of this thesis.

Managed Extensibility Framework (MEF) is a new library, released in April 2010, that is been developed by Microsoft aiming to create lightweight and extensible applications. It is a component of .Net platform, more precise of .Net Framework 4.0.
This library provides an infrastructure for enabling an application to easily consume extensions by providing the ability to dynamically bind internal and

add-in components together through contract-based declarations.
MEF gives the possibility to discover the components implicitly instead of registering them explicitly. It makes this available via composition.

The main components of the framework are:

- *Parts*: are logical units of composition within MEF. A part declaratively specifies its dependencies, that are known as imports and what capabilities it makes available, that are known as exports. By doing this, the parts are discoverable at run time, which means that an application can make use of them without hard-coded references or fragile configuration files. Each part is identified by a contract, which are used to identify the dependencies between parts and takes the form of a string identifier. A part is any object that imports or exports a value.

- *Catalogs*: are responsible for discovering dependencies that exists between the registered parts. It provides a collection of parts from a particular source.

- *Containers*: are responsible to manage the creation and composition of the dependencies between parts. It uses the parts that are provided by the catalog to perform composition, the biding of imports to exports.

The Managed Extensibility Framework helps developers with providing tools needed to easily create extensible .Net applications.

1.1 Project statement

This thesis is based on using the C# language and especially the MEF library that are briefly explained in the introduction.
The principal idea of this thesis was to begin using the C# language, to show its potential and also to work with the libraries, frameworks and all the features that it offers.
The project that this thesis carries on, is focused on creating an application which is capable to be extended at run time and this can be possible by using the library that this language offers.
It has always been hard for the developer to create an auto-extendible application at run time, deal with dependencies between the components of the code, struggling to change the code every time that wants to add a new feature or to delete it. Using MEF makes it easier and provides the tools that let the user to be able to focus less on the concerns of the infrastructure development and more on simply creating excellent software.

What the application consists of?
As it is already specified, in the center of this application is the extensibility, how the user can extend the own program on run time, but also how to delete some functionalities that for him are not important.

The application has different options to permit the user to upload or unload the libraries.

- First of all, let the user to choose the libraries that he wants to add. These libraries are saved on his machine (libraries created by him or someone else) and of course are related to the relative application.
- Then, the user can choose to unload the libraries that do not want to have in his application. These libraries are the existing libraries in the application.

- A fundamental point of this project, is that the user can create a new library (new feature to add to its application) at runtime by opening the code editor that is available in this application, which act like an IDE to create scrips of code, compile them and immediately add new feature to the project, everything at runtime, without having to save this code before or to compile and run it on a separate project. In addition, the code editor gives the auto-completion of the code, code highlighting, comments, tree-parser etc.
- And at least, but an important part of this thesis, is the possibility that the code editor gave to optimize the algorithm written by the user by giving him, at the same window, the new optimized algorithm, that differently by the original one, has a reduced execution time. This operation is done only on the algorithms that can be optimized by substituting loops or statements that can change the executing time. This part comes in form of researches done on how an algorithm can be optimize, which are the methods to be substitute and how can be changed an algorithm to be more efficient. All these researches led to a new feature work that can improve the project that is going to be developed throughout this thesis.

1.2 Project Goals

In the project statement is defined the project that is going to be carried on in this thesis. The major part of the work is focused on creating a code editor inside an extensible project, which provides the way to create new libraries using the C# language that can be added to the main program and extended its features and functionalities making use of MEF.
What makes all this really useful is the fact that the user can open the code editor, write his own code, choose to optimize it, compile and so extend the main project, everything at run time, without having to create the new library

separately, save it and then upload it form his project.
Therefore, it will be explained how all this is possible, what tools are used, their advantages and disadvantages, how this application should look like and finally create it.

Summary

The primary goals of this work are:

- Take a closer look at C# language and its libraries, and use it to create the application
- Analyze the available tools that can be used to carry on this project and then identify their advantages and disadvantages
- Analyze the requirements for this new application, the application extensibility, the code editor, the optimization tool, explain how should perform and create them
- State all the researches that are done on how an algorithm can be optimized and what kind of optimization can be done

The result will be an extensible C# application, which provide a small and easy to use tool for uploading or unloading libraries to the project and creating extensions at run time that will add new features to the project taken in consideration.

2. Problem analysis

In this chapter will be analyzed the available tools that can be used for this project. Based on this analysis will be formulated the requirements for this application and specify the goals of this work.

2.1 Features of interest

What follows is an analysis of the features that are of the interest in the development of this application and explain why each feature is important and how can be ideally solved with respect to the aim of this work.

Ease of use

First of all, what is important in the development of this application is how effectively and easily the given tool can be used for extending an application. An ideal tool should be small, easy to use, quick and not strain system resources too much.

Free to choose the libraries to upload or unload

As this application will provide the possibility to extend a project, it is important that the user can have the freedom to choose which library to upload or which one to unload. To give this possibility, the user should have the chance to navigate on his own machine and choose the library that want to upload and as well, to have access to all the libraries of the project and to select which one to unload.

Referencing external assemblies

When the user wants to write the own code for creating a new extension to his project, an essential information of this process is the list of the

referenced assemblies for a successful compilation of any C# source code. So, the code editor that is provided need to offer a way to pass the information to the compiler. Because it is looking for an easy to use tool and the compilation will be at run time and not using the command line, the external assemblies will be specified in the same file in which is written the code.

Presence of a ‘specialized’ C# code editor

It is important that the code editor includes features that make the code authoring easier, such as syntax highlighting, error highlighting, code competition, adding comments to the code or the presence of the parse tree. An ideal code editor should include an advanced C# code editor, like having the debugger option to make the code authoring easier. But int this project the focus is on creating an application that can extend the principal project and not creating a specialized source code editor.

Presence of the optimization tool

To make the code authoring more efficient, besides the features listed before, another one that is helpful to be part of this application, is the presence of an optimization tool. What this tool should do, is to optimize the execution time of the user’s code by changing the loops or statements present in the code and making it to be executed faster than before.
This part of feature that is useful to have, is a research done during the development of this project and all this work will be presented throughout this thesis.

2.2 Available tools

To begin with, if the chosen language to develop is the C#, then there are some different IDE-s that can be used to carry on this project.

- **Visual Studio Code**

Visual Studio Code is one of the most popular code editors for C# development. It supports different extensions for powerful editing, full support for C# IntelliSense and debugging. Some of its features are:

  - Excellent auto-complete code
  - Build-in Git integration
  - Large list of extensions to further enhance the platform

- **SharpDevelop**

It is a free, open source IDE for C#, VB.net etc. It is a lightweight alternative to the Visual Studio Code.
Some of its features are:

  - Some features offered by Visual Studio, like code editing, debugging and compiling
  - Some advanced features like background syntax check and context actions

- **Rider**

Rider is a cross-platform .Net IDE from the Jet Brains suite of products. It is based on IntelliJ platform and ReSharper.
Some of its features are:

  - Supports both .NET Framework and .NET Core
  - Integrates seamlessly with other Jet Brains products
  - Context actions

These are a few IDE-s for C# development that can be considered for developing this project.

Another tool that is fundamental in this work, is the **Managed Extensibility Framework**.

This Microsoft library makes extension easier and lets the programmer to develop applications that can easily consume extensions by providing the ability to dynamically bind internal and add-in components together through contract-based declarations.

An extensible application that is written using MEF declares an import that can be filled by extensions components and may also declare exports in order to expose application services to extensions. Each extension component declares an export and may also declare imports. In this way, extension components themselves are automatically extensible.

To determine which exports can be matched with the imports, there exists a contract. This contract can be a specified string, or it can be generated by MEF automatically from a given type. The contract is fulfilled when any export is declared with a matching contract.

There are two different types of imports that can be specified:

- *ImportAttribute* that is filled by one and only one *ExportAttribute*
- *ImportManyAttribute* that can be filled by any number of exports

Each import should have an export, because every import left unfilled will result in a composition error. To get rid of this problem it is possible to declare imports to be optional or to assign them default values.

Another feature of MEF, which will be used in this project, is that provides the ability to look outside an application's own source for parts. This means that it extends the application with assemblies that are not parts of the source, letting the user to choose the assemblies (libraries) to upload.

As part of this project is also the **code editor** and as explained in the previous paragraph, to make the code authoring easier, it will support the code completion, syntax highlighting, error highlighting etc.
In C# there are some libraries that can help to develop these features in an application.

- **AvalonEdit**

Is a WPF-based text editor component. AvalonEdit can be used for syntax highlighting and it also comes with a code completion drop down window, but it needs to handle the text entering events to determine when to show the window. It was written for the SharpDevelop IDE.
While the WPF RichTextBox is quite powerful, its quickly running into its limits when trying to use it as a code editor: it is hard to write efficient syntax highlighting for it, and the user can not really implement features like code folding with the standard RichTextBox. The problem is that the RichTectBox edits a rich document, in contrast, AvalonEdit simply edits text.
However, AvalonEdit offers lots of possibilities on how the text document is displayed, so it is much more suitable for a code editor where things like the text color are not controlled by the user, but instead depend on the text (syntax highlighting).
The normal version of AvalonEdit requires .NET Framework 4.0 or higher.
[http://avalonedit.net/documentation/html/c52241ea-3eba-4ddf-b463-6349cbff38fd.htm]

- **Roslyn Microsoft Analyzer**

Roslyn is the compiler platform for .NET and consists of the compiler itself and a powerful set of APIs to interact with the complier.
Roslyn Analyzers a C# analyzer that can be used to analyze the code for style, quality, design, manageability and other issues. It can be used as an extension for the Visual Studio Code or can be download as NuGet and used in your own applications for code analysis, error highlighting, code

completion etc. It is an open source analysis library for C# and Visual Basic .NET languages. It allows syntactic and semantic analysis, dynamic compilation to IL and refactoring. [https://github.com/dotnet/roslyn-analyzers]

- **RoslynPad**

Is a cross-platform C# editor, based on Roslyn and AvalonEdit. It is available as NuGet packages that can be downloaded and allow to use Roslyn services and the editor in your own apps.

- **NRefactory**

Is the C# analysis library used in some C# IDE-s like SharpDevelop. It allows applications to easily analyze the syntax and semantics of the C# programs. It is similar to Microsoft's Roslyn project, but it is not a full compiler, it just analyzes C# code and does not generate the IL code. NRefactory 5 allows parsing and semantic analysis and it also contains supports for refactoring, formatting, code issue analysis and code completion.

Summary

As can be seen from the available tools that we can use to develop this application, there are a few of them that are quite similar and can be used equally to complete this work. Based on what is the aim of the project and based on the different tools that can be used for the same scope, below will be stated same requirements for this work, and then decide which of them can be more effective and easy-on-use for this purpose.

2.3 Requirements

In the pervious section are examined some tools that can be used for developing this application using the C# language. There are stated a few features of these tools and based on the points of interest stated in section 2.1 and based on the available tools, in this section is going to be formulated the exact requirements for the application.

2.3.1 Non-functional requirements

It is been stated several times that the aim is to create a small, quick and easy to use application.

Familiarity

Since this application is destinated to developers, which are used to work with classic IDE-s, it is a good choice to make the application look alike to them, such as the syntax highlighting for C# language on the code editor, the code completion etc.

Extensibility

This application is not expected to be a complete, all-inclusive features, but a well-usable basis that could be further extended and improved in the future by other developers based on what is been done till now.

2.3.2 Code Editor

The application will support C# code editor which let the user to create the own code. It supports a rich set of features that makes the code authoring easier.

Syntax highlighting

This is an important feature for every code editor because makes the code easier to read and understand. It is a good choice to include advanced syntax highlighting, that is not based only on simple regular expressions, but also on the semantic and syntactic analysis of the C# source code.

Code completion

While the user types, code completion offers a list of available types, members, variables etc. It allows to choose the identifier without the need to remember the exact name or without having to type out the full name. The list of code completion will appear automatically when it is typed an identifier, but only on places where it is expected.

Compilation

The code editor will be able to compile and run the scrip that is written internally and at run time. What we obtain from the compilation and execution of this file is a library, *.dll*, that will be automatically added to the principal program and extend its functionalities.
The source code written in the code editor should be compatible with the standard C# language but also compatible with the project that is going to extended.
This C# code file cannot be compiled on its own. It needs additional information such as the list of referenced assemblies. The referenced assemblies are essential for successful compilation of the code, but they are not part of the C# source code files themselves.
Since it will not have project files, the referenced assemblies should be specified in an appropriate textbox specified in the code editor.
As this will not be a specified IDE for C# language, it will not contain debugging, refactoring etc.

2.3.3 Optimization

This thesis, besides providing an extensible application and the code editor to write new extension for the existing problem, is focused on the optimization of the algorithms that are written in the code editor. What is required form this work is to give a way how the written code can be optimized, such as, rewriting the same code changing statements or loops that make its execution faster. But this is not always possible, because not always changing a statement with another or changing a loop with another will reduce the execution time. Beside this simple way of optimization, will be presented a research that is done during the development of this thesis on how an algorithm can be optimized, in what terms is possible to optimize an algorithm, like execution time, memory etc. and why is important to have an optimized algorithm.

3. Implementation analysis

In the pervious chapters it is been analyzed how this project can be developed, which are the available tools that can be used to write down this application and some general information about their functionality, what are the requirements that this project should achieve and how it is thought to work.

In this chapter the focus is on technical decisions regarding how this application will be implemented.

3.1 Tools and libraries

In the section 2.2 are represented the available tools that can be used for this project and also the MEF library which is going to be used for the implementation of this application.

The goal is to create a light-weight application, but creating a development environment such as the code editor that is part of this project is a demanding task, especially when are included features like code completion or compiling and execution.
To make the application extensible and able to consume parts that will be uploaded or created by the user is going to make use of the MEF library. On the other hand, to create the code editor there are two possibilities:

a) Making use of an existing open-source classic IDE.
b) Create it from scratch making use of the existing libraries and components that are designed to be used in IDE development.

The existing open-source classic IDE are large and complicated to use, so the best option is to create a form for the code editor from scratch, but using the libraries, which are represented in the section 2.2.

3.1.1 Programing environment

The implementation of this project is done in C# language, which will be also the language of the development on the code editor, and in the section 2.2 are stated some of the development environments (IDE) that can be used. The target platform to develop this project will be *Microsoft .Net Framework 4.0.* The IDE that is been chosen is the Visual Studio Code 2019, because it is more complete, it provides full support for C# IntelliSense and debugging, and it is an excellent tool for powerful editing.

3.1.2 User interface

Microsoft .Net Framework offers two different sets of APIs for user interface development, that are:

- Windows Forms: used to create powerful Windows-based applications
- Windows Presentation Foundation (WPF): offers great flexibility and has many features and abilities that Windows Forms lacks.

Windows Form and WPF will be part of this application.

Source code editor

A source code editor is a text editor program designed for editing source code. It may be built into an integrated development environment (IDE) or web browser. The source code editor is a fundamental programming tool and is the component the user interacts with the most.
To make possible the code highlighting, the two most known libraries are *AvalonEdit* and *ScintillaNET.* Both support a wide range of additional features that are useful for source code editors.
AvalonEdit is written in C# and WPF, whereas ScintilaNET is a managed Windows Forms wrapper around an unmanaged Scintilla control.
AvalonEdit library is also an easy to use one, which is chosen also for the development of this project.

3.1.3 Syntactic and semantic analysis

Creating the code editor, one of the most important goal is the ability of it to analyze and understand the code that is been written by the user. This ability is called syntactic and semantic analysis. There are some features that need source code understanding to work properly, such as code completion, syntax highlighting or code folding. Doing the syntactic and semantic analysis of C# source code is a large and difficult problem, but to make this possible it is making use of some libraries that are available for the C#

language.
In the section 2.2 there are listed some of the available libraries that can be used for such purpose. But for the development of this project, it was thought to use not just one of them but a combination of them and the reason why is doing so is to understand better how this libraries works, to be familiar with the different disponible libraries, what they offer, how easily to use they are, and which are the limitations that they have. The libraries that are chosen to be used are:

- AvalonEdit: it is used only for the syntax highlighting in a small part of the code editor where are defined the extra required assemblies
- NRefactory: this library is chosen for making the parsing of the code and creating the parsing tree
- Microsoft Roslyn (Roslyn Pad): it is a .NET Compiler Platform analyzer that examine C# code for style, quality, design and other issues. The analysis is performed at design time on all the open files. It is a well documented library, easy to understand and to use. This library is used to make the syntax highlighting of the written code, code completion and error highlighting.

3.1.4 Extensibility

As it is stated several times till now, the key part of this thesis is the extensibility, making the application extensible. Because of the aim of this application is to give the possibility to the user to extend his own project by choosing from his machine to add new libraries (as plugins) or unload them, or creating new plugins at runtime, the application has to provide the extensibility. A plugin in this context will be a separate unit of functionality that extends the application. Therefore, it will need a means of discovering, loading and initializing these plugins at runtime. The required functionality is this:

- The user chooses and loads the assemblies with plugins in his own computer
- The user chooses the assemblies on the application's directory to unload
- In these assemblies, find all classes that implement a specific interface
- Create instanced of these classes and return them on demand, or unload their assemblies

Microsoft .NET Framework contains two framework that support extensibility:

- Managed Add-in Framework (MAF): this does not suit very well to the needs of this application. It is intended for large systems where the host application and add-ins are strictly separated, run in different application domains and communicate across these domains.

- Managed Extensibility Framework (MEF): it is light-weight and covers the features this application needs, so this is the library that will be used

3.2 Pervious work

One of the goals stated at the beginning of this thesis was: *Take a closer look at C# language and its libraries and use it to create the application.*

Working with the C# language and their libraries was new and in fact, one of the most important goals were to study deeper a new programming language such as the C# language, be familiar with it, take a closer look to the features that this language offers and the libraries that are implemented to make the code authoring easier.

At the beginning, after having studied all the C# language from scratches, its features and having explored on simple projects developed with C# language and Visual Studio, was started with creating a simple button that shows a simple message when is clicked on it. Then creating buttons that are able to open a dialog window when it is clicked on them, selection files and uploading them on the project. When is been explored with these simple development features, was working on MEF and studied how it is implemented, why and all the features and facilities that it offers making the programming easier.

To understand better how MEF can be used and how it should be implemented in a project, it was created a small project Calculator, that will be used for the final application as the project that will be extended.
As can be seen from the project, it is structured in different windows (tab items) at the same form, to show all the progresses that are done till arriving to the final one, that is the final project and the main application.

It was a long and hard work to lean all the different characteristics of this language and the library taken in consideration, but it was worth it and at the end, all this work helps to develop the final project.

3.3 Application architecture

Creating an application able to extend a project and integrating a code editor on it is not an easy application to build. To make this simpler, it is thought that will be better to separate it into several components.

There will be these primary components:

- MainWindow, which will contain the main entry point and infrastructure of the application

- CodeEditor, which will implement the code editor window and all the features that it should contain and also the compliler
- Folding and RoslynEditor, which are files that contain the basic functions of code completion and code folding that will be used on CodeEditor
- SimpleCalculator, which is the small project created for learning how to use MEF that will be used as the project that can be extended by the application

3.3.1 Plugin, how this infrastructure works

The extensibility is the core of this application. Because of this, the application should provide the possibility to extend a project adding new plugins or unloading them. It is the user that will implement a plugin or choose the different plugins to load or unload and in this way the application should discover these plugins and load or unload their assemblies from the project.

Plugin discovery

In order to add new functionality into the application, all that will be required is to include the plugins into the application's code and make sure they implement a specific interface. These interfaces will have MEF's **InheritedExport** attribute which will ensure that they will be discovered, or there will be a special **Export** attribute if they are also need additional metadata.

Plugin initialization

A plugin will require some kind of initialization. This initialization could happen in the plugin's constructer, but plugin instances will be created by

MEF. For the plugins will be introduced a common base interface, that is the following one:

```
public interface IOperation
    {
        int Operate(int left, int right);
    }
```

Any plugin that implements this interface will have its *Operate* method called when it is loaded into the application.

When a plugin is created and it is not part of the same assembly as the project is, it should implement this interface and it will look like:

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata(“ ”)]
    public class Name: SimpleCalculator.IOperation
    {
      public int Operate(int left, int right)
      {
            Do something
      }
   }
```

Otherwise, it has to be implemented like this:

```
[Export(typeof(IOperation))]
[ExportMetadata(“ ”)]
    public class Name: SimpleCalculator.IOperation
    {
      public int Operate(int left, int right)
      {
            Do something
      }
   }
```

Plugin dependency

Because of how this application is thought to work, the plugins should not be dependent on each-other. This application gives the possibility also to unload a plugin, that means, when a user chooses a plugin to unload, it will

unload its assembly too. If the plugin that the user has chosen is dependent by another plugin that is currently uploaded in the project, it will not be possible to unload it, or if the user unload a plugin and a currently loaded plugin dependent on it, there will be an execution error, since the loaded plugin can not work properly if something is missing.
There is also another ‘problem’, if the user decided to upload a plugin that depends on another non loaded plugin, it will be an error execution since it is missing a plugin.

3.4 Compilation

The requirement of the compilations on the code editor are stated in the section 2.3.2.
In this section it will be discussed how that requirements will be satisfied.

3.4.1 Additional assemblies

Since it is required that the scripts written in the code editor be self-contained, it is needed a way to pass the additional information, that are the additional assemblies needed for the compilation and build. It is decided to divide the window (form) of code editor in two parts, where in the upper part will be specified the additional assemblies that should be included, in the middle part will be the part reserved to implement the code, and the third part will be the part to show the results of the compilation, if it is correct or something went wrong and shows the errors too.
This is how the code editor will look like:
(as you can see there is also written what each part of the window is supposed to do)

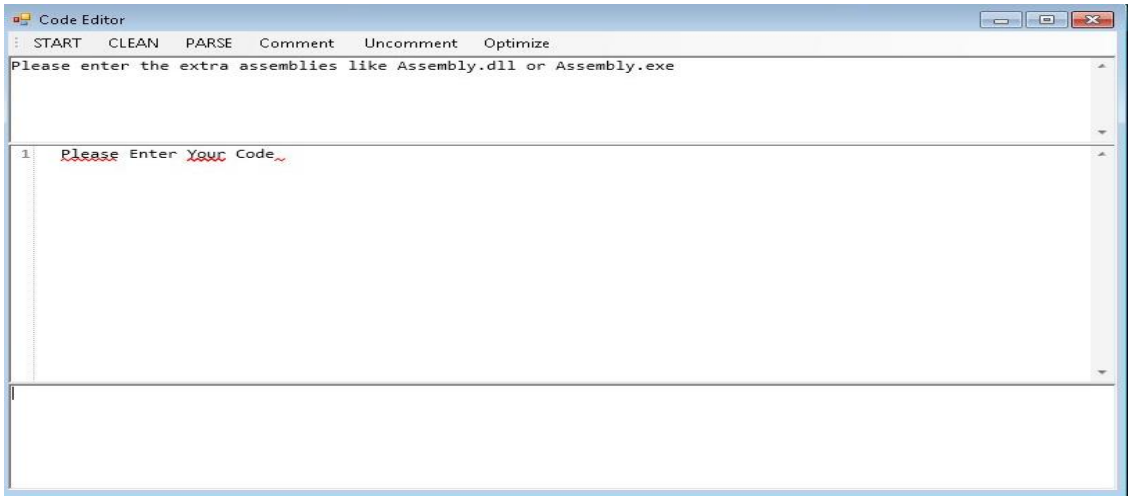

Figure 1

Since this compilation will produce a library, *.dll*, and not an executable, *.exe*, the assemblies will be of the form Assemblies.exe, if it is required to use an external project, or Assemblies.dll.

To make possible the compilation of the code and the creation of the assembly as a result, it will make use of the mechanism called Code Document Object Model (CodeDOM).

The System.CodeDom namespace defines types that can represent the logical structure of source code, independent of a specific programming language. The System.CodeDom.Compiler namespace defines types for generating source code from CodeDOM graphics and managing compilation of source code in supported languages.

Error Highlighting

Error highlighting is a very useful feature of any source code editor because it allows users to immediately see the errors within the source code without having to look them up manually. As was mentioned before, for the error highlighting it will be use Microsoft Roslyn.

Code completion and Syntax Highlighting

Code completion make the code authoring easier because it gives you suggestions on what you can write and do not need to remember the whole name of different classes, methods or libraries.
Syntax highlighting make the code easier to reed and understand but also make the code editor familiar to the user, since every IDE has the syntax highlighting for different languages. In this case, the syntax highlighting will be compatible only with the C# language.
To make these two attributes possible, it is used the Microsoft Roslyn (RoslynPad).

RoslynPad

It is a cross-platform C# editor based on Roslyn and AvalonEdit. RoslynPad is available as NuGet packages which allow the users to apply Roslyn services and the editor in their apps.
Some of the features that RoslynPad provides and that are used in the developments of this project are:

- Completion

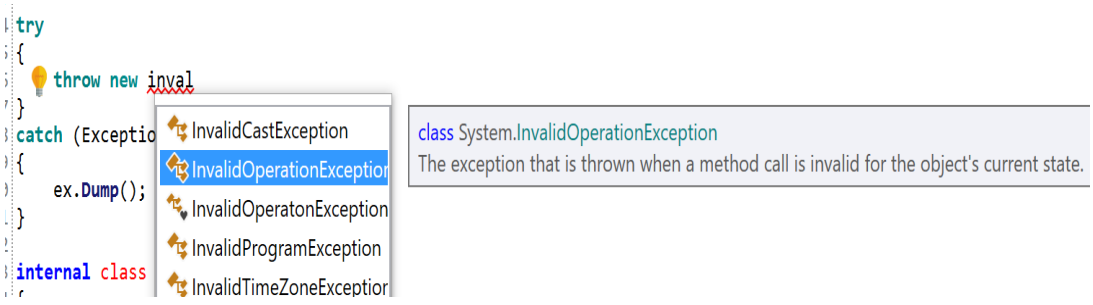

Figure 2

- Signature Help

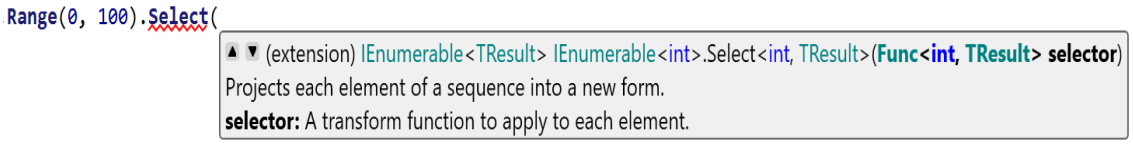

Figure 3

- Diagnostics

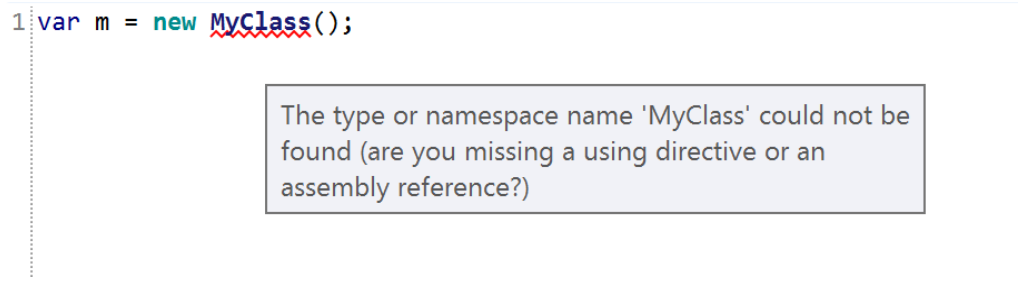

Figure 4

NRefactory

NRefactory is a C# language that allows parsing of the C# source code. It is not so easy to understand and work with it. In this thesis it is used only for parsing and creating the parsing tree.

How is created the parsing tree?
A parser is invoked on the C# source code file. The result of this parser is an abstract syntax tree that is called AST, which represents the syntactic structure of the original file.
The AST contains additional information about comments, whitespaces and exact position of individual syntactic elements within the original file, so that refactoring transformations can alter the SDT and render it back into source code text with minimum changes. It is represented by the SyntaxTree class and its nodes by AstNodes abstract base class.

4. Implementation

In this chapter will be explained how the whole application is been implemented. The source code of the implemented application can be found on the enclosed folder. The project is divided in two parts: the *SimpleCalculator* project and the *ShowMessage* project.
This chapter will be divided in 2 mini chapters dedicated respectively one to *SimpleCalculator* and the other one to *ShowMessage.*

4.1 Simple Calculator

In this section, will be explained how this project is implemented, what libraries are used and how does it work.
As was mentioned in the section 3.2 , Pervious work, this project was created to begin working with MEF library, to understand how does this library works, what are the facilities that it offers and how MEF can be integrated on a project to make it extensible.
Having this simple project done at the beginning of this thesis, it was then used as the project to be extended form the application that will be explained in the second part of this chapter, ShowMessage.

Libraries

This project is a C# project and to implement it is used Visual Studio 2019. The library that is fundamental for this project is the MEF library. Is possible to use MEF in client applications where it is used Windows Form, WPF or in server applications that use ASP.NET.

To use MEF in a project is necessary to add a reference to the *System.ComponentModel.Composition* assembly, that is the assembly where MEF is located.
On the project file it is needed to add *Imports* instruction for *System.ComponentModel.Composition* and *System.ComponentModel.Composition.Hosting.* These two namespaces contain the MEF types that will be needed to develop an extensible application.

Components

To begin with, this project is created as a Console Application in Visual Studio.
The components of this project are:

- *MyCalculator,* which is the core of this project. It is created as a *Console Application* in Visual Studio and the target framework is .NET Framework 4.7.2.
  In *MyCalculator* is the principal file where all the project is developed that is called *Calculator.cs.*

- *Extensions* directory, which contain a *Class Library* file, *ExtendedOperations* where are implemented the external extensions of the project. The file where the code is implemented is called *Extensions.cs.*

- *NewParts* and *Estensioni* directory, these are also *Class Library* in which are implemented more extensions for the project.

MyCalculator

As is stated before, this is the core of this project. In the file *Calculator.cs* is implemented all the application.

To begin with, in order to make use of the MEF library, is necessary to be added the references to *System.ComponentModel.Composition* assembly, that is the assembly where MEF is located. To add these references to the project, goes to add references of the project, search for this assembly and then add it to the references of the project.

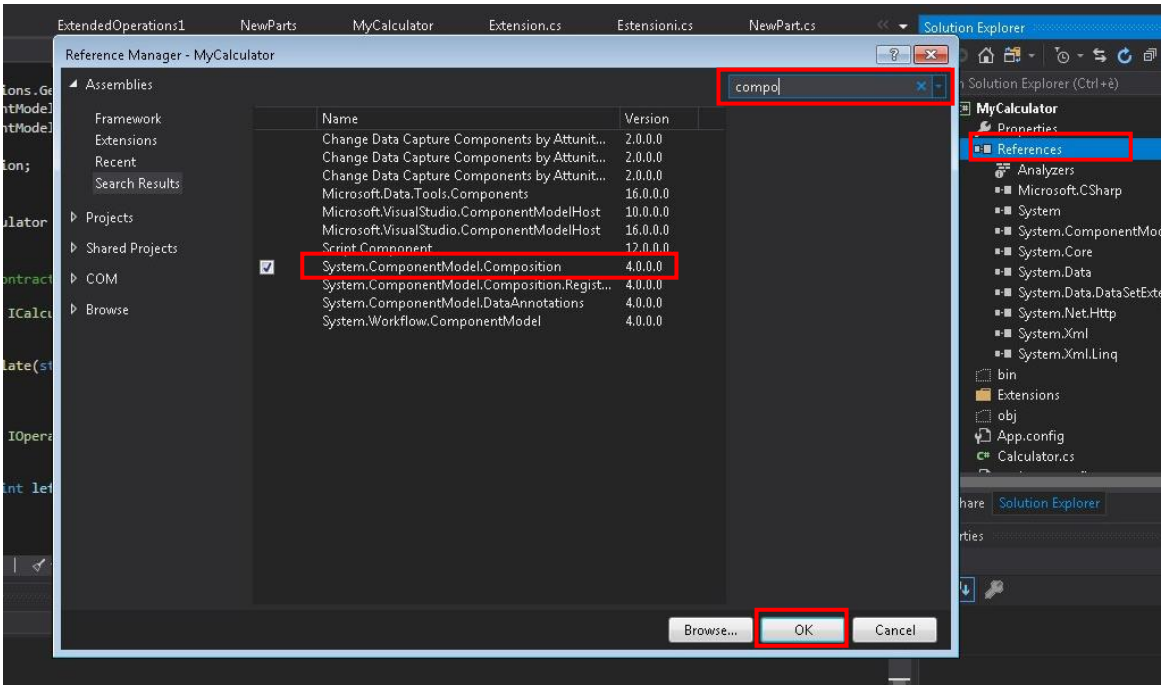

Figure 5

Now, to begin using MEF in the project, as the first point it is important to declare the contracts which are going to be used when an extension will be implemented.

The first interface that should be declare is the one that define the base of a calculator, the method calculate.

```
public interface ICalculator
    {
        string Calculate(string input);
    }
```

This interface has to be implemented by a class, which will implement the method Calculate, that will contain the logic of the calculator.

```
[Export(typeof(ICalculator))]
    public class MySimpleCalculator: ICalculator
    {
      ...
      public String Calculate(String input)
         {
            ...
         }
    }
```

Having declared the interface for the calculator, now is important to declare also the contracts for the extensions. As is explained in the Introduction how MEF works, it is required a contract which will be declared both from *Imports* and *Exports,* so in this way the extensions will be discovered from the project.
There are two interfaced to be declared, the first one for the *Operation* that will be done and the second one the second one is needed for the *Metadata.*

```
public interface IOperation  //OPERATION
    {
        int Operate(int left, int right);
    }

public interface IOperationData  //METADATA
    {
        Char Symbol { get; }
    }
```

But how can the application discover the different extensions that are declared under these contracts?
The basis of the composition on MEF is the *Composition Container*, which contains all the available parts and perform the composition between imports

and exports that are declared. The composition container uses a catalog to identify the parts that can be accessible. The catalog is an object that make available the parts found in source. There are different catalogs for different implementation of extensions, such as, *DirectoryCatalog, AssemblyCatalog, AggregateCatalog* etc*.*

So, in this project the catalog and the composition container are specified in a new class called *Program,* where all the composition and discover of the parts take please.

```
public class Program
{
  public static CompositionContainer _container;
  public static AggregateCatalog catalog= new AggregateCatalog();

      ...
      Public Program{
            _container = new CompositionContainer(catalog);
            try
            {
              _container.ComposeParts(this);
            }
            catch (ReflectionTypeLoadException ex){...}
      ...
      }
}
```

The class *MySimpleCalculator* should be discovered from the *Program* class in order to use the method *Calculate* that is implemented there.

To make this possible, before the declaration of the class *MySimpleCalculator,* is declared the *Export* under the contract *ICalculator,* which means that, to discover this class it is required to declare the an *Import* under the same contract in the class *Program.*

```
// export in MySimpleCalculator
        [Import(typeof(ICalculator))]
        public ICalculator calculator;
```

In this way, from the object *calculator*, it is possible to call the method

Calculate implemented in the class *MySimpleCalculator.*
It is specified that to discover an extension it should implement a contract, the *Export*, that is compatible with an *Import.*
In order for an *Import* to be matched with more than one *Export,* it is required to declare it in this way:

```
[ImportMany]
IEnumerable<Lazy<IOperation, IOperationData>> operations;
```

`Lazy<T, TMetadata>` is a type provided by MEF to hold indirect references to exports. Each `Lazy<T, TMetadata>` contains an object representing an actual operation and an object representing its metadata. The metadata, in this specific case, is the symbol that represent the operation that can be done, such as '+', '-', '*' etc.
ImportMany gives the possibility to associate to it more than one *Exports*, so more than one extension.
This statement is declared in the *MySimpleCalculator* class which is than imported in the *Program* class, where is declared the composition container and the catalog.

Having prepared all the elements that are necessary to discover an extension, it is time to implement the extensions.
There are two different ways how to implement an extension and where. The first one and most natural, is to implement the extensions directly to the same file where are implemented all the other elements of the application, which means to implement them in the same assembly. The other way, is to implement the extensions in separate files, class libraries, that means that the extensions are not in the same assembly of the principal file of the application, but have to be discovered outside of it.

To implement an extension in the same file is easy. The essential thing to do is to declare the contracts, the *ExportAttribute* for the operation and the one for the metadata, *ExportMetadataAttribute*.

When the application will be Bild, it will discover this new part added to the program and it will upload it to the application extending it with new features and new functionalities.

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '+')]
public class Add : IOperation
{
     public int Operate(int left, int right)
     {
          return left + right;
     }
}
```

This represents an extension of this project. The *ExportAttribute* with the contract *IOperation* and the *ExportAttributeMetadata,* which contain the operation '+'. This class implement the *IOperation* interface, that means it should implement the method *Operate*.
And at last, an important part of this components is the *Main,* which execute all the methods that are declared before and pint on the console the results of this application.

```
public static void Main(string[] args)
{
     var p= new Program();
     String s;
     foreach (var item in catalog){
          Console.WriteLine(item.ToString());
     }
     Console.WriteLine("Enter a command:");
     while (true){
           s = Console.ReadLine();
           try{
              Console.WriteLine(p.calculator.Calculate(s));}
           catch (Exception e){

     Console.WriteLine(e.InnerException.ToString());}}

 }
```

Extensions directory

In this section will be stated how to implement an extension in another file, on a separate assembly and how it is discovered by the application.
MEF offers the possibility to search for parts outside of the application's source code. In order for this application to be able to discover parts outside of its source code it is added a *DirectoryCatalog,* which tells the application where to search the parts. Done this, it is important also to change the directory output of the *ExtendedOperations Class Library.*

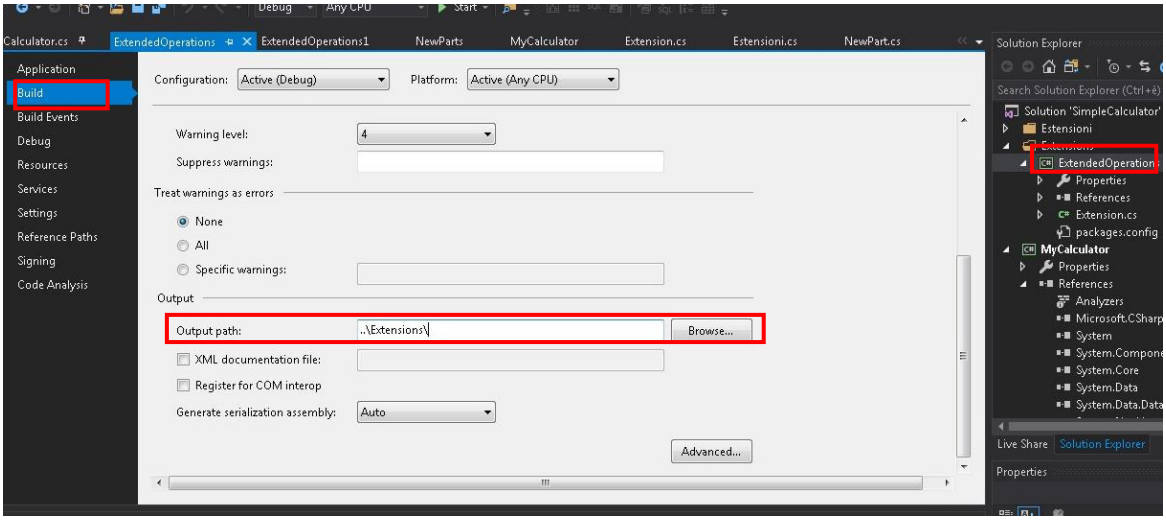

Figure 6

This is the way how the application reaches the parts defined outside the source code and this is the second way how an extension can be defined.
In this case the new extensions are defined in the file *Extensions.cs.*
This is how the extension out of the source code should be defined:

- First of all, should include the references for the MEF library, otherwise it will generate an error

```
using System.ComponentModel.Composition;
```

- Then implement the extension inside the namespace *ExtendedOperations*

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata("Symbol", '*')]
public class Multiplication : SimpleCalculator.IOperation
{
     public int Operate(int left, int right)
     {
            return left * right;
     }
}
```

Differently from the extensions declared on the same source code of the application, in this case is necessary to specify in the contract the namespace where the contract is defined and also is necessary to define the namespace when the interface is used, *SimpleCalculator.IOperation.*

NewParts and Estensioni directory

These two components of this project are similar to the previous one. In them are declared new extensions that are in different assemblies from the source code and is preceded in the same way as it was done in the section before.

Output

The output of this small project is a console extensible application.

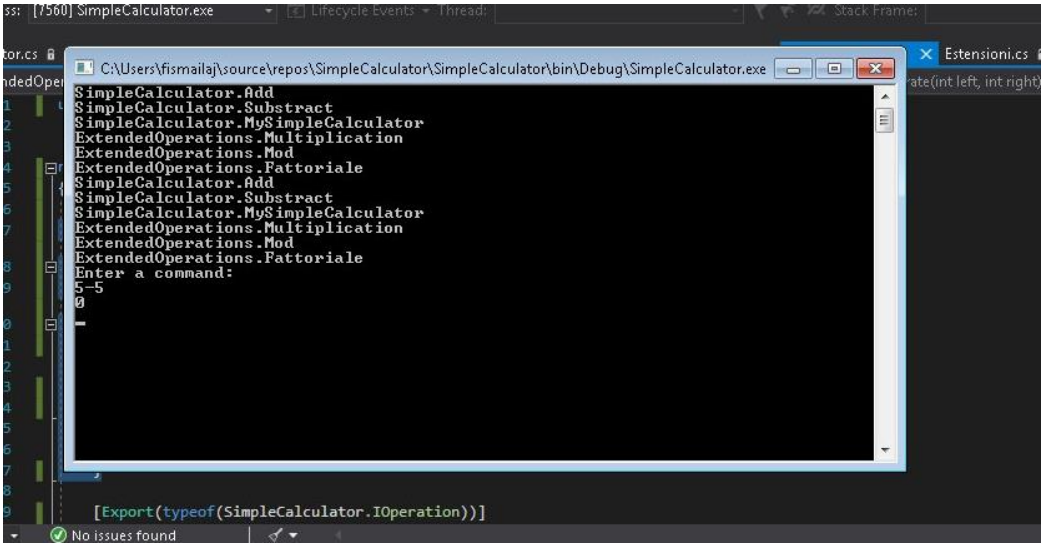

Figure 7

The output of this small project looks like this. It shows all the extension that are in the catalog, (in this case the extension is added twice) which are also the available one that can be used.
After the line *Enter a command,* it is possible to enter an expression, just like it is shown on the figure, and the application will give the result of that operation. When an extension is deleted from the catalog it will not be shown on the output window and also will not be available to use it. It is the same, when a new extension will be added, it will be shown on the output window and it will be available to use.

4.2 Show Message

Int this section, it will be treated the principal part of this thesis, that is the application that extends the project explained before.
This application, besides this application, contains some other small projects, which are divided in different tab windows and contains the small projects developed before arriving to the final one. These projects are the ones explained in the section 3.2, *Pervious work.*
The *Figure 8* shows the interface of this application. There are 8 different tab windows, where the first seven windows contain the pervious works that are done before arriving to the creation of the final application for the thesis. It is preferred to let them with the final application, in the way to show how this work has evolved and to show that creating an application such is that one is not an easy work that can be done in some days.
As was explained for the project Simple Calculator, in this section will be explained the different libraries that are used, the references added, the components of this application, how it is implemented and the final output.

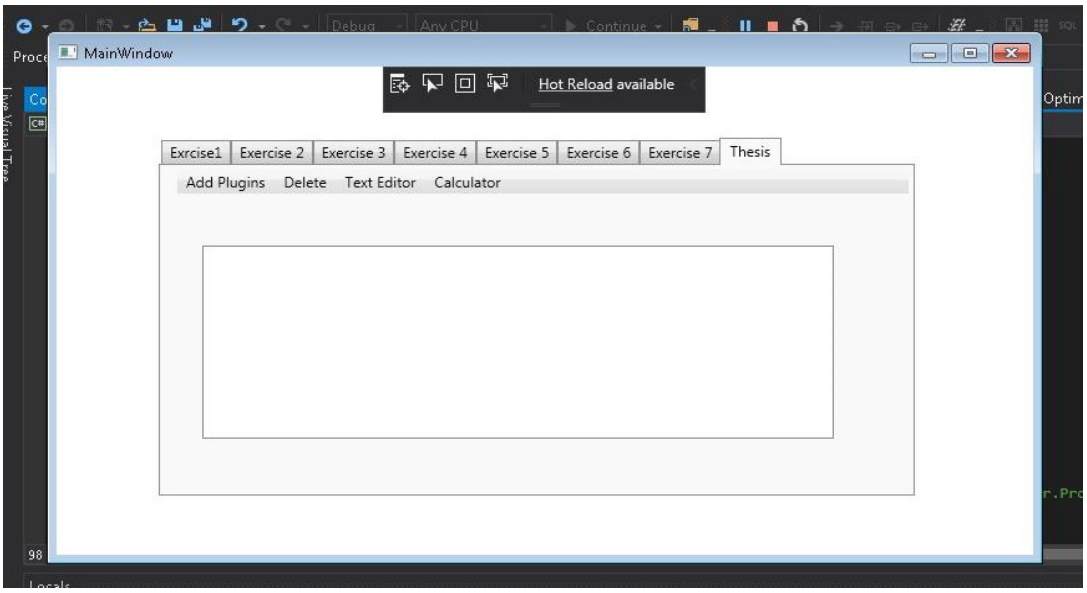

Figure 8

Libraries

The MEF library is important in this project too, since it will provide a way to extend the *Simple Calculator* project. So, in this project will be added the references to the libraries of MEF like in the previous one.
Besides these references there are a few more that should be added to the application. The principal libraries that will be included and as a consequence, will be added their references are:

- AvalonEdit, *ICSharpCode.AvalonEdit*
- RoslynPad, *RoslynPad.Editor, RoslynPad.Roslyn*
- NRefactory, *ICSharpCode.NRefactory*

To add their references to the project, is necessary to download them as NuGet and then reference them inside the application. The references are added in the same way that was added the references to MEF library.

Components

Since this is a long and complicated project, it is preferred to divide it in

different components, which are:

- *MainWindow*, ShowMessage is the application. It is a *Windows Application* in Visual Studio and the target framework is the *.NET Framework 4.7.2.*
  The main file of this application is the *MainWindow.xaml,* that is composed by *MainWindow.xaml.cs.*
- *CodeEditor,* which is a Windows Form file and contains all the code for the implementation of the code editor.
- *Folding directory,* which contains two classes that are used for the code folding in the code editor.

- **MainWindow**

In the application ShowMessage the principal file is the MainWindow. MainWindow is a Windows Application that is composed by *MainWindow.xaml*, that contains the design of the window and the *MainWindow.xaml.*cs that contains the code of the application.

In the file *MainWindow.xaml* are declared all the different tab items, the different buttons, textboxes etc., that make part of the output window of the application.
As it was shown before, there are specified eight different tab items, one for each exercise that is done, and the final one is the application that is treated in this thesis.
What does the application window contain?
As it is shown on the *Figure 9,* the application window is composed by a *Menu,* on the top of this window, in which are specified three different buttons:

- Add Plugins
- Delete
- Text Editor

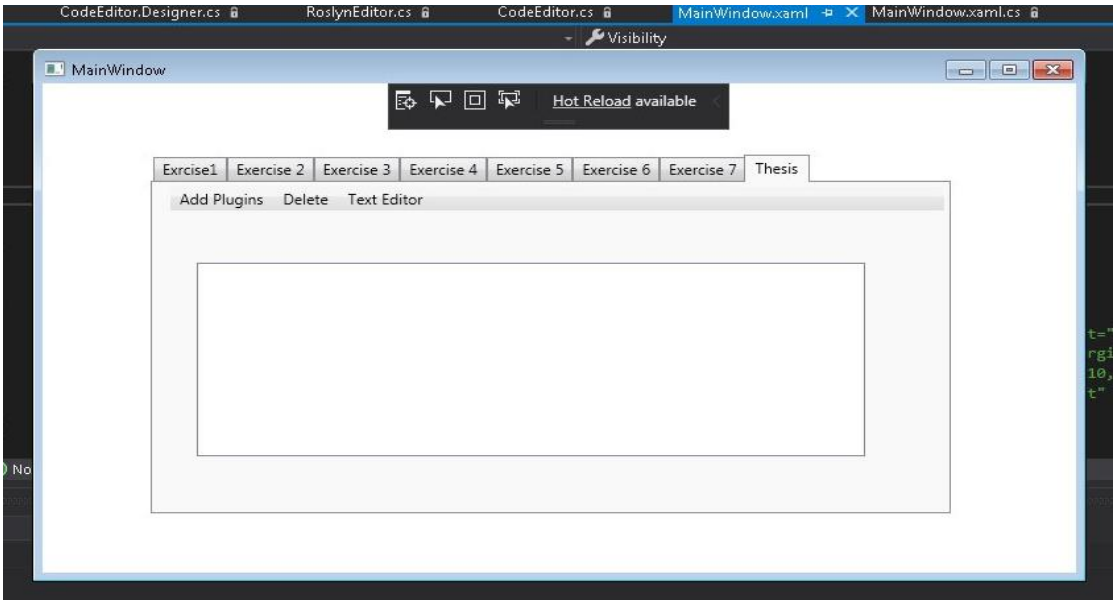

Figure 9

Add Plugin

Add Plugin is a button on the Menu of this application.

```
<MenuItem Header="Add Plugins" Click="AddClickDll" Visibility="Visible"/>
```

The functionality of this button is to add new plugins on the application and in this way extending the project Simple Calculator by adding new libraries to the catalog and then, show the plugins that are added on the text box that is in the main window.

How does this button work?

First of all, when it is clicked it shows a Dialog Window, which is a directory from the user computer, and gives the possibility to the user to choose either a library or a zip folder which contains a library.

```
var openFileDialog = new OpenFileDialog();
openFileDialog.Multiselect = true;
```

This operation permit to the application to open a directory from the user

computer and setting the Multiselect at true, permit to the user to choose more than one file simultaneously as is shown on the *Figure 10*. The directory that will be open by this method is specified and in this case, it is specified at a directory where are saved the plugins.

```
string root = @"C:\Users\fismailaj\source\repos\Plugins";
```

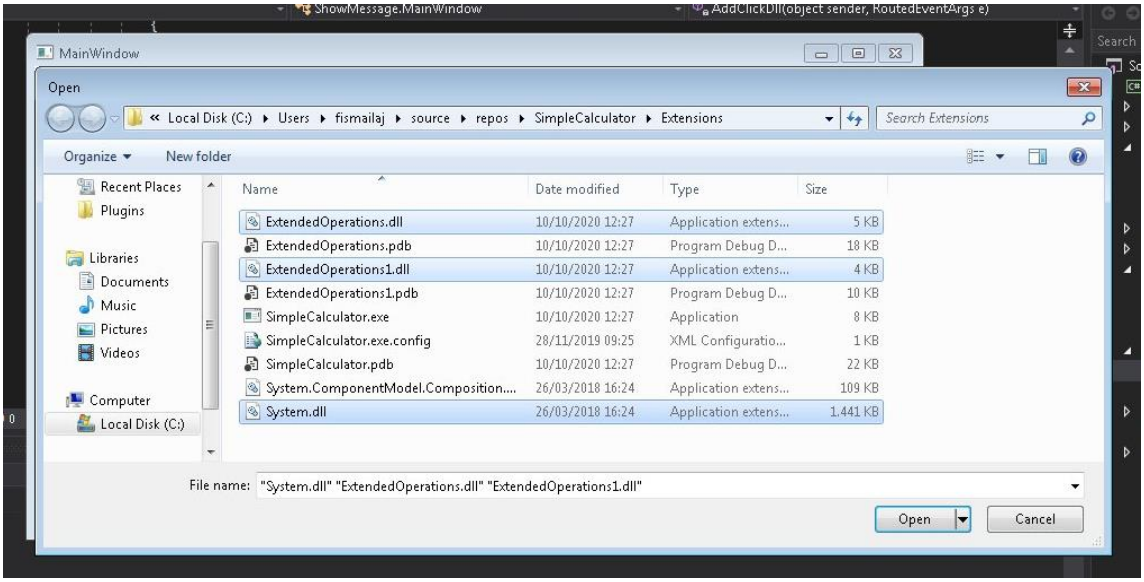

Figure 10

When this window is open, the user can choose the library or the libraries that wants to add to the project. Ones the libraries are selected, and the user clicks at Open, the libraries are uploaded to the catalog. If it is the case that the user chooses to add a zip folder, the application does the unzip of the folder, search on it if there is a library by searching the extension *.dll* and then performs in the same way that it does when the user selects directly one or more libraries.

```
Program.catalog.Catalogs.Add(new DirectoryCatalog(destinationFile));
System.Windows.MessageBox.Show("File is been reupload");
```

When the library is uploaded, a message is shown by the application, which says that the files are uploaded to the catalog and then, to the central text box are shown the new libraries that are uploaded.
The library that is added is shown on form of a tree, where at the top there is

the name of the library that was uploaded and the childes of this tree shows the plugins that this library has. To make this possible, a *TreeView* is declared.

```
<TreeView x:Name="TreeView3" Margin="30,64,60,43"
Visibility="Hidden"  Grid.ColumnSpan="2"/>
```

To add all the component on the *TreeView*, the *ShowPlugins* method is implemented, which add to the header of the tree, the name of the library added, and as chiles, the names of the plugins (extensions) that this library contains.

```
foreach (var part in SimpleCalculator.Program.catalog)
{
  foreach (var dll in Directory.GetFiles(Path.Combine(root,
  filename), "*.dll"))
  {
    if (Trim(part.ToString()) == Trim(Path.GetFileName(dll)))
    {
       if (!TreeViewContains(part.ToString()))
           treeitem.Items.Add(new TreeViewItem() {
                Header = part.ToString()
           });
     }
   }
 }
TreeView3.Items.Add(treeitem);
```

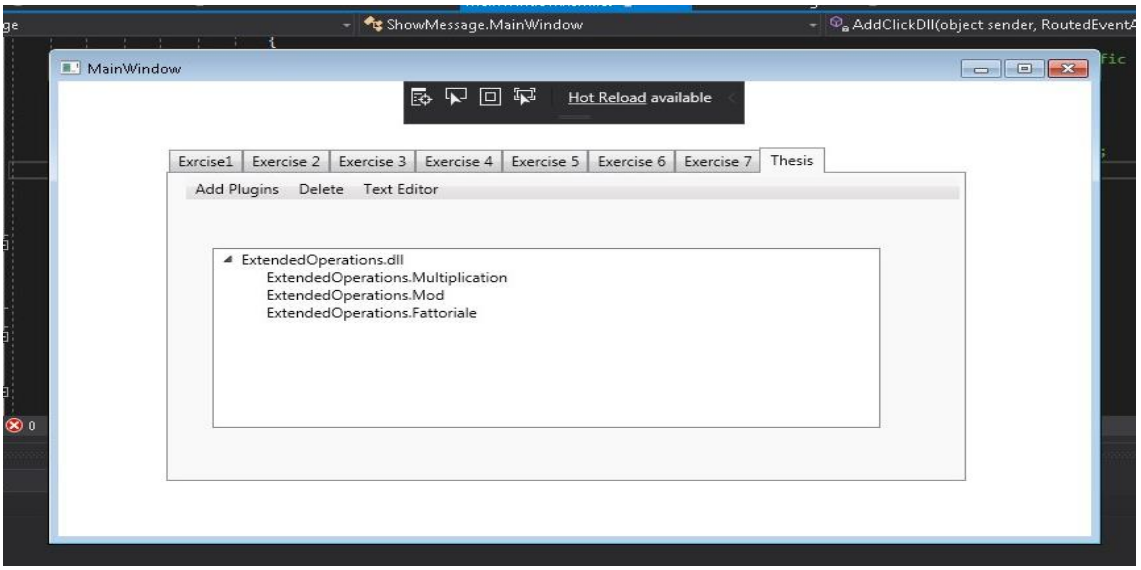

Figure 11

In the Figure 11, is shown that a library named *ExtendedOperations.dll* is uploaded to the catalog. This library contains three different extensions in it, which are the *Multiplication, Mod* and *Fattoriale.*

Delete

This button, as the Add Plugin button, is part of the menu of the application. The button *Delete* is deigned to perform the unloading of the libraries from the application. When the user wants to delete a specific plugin from the application, all the library that contains that plugin will be deleted, since it is not possible to delete just one plugin that is contained on a library. Also, when the user selects a library to unload, all the plugins of that library will be unloaded from the project.
When the user selects a specific plugin and then clicks to delete, the parent of that plugin is searched, then the library is removed from the catalog and also tree of the library is deleted from the text box.

```
SimpleCalculator.Program.catalog.Catalogs.Remove(pointer.Current);
TreeView3.Items.RemoveAt(TreeView3.Items.IndexOf(treeViewItem));
```

In order that the text box always shows the current state of the catalog, the catalog is updated. When the application is runed, the catalog is updated and then are showed the libraries that are available on the catalog at that specific time.

- **Code Editor**

The third button that is on the *MainWindow* is the Text Editor. When the user clicks at this button, the code editor window is opened. The code editor is implemented on the CodeEditor file.
CodeEditor.cs is a Windows Form file in Visual Studio. It contains the CodeEditor.Designer.cs file, in which is implemented the design of the code editor.

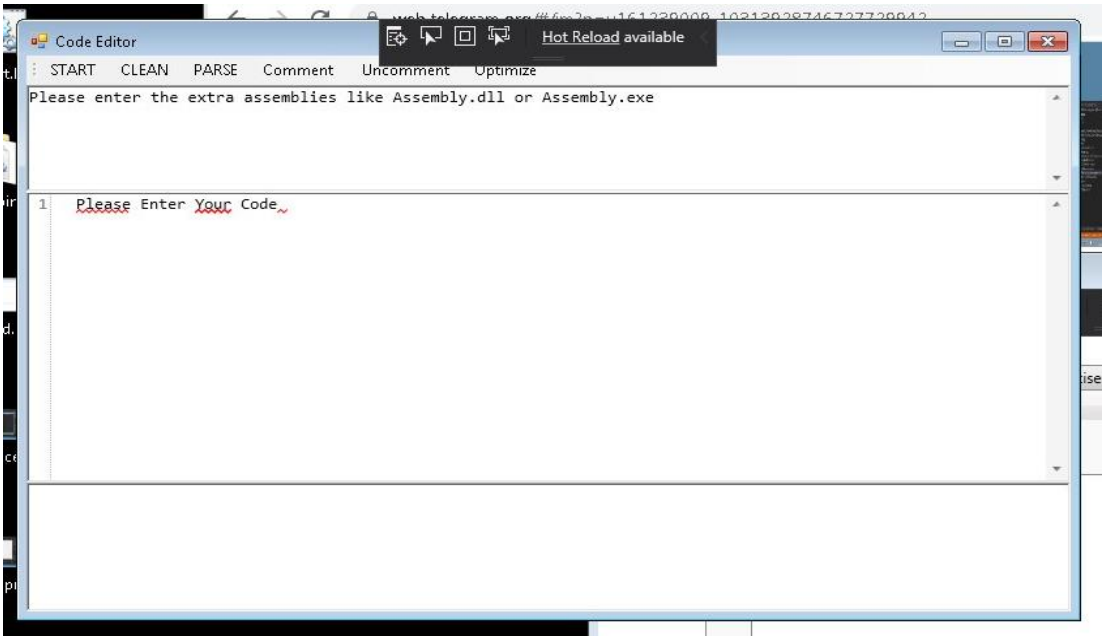

Figure 12

In the *Figure 12* is shown the design of the code editor. Its window is divided in three parts, the part where is declared the required assemblies, the central part where dedicated to code implementation and the third part which shows the output.
The first part, where are declared the extra assemblies, is implemented just the code highlighting, since the assemblies that should be added are specific and it was thought that the code competition is not necessary.
To make possible the code highlighting, AvalonEdit is used.

```
public TextEditor editor;
private void Editor()
{
     ElementHost host1 = new ElementHost();
     editor = new TextEditor();
     ...
     editor.SyntaxHighlighting =
     HighlightingManager.Instance.GetDefinition("C#");
     ...
}
```

Where *TextEditor* is an istance of the class *ICsharpCode.AvalonEdit.TextEditor.*

The second part is the part dedicated to the code implementation, where the user can implement his libraries. As specified in the requirements of this application, this part should support syntax highlighting, error highlighting, code completion etc. To make all these possible, the RoslynPad is used. This part is declared as:

```
private RoslynCodeEditor roslynCodeEditor;
public void InitializeEditor(string sourcecode)
{
      roslynCodeEditor = new RoslynCodeEditor();
     ...
}
```

After having declared the RoslynCodeEditor in this part of the code editor, it is important to initialize it, in such a way that all the features of RoslynPad can be available to use on this application.

In the *Figure 13,* is represented one of the features that RoslynPAd offers to the application that is the code completion. When the user starts typing something, it automatically shows the different possibilities that the user may be write down. As in this case, when at the user starts typing *us,* as that is not written anything else, the two available things are the using statement and ushort. But if the user is declaring a new variable, as it is represented in the Figure 14, the autocompletion is not shown since it is a new variable that

is going to be declared and no suggestions are available in this case.

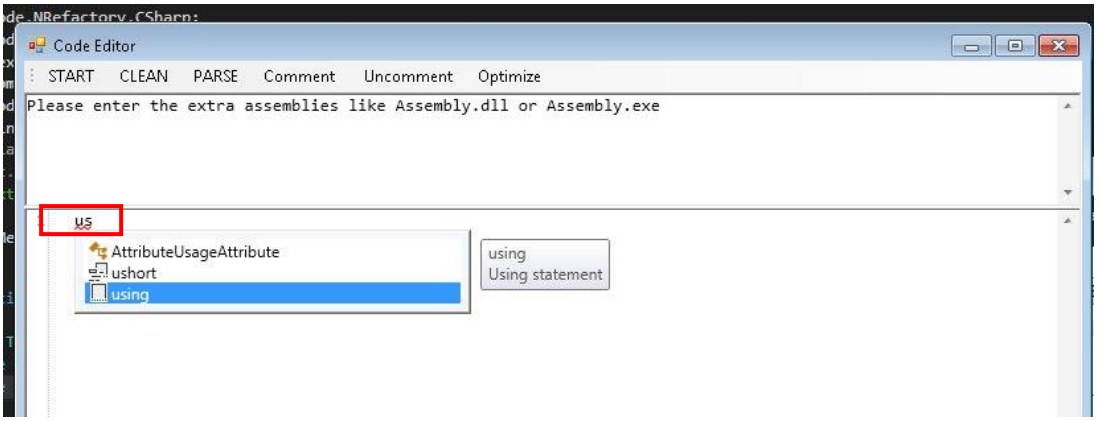

Figure 13

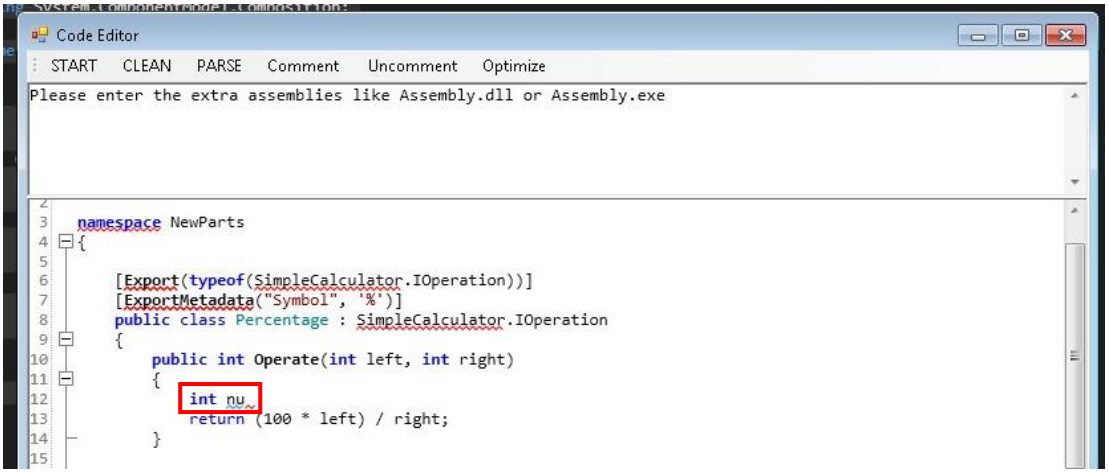

Figure 14

From this code editor it is required to generate a new library, *.dll,* and not an executable, .exe. So, the code of the library will be as the code of plugins that were specified in the SimpleCalcualtor project. In the Figure 15, is shown how a new plugin can be implemented and how the assemblies are specified.

In this case, there are some errors, but this is because what is defined is not a code for an executable, but just a script that specifies a plugin for the SimpleCalculator project.

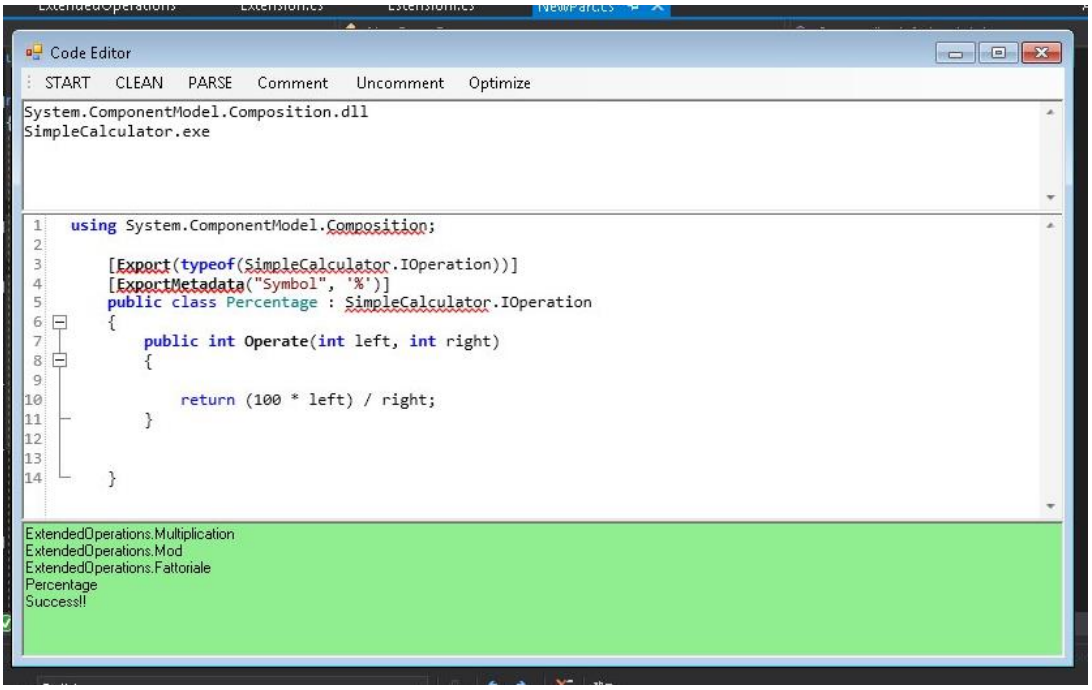

Figure 15

At the top of this window there is a menu with some buttons that has different functionalities.

Start

The start button is the most important one. It makes possible to compile the code and execute it on run time. To make it possible the *CSharpCodeProvider* is used.

```
CSharpCodeProvider codeProvider = new CSharpCodeProvider();
```

There are specified the compile parameters, the basic reference assemblies and are also added the extra assemblies that are specified when the library is implemented.
When all the components are added, the compile is done.

```
CompilerResults compilerResults =
codeProvider.CompileAssemblyFromSource(compilerParameters,
text);
```

If the *compilerResults* does not have errors, it means that the compilation is done successfully and at the output is given the available plugins on the catalog. As can be see from the *Figure 15*, the compilation is done successfully, the output color is green, and it shows the plugins available at that moment on the catalog.

At the same time, the result of the code editor are added to the text box of the application in form of a tree view, showing the name of the new library added and the plugins that that library contains (*Figure 16*).

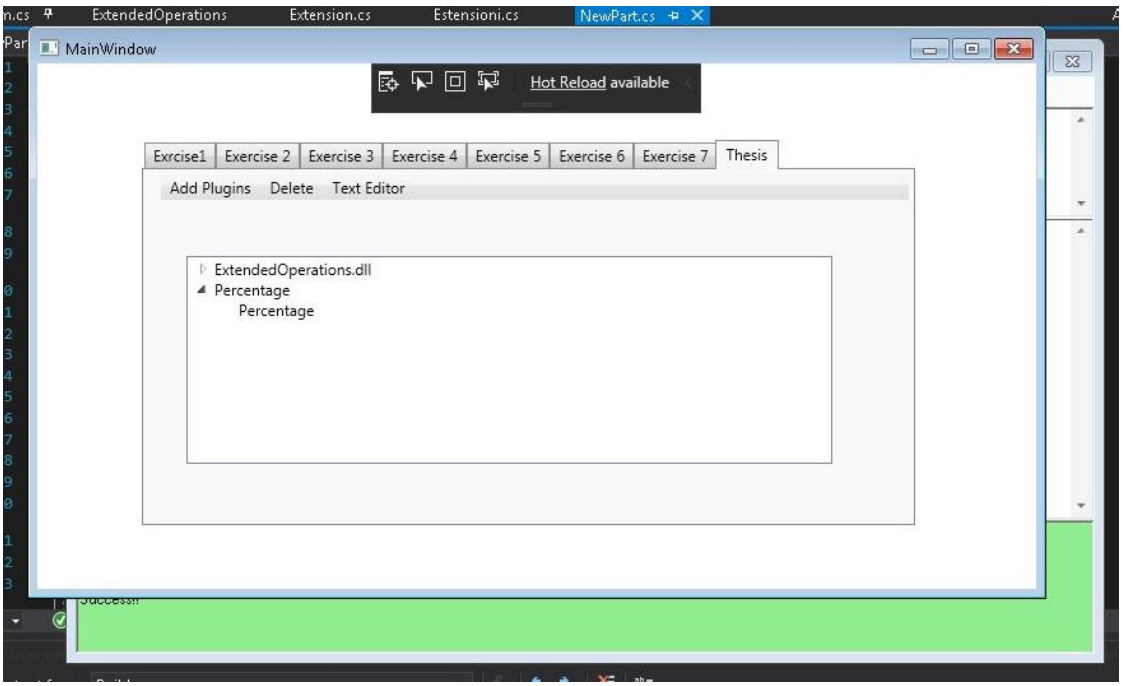

Figure 16

Clean

It is another button present on the code editor window. This button cleans all the three parts of the code editor window, letting the user to write the new code again from the beginning.

Comment/Uncomment

These two buttons give the possibility to the user to comments and

uncomment parts of code. It is necessary to select the part of code that want to comment or uncomment and then click to the Comment/Uncomment button.

//comment

```
roslynCodeEditor.SelectedText = "/*"+roslynCodeEditor.SelectedText+"*/";
```

//uncomment

```
roslynCodeEditor.SelectedText = roslynCodeEditor.SelectedText.Replace("/*", "");
roslynCodeEditor.SelectedText = roslynCodeEditor.SelectedText.Replace("*/", "");
```

Parse

The code editor offers another feature that is the possibility to build the Parse tree. To realize this feature, it is used the NRefactory library, that helps on building the parse tree of the implemented code. To build the parse tree it needs to build first all the nodes of the tree and make all together to create the final parse tree.

When a node of parse tree is selected it shows on the code which part of it is representing. As in the Figure 17, the node *class:CSharpTokenNode*, shows on the code the statement *class*.

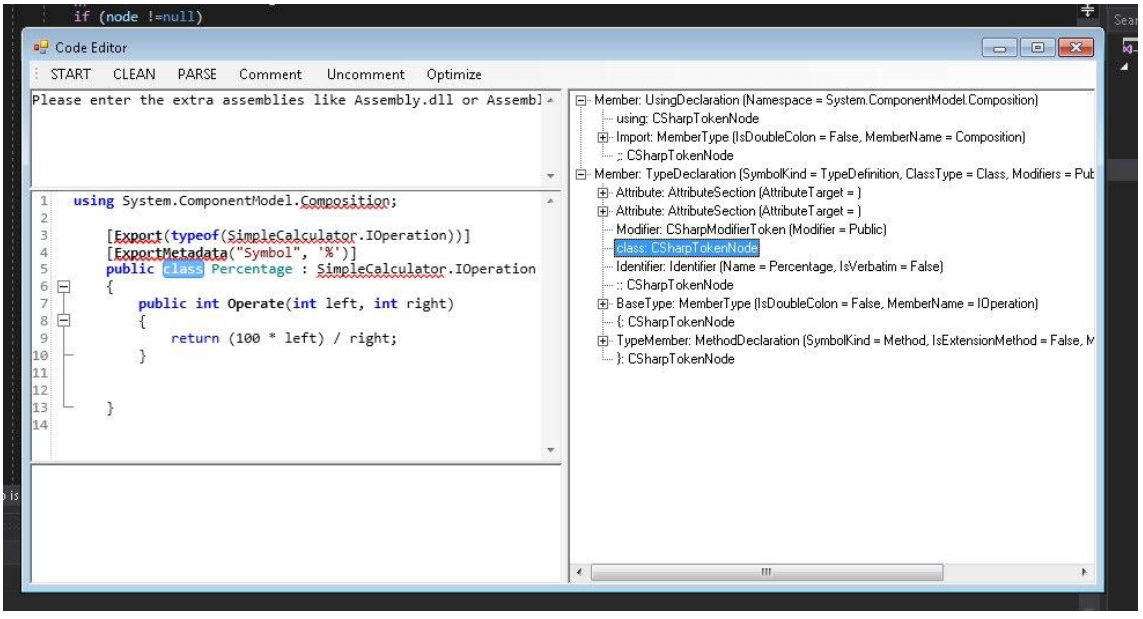

Figure 17

- **Folding directory**

To make the code more legible the folding is necessary. In this case, it is preferred to do the folding just for the curly brackets. In fact, in the folding directory there are two different files, *AbstractFoldingStrategy.cs* and *BraceFoldingStrategy.cs*.

In the first one is implemented the folding strategy, that is used in the second file, which implements a specific folding strategy that in this case is the brace folding.

Brace folding strategy means that the folding is done regarded the curly braces that are present in the code.

As can be seen from the *Figure 18,* when a curly bracket is declared, in the left there appears a '-' symbol that can be clicked and the code regarded to that block is folding. When the code is folding a '+' symbol appears.

```
FoldingManager foldingManager;
AbstractFoldingStrategy foldingStrategy;

...

     foldingStrategy = new BraceFoldingStrategy();

...
```

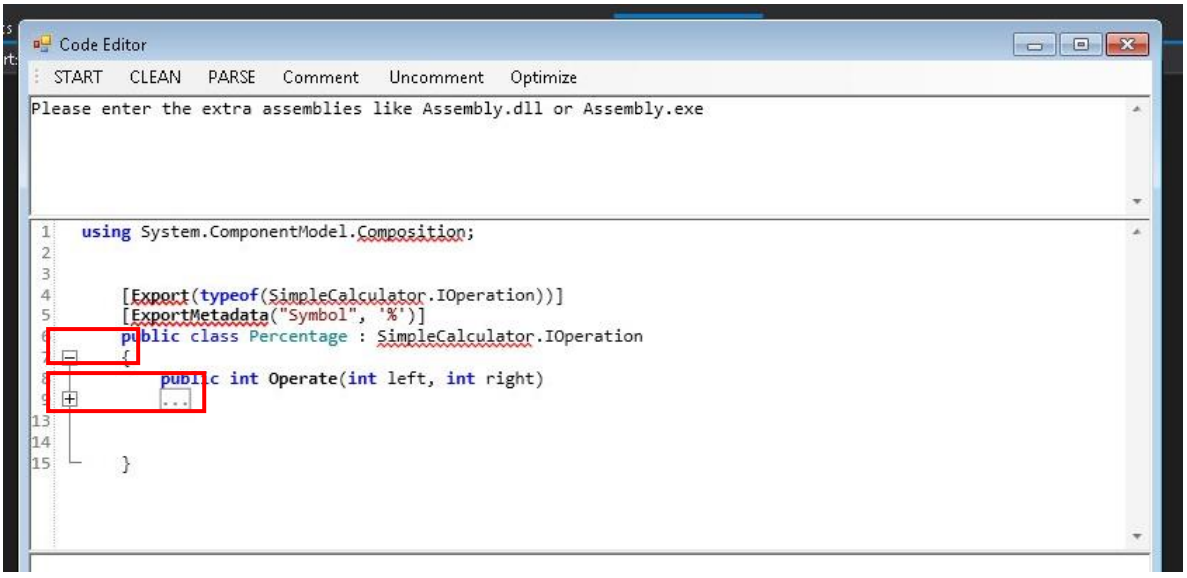

Figure 18

- **Optimize**

Optimization is an important issue in this thesis. What is going to be stated in this section is a research on how an algorithm can be optimized and why.

When an algorithm is implemented, the first question that is posed is if this algorithm can be optimized to obtain a better one.
But what is algorithm or code optimization?
Often, code optimization is defined with code that perform better. Code optimization is said to be the writing or rewriting of code so an application minimize its CPU time, so the time of execution, it uses the least possible of the memory or disk space or it makes a better use of the additional cores.
But there is also another definition of code optimization, thought as writing less code.

Before starting with optimization, let see what premature optimization is. When it is starting coding, it is also thought to develop an optimized version of the application that is going to be developed. But often happens to think about optimization before starting, having a premature optimization which is the attempt to optimize performance when first coding an algorithm or before profiling pinpoints where it makes sense to optimize. It is right to say that premature optimization is bad, because performance-optimized code is not the first priority when coding, it is not above correctness, clarity, testability and so on.

Let's start this section with two rules by Jackson's code optimization rules:

- Don't do it
- (*For experts only!*) Don't do it yet.

Don't do it, is meant for all of them programmers that think to optimize before having cleared what they are going to implement. It is also true that more a programmer knows, the more they will be tempted to premature optimize. The best way is to leave the optimization out for a while since it is time to do

it. When is the time to optimize? There are two types of optimization that can be consider, higher-level optimization that can be done earlier in the project and lower-level optimization that should be left for later.

So, a good way to optimize is in this order:

- Architecture
- Algorithms
- Assembly

Algorithm and Assembly, data structure, are the most effective place where to optimize because is there where performance is concerned. But, it should be kept in mind that sometimes is the architecture what determines what algorithms and data structure can be used.

Architecture

When is developing a project, the architecture is the most expensive parts to change and this is a place where makes sense to optimize at the beginning. It is partially about anticipating to what degree the project will need to be scaled and in what ways. Architecture is high-level, so it is difficult to say what should be done and what shouldn't without narrowing the focus to specific domains and technologies.

Algorithms and Assembly

When the code is been implemented and it works, is really the time to let it on *'Don't do it'?* Will you optimize?
You are right. The next rule is, for experts only, *Do not do it yet.*
Check the standard library, check the framework's ecosystem that solves the problem already. Check for the concepts that you are dealing, it may have pretty standard and well-known names, so making some research will save time on implementing. When you do not have found nothing that can solve the problem, think about designing it so that would be simple to explain to a

new programmer.
When it is arriving at this point and the algorithm is implemented, it is time to benchmark.
What is benchmark? It is time to make some test if the code needs to be optimized or it is fine what is been obtained. It needs to set all the algorithmic benchmark baselines and once it is done, create and benchmark end-to-end tests that covers the most real-world usage of the application created.

Having done all this, it can say that is time to optimize and use some profiling tools. Profiling is always the first thing that needs to do in the process of optimizing code for performance. It is used the profile to look more for functional level profiling than statement level profiling, because the goal is to find out which algorithm is the bottleneck. When the bottleneck is found, is time to optimize being confident that the optimization is worth doing. The optimization can be proved if it was effective or less thanks to the baseline benchmarks that were set along the way.

Overall techniques

Stay high level as long as possible. At the whole algorithm level, one technique is strength reduction. When it is reducing loops to formula, be careful with the use of mathematical operations. It may happen that, what it is thinking to be strength reduction, at the end results it is not. Whether it is used a formula or replaced a loop-based algorithm with another loop-based algorithm, it needs to measure the difference. It may be that gets better performance simply by changing the data structure, such as hash, which looks a bit messier to work with, but is the superior search time worth it over an array?

Micro optimization

When the system's functionality is done, it is time to go ahead with statement level profiling. This level of optimization is kind of trade off between maintainability and clarity.
There are various categories of code optimization techniques:

- Caching

Is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. The use of a cache also allows for higher throughput from the underlying resource, by assembling multiple fine grain transfers into larger, more efficient requests.

- Loop optimization

Loop optimization is transforming a loop in such a way that improves performance without changing the output. Because many transformations can come at the cost of readability or maintainability, but this is often left to a compiler.
Optimizing loops is important in compilation, because loop, in particular the inner loops, are responsible for much of the execution's times of many programs. But what is a loop? The key to a loop is a back edge in the control flow graph from a node *l* to a node *h* that dominates *l. h* is called the header node of the loop. The loop itself then consists of the nodes on a path from *h* to *l*. When loops are nested, the inner loops are optimized before the outside loops, because the inner loops are likely to be executed more often and it could move computation to an outer loop from which it is hoisted further when the outer loop is optimized and so on.
Hoisting loop invariant computation is significant, optimizing computation which changes by a constant amount each time around the loop is probably even more important. This variable is called basic induction variable. The opportunity to optimize arises from derived induction variables, which are variables computed from basic induction variables.

- Memory hierarchy optimization

Besides these techniques there are some general advises to take in consideration when optimize:

- Do not reuse a variable for multiple distinct purpose
- Do not hand unroll loops
- Do not use macros and inline functions without knowing why

5. Conclusion

At the beginning, the aim of this thesis was to create a small application with an integrated code editor able to extend a project, loading new plugins or creating them directly from the code editor at run time.

5.1 Evolution

Throw-out this thesis is explained all the work that is done for the development of the application and the result is a small application able to extend a project by adding new plugins in it or creating them directly using the code editor. All the plugins are added at run time and also the code editor let the user to create the new libraries at run time. This code editor lets the user compiling the code, creating the new plugin and extending the functionalities of the project without having to save it before.
There are also stated all the research done during the development of the thesis about the code optimization.
The application is an easy to use tool, which is developed using the Visual Studio Code IDE that make the code authoring easier, the C# language and the MEF library which gives the possibility to develop extensible projects.

5.2 Future Work

The aim was to create an application able to extend a project including also the code editor.
There are also some other features that can be added at this application to make it more complete.

- Debugging: adding to the code editor the possibility to debug the

written code. This will make the code editor more complete and also make the code authoring easier.

- Optimization: another features that can be added to this application is the possibility to optimize the code implemented by the user and giving him the new optimized algorithm in the same code editor window or in another one. Then, the user can choose rather to use his code or the new optimized one.

6. Bibliography

[1] Microsoft: *Visual Studio*, http://www.visualstudio.com/

[2] Microsoft: *.NET Compiler Platform ("Roslyn")*, MSDN, http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

[3] Grunwald D.: *Using NRefactory for analyzing C# code*, CodeProject 2012, http://www.codeproject.com/Articles/408663/Using-NRefactoryfor-analyzing-Csharp-code

[4] IC#Code: *ILSpy – .NET Decompiler*, http://ilspy.net/

[5] Microsoft: *Windows Forms*, MSDN, http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx

[6] Microsoft: *Windows Presentation Foundation*, MSDN, http://msdn.microsoft.com/en-us/library/ms754130.aspx

[7] IC#Code: *AvalonEdit*, http://avalonedit.net/

[9] Community: *AvalonDock*, Codeplex, http://avalondock.codeplex.com/

[10] Code Optimization: https://www.toptal.com/freelance/curse-premature-optimization

[11] Using NRefactory for analyzing C# code: https://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code

[12] Compile code programmatically: https://docs.microsoft.com/en-us/troubleshoot/dotnet/csharp/compile-code-using-compiler

[13] Managed Extensibility Framework: https://docs.microsoft.com/it-it/dotnet/framework/mef/

[14] Debugger visualizer for SharpDevelop IDE: http://artax.karlin.mff.cuni.cz/~konim5am/thesis/MartinKonicek_DebuggerVisualizers.pdf

[15] An IDE for C# scritp development, Jan Pelc, Prague 2015